Under consideration for publication in Formal Aspects of Computing

Explanation by refinement and linearisability of two non-blocking shared-variable communication algorithms

Richard Bornat and Hasan Amjad
Middlesex University, London, UK

Abstract. Simpson and Harris have described multi-slot algorithms implementing a single-place buffer, each operating without explicit hardware synchronisation mechanisms. Conventional refinement and proof techniques have explained *that* these algorithms work, but do not give convincing descriptions of *how* they work. An unconventional refinement process starting from the classic single-variable buffer, using both data and atomicity refinement and drawing information from unsuccessful steps, derives each algorithm. The logic used is RGSep, a marriage of rely/guarantee and concurrent separation logic. Extensive detailed verifications are described. The result is an explanation of how the algorithms work and some pointers to how such algorithms might be devised.

Keywords: separation logic, rely-guarantee, concurrency, proof, refinement, atomicity

1. Introduction

Most work in program verification is, rightly, about practical concerns. At the time of writing there is a great deal of practical progress. Separation logic allows us to begin to deal with pointers, and there has been significant work in dealing with concurrent algorithms. Developments in theorem proving and static analysis techniques have advanced already to the point that automatic tools can guess and verify safety properties of entire programs as large as a million-line operating system. Better still, we can find mistakes in those huge programs, mistakes subtle enough to elude a type system.

In another direction we can use program logic to study programs that don't have mistakes. Many simple to understand algorithms lack formal proof. Developments in program logic mean that we can give some of them straightforward simple proofs, proofs that justify our intuitions of how they work. Simple, clear proofs are beautiful. Beautiful algorithms deserve beautiful proofs.

Correspondence and offprint requests to: Richard Bornat, Department of Engineering and Information Sciences, Middlesex University, LONDON, NW4 4BT, email: R.Bornat@mdx.ac.uk

But there are algorithms which aren't simple, which aren't clear, which we understand with difficulty if at all. If their complexity and intricacy is essential then perhaps they are as beautiful as they can be. But because they are difficult to understand, not very many people will see their beauty. If we could make a beautiful proof of those algorithms, perhaps the proof itself could serve as explanation.

Concurrency is the area of programming where such beasts are the most common, and where it seems we need them: we have to pull tricks to solve tricky problems. This paper is an attempt to use formal reasoning to reveal the beauty of two tricky concurrent algorithms. If it generates understanding along the way then that can be seen as practical; to build a better mousetrap you must understand current technology.

2. Background

Dijkstra [Dij65] defined the first classic concurrency problem, that of two processes/threads communicating through a single shared 'buffer' variable: one writing; the other reading. The difficulty is that writing and reading aren't instantaneous operations, so that if their executions overlap in time, the reader may extract a value from the buffer which is made up of parts of two or more distinct written values. The solution that Dijkstra put forward, use of semaphores to ensure turn-taking, is nowadays challenged by so-called *non-blocking* algorithms in which the processes do not pause for each other but do manage to avoid overlapping reads and writes.

One famous non-blocking algorithm is by Simpson [Sim90], which uses an array of four data slots and some single-bit index variables to control their use. It apparently uses no synchronisation mechanisms and is *wait-free* [Her91], or *asynchronous*: neither reader and writer loops during its operation on the shared buffer. Another is Harris's algorithm [Har04], which is also wait-free but uses only three slots. Simpson's algorithm was devised as an illustration of a particular interprocess communication mechanism in the MASCOT software engineering framework [SJ79, Sim86] and is practically useful, for example in connecting a sensor to a data collector, but its economy and novelty attracted those keen to extend the range of formal proof of algorithms. Harris's algorithm was devised in response to the challenge to construct an equally capable algorithm with fewer than four slots. Both algorithms are maddeningly concise and lack accessible explanation of how they work, why they never go wrong and how they achieve communication.

There can be no doubt about the validity of these algorithms: they do achieve reliable and effective communication. Simpson's algorithm can relatively easily be model-checked: Simpson himself justifies it in those terms [Sim92, Sim97a] as does Rushby [Rus02]. There are also formal treatments which don't use model checking [HP02, Hen03, PHA04] and our own contribution in [BA10]. Harris's algorithm could equally easily be model-checked and is formally proved by us in [BA10]. If a beautiful proof of an algorithm is one which elegantly and simply justifies our intuitions of how it works, none of these proofs, including our own, is beautiful. We lack intuitions which can be justified, and none of these proofs provides insight: all of them elaborate a great deal of detail in order to justify the claim that good things happen and bad things don't. We need more than that to understand what is going on, either to be able to reproduce the algorithm or to build new similarly intricate algorithms.

Refinement is a formal approach which might be said to concentrate upon explanation. It normally starts with a highly abstract specification which is implemented by a magical program and modifies specification and program step by step together, making the specification less abstract and the program less magical, to produce an implementable algorithm. Each step preserves the correspondence between specification and program and is small enough to be intuitively justifiable. If the end result is a known algorithm like Simpson's or Harris's, the process can reasonably be said to stand as an explanation of why the algorithm is written as it is and how it really works. This is the approach followed by Abrial [AC06, Abr08] and Jones and Pierce [JP08, JP09].

We believe that these explanations start from too low a level, conceding too much in their specification to the final algorithm. In this paper we attempt an alternative approach, starting with a simple but difficult-to-implement algorithm which has a simple specification, and working by steps of atomicity refinement and data refinement towards the algorithm we want to explain. Jones and Pierce present but reject this simpler specification, claiming that it does not properly deal with overlapping concurrent read and write operations; we argue in §5.2 that this objection is invalid. Abrial's specification is in terms of interleaving of sequences, as is ours, but he too finds it necessary to discuss the properties of overlapping executions, which we are able to deal with implicitly.

3. Outline

In order that the structure of our argument is not obscured by its details, we give an overview. Our goal is a buffer algorithm whose only atomic actions are single-bit reads and writes. We start with a single-variable buffer algorithm (fig. 2), which has a fairly straightforward specification, is easy to verify, but requires an implementation of atomic shared-variable access. We make a data refinement to replace the single variable by a two-slot array plus a single-bit index variable. This generates an algorithm (fig. 11) which fails atomicity refinement, because the refined reader is not guaranteed to read values in the sequence that they are written. An alternative data refinement, to two slots plus two index variables, gives fig. 19, which we atomicity-refine into fig. 31. A second data refinement divides each of the buffer slots into a pair (fig. 32), using an extra two-element array of single-bit indices. Atomicity refinement then gives us Simpson's 1990 four-slot algorithm (fig. 40).

A second alternative refinement to two slots plus an indexing variable and a danger-signalling variable gives fig. 43, but because there is a loop in the reader the algorithm is not wait-free. Data refinement on the danger signal produces a three-slot algorithm which is once again wait-free (fig. 47), and atomicity refinement then produces what is essentially Harris's algorithm (fig. 52).

4. Our logic

Our verifications use rely-guarantee [Jon83] in a presentation derived from Vafeiadis [Vaf07]. This presentation was devised to support a combination of concurrent separation logic [O'H07] and rely-guarantee. We don't use separation or ownership transfer, unlike our earlier effort in [BA10]; nevertheless we find that the Vafeiadis presentation style has some advantages.

The quintuple $C\,\mathbf{sat}\,(P, R, G, Q)$ states that any execution of C, in a state which satisfies P, under interference at most R, will cause at most interference G and, if it terminates, will produce a state which satisfies Q. P and Q are assertions describing sets of states; R and G are state to state relations. The rule of parallel composition is straightforward:

$$\frac{C1\,\mathbf{sat}\,(P1, R \cup G2, G1, Q1) \quad C2\,\mathbf{sat}\,(P2, R \cup G1, G2, Q2)}{(\ C1\ \|\ C2\ )\,\mathbf{sat}\,(P1 \wedge P2, R, G1 \cup G2, Q1 \wedge Q2)}\ \text{parallel composition} \tag{1}$$

Each thread must suffer the interference of the environment and its partner together; the environment must suffer the interference of both. Pre and postconditions must satisfy the requirements of each thread separately.

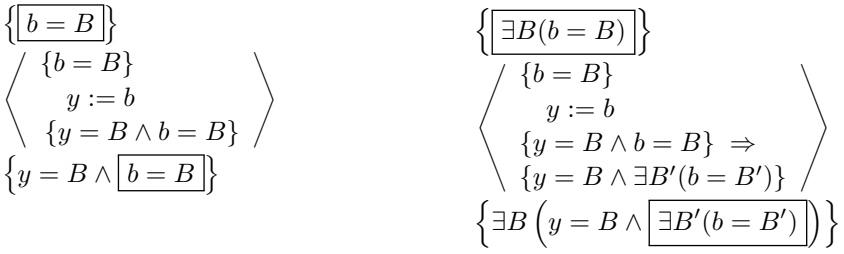
It is with the treatment of shared store and interference that our presentation differs from the standard. Our assertions are about state, as in Hoare logic: that is, postconditions do not describe a relation with the precondition. Assertions are written in two parts: assertions about shared store are boxed; assertions about thread-local store are unboxed. Boxed assertions may only refer to shared store, unboxed to local store. Atomic instructions may access shared and local store; non-atomic instructions may only access local store. Actions $P \rightsquigarrow Q$ state how an atomic instruction can change shared store from state P to state Q; since both P and Q describe sets of states, actions are relations between sets of states and we can form relies and guarantees as usual.

The presentation hinges on the treatment of atomic instructions, which may access shared store. The instantaneous effect of an execution $\langle C\rangle$ – by convention we use angle-bracketing to indicate atomicity – is dealt with by the rule (adapted from [VP07])

$$\frac{\{P \wedge sP\}\ C\ \{Q \wedge sQ\} \quad sP \rightsquigarrow sQ \subseteq G}{\langle C\rangle\,\mathbf{sat}\,(P \wedge \boxed{\exists \vec{x}(sP)}, R, G, \exists \vec{x}\left(Q \wedge \boxed{sQ}\right))}\ \text{atomic instruction} \tag{2}$$

P is the local-store precondition. sP is the shared-store precondition. P can't mention shared store, and sP can't mention local store. The corresponding postconditions are Q and sQ. The Hoare-logic triple $\{P \wedge sP\}\ C\ \{Q \wedge sQ\}$ shows the effect of the command on the combination of local and shared store. The *action* of $\langle C\rangle$ is its effect on shared store, and is written $sP \rightsquigarrow sQ$.[1] That action, which is itself a relation

[1] In RGSep proper, we have to be concerned about *precise* assertions in order that resource splits are unambiguous. In this



(a) Mid stability (b) Full stability

Fig. 1. Two alternative verifications of an atomic read

between states, must be contained in the guarantee G of the maximum effect of the thread on shared store. Because local (unboxed) assertions can only mention local store, and shared-store (boxed) assertions can only mention shared store, we use existential quantification to allow the local postcondition to refer to values retrieved from the shared store by $\langle C \rangle$. In our treatment this happens only occasionally: see, for example, figures 8 and 29.

4.0.1. The treatment of stability

This rule is deliberately crafted to deal with the *instantaneous* effect of the atomic instruction $\langle C \rangle$ and says nothing about stability. If sP holds in the instant before the instruction is executed, then sQ holds immediately afterwards. This is a variation of Vafeiadis's *mid-stability* presentation, which we have adopted because it makes certain properties of atomic instructions easier to understand.

Consider, for example, the atomic instruction $\langle y := b \rangle$ where y is a local variable but b is shared and subject to interference – i.e. R is such that the environment can interfere by rewriting b. It's obvious that the assignment retrieves the value of b at some instant, but that the value it retrieves isn't a stable value of b, because under those circumstances there is no stable value of b. Our treatment of the command is shown in fig. 1(a): by default upper-case identifiers stand for universally-quantified values, and the verification shows clearly that whatever the value of b in the instant before execution, that same value is in y and in b in the instant afterwards.

By contrast, if we were to insist, as in the *full stability* presentation, that pre and postconditions must make *stable* assertions about shared store, invariant against actions contained in R, we should have to reason as in fig. 1(b). The stable precondition states that there is a value in b. The internal Hoare-logic verification (note that the rule strips away the existential in the antecedent precondition) shows that this value is copied to y; we can see that at least for an instant y and b contain the same value. But then the requirement that the shared part of the postcondition be stable means that we must immediately weaken that shared part – that is, $b = B$ – to allow for the interference of the environment. We do that by introducing an existential, and the link between y and b is lost. The deduction is sound and not misleading: there isn't a stable connection between the values of y and b after the atomic command executes. But something is obscured, which is that the command retrieves a value which is, if only instantaneously, in the shared variable b.

We can't ignore stability, and we require it in sequences of actions. If we consider the sequence $C1; C2$ and we assert

$$\{P \wedge \boxed{sP}\}\ C1\ \{P' \wedge \boxed{sP'}\}\ C2\ \{Q \wedge \boxed{sQ}\} \tag{3}$$

then we can say that sP is the instantaneous shared-store precondition of the sequence and sQ its instantaneous shared-store postcondition. But the intermediate shared-store assertion sP', which holds from the instant after the execution of $C1$ until the instant before the execution of $C2$, holds for an interval and must

paper that care is unnecessary: we can see how to separate assertions P and sP because one refers only to local store, the other only to shared store, and there is no means of resource transfer.

therefore be stable. We impose the requirement of stability in the sequence rule

$$\frac{C1\ \mathbf{sat}(P \wedge \boxed{sP}, R, G, P' \wedge \boxed{sP'}) \quad sP' \text{ stable under } R \quad C2\ \mathbf{sat}(P' \wedge \boxed{sP'}, R, G, Q \wedge \boxed{sQ})}{(C1; C2)\ \mathbf{sat}\ (P \wedge \boxed{sP}, R, G, Q \wedge \boxed{sQ})}\ \text{sequence} \tag{4}$$

Where necessary we use the rule of consequence to provide the necessary stability.

4.0.2. Triples rather than quintuples

Quintuple assertions $C\ \mathbf{sat}\ (P, R, G, Q)$ are unwieldy. For simplicity and clarity in presentation of proofs, since we are dealing with algorithms in which there are only two threads and a single parallel composition, we keep R and G implicit: the writer's R is the reader's G and vice-versa. Our versions of the atomic instruction rule and the instruction sequence rule are then

$$\frac{\{P \wedge sP\}\ C\ \{Q \wedge sQ\} \quad sP \rightsquigarrow sQ \subseteq G}{\left\{P \wedge \boxed{\exists \vec{x}(sP)}\right\}\ \langle C \rangle\ \left\{\exists \vec{x}\left(Q \wedge \boxed{sQ}\right)\right\}}\ \text{atomic instruction (abbreviated)} \tag{5}$$

$$\frac{\left\{P \wedge \boxed{sP}\right\}\ C1\ \left\{P' \wedge \boxed{sP'}\right\} \quad sP' \text{ stable under } R \quad \left\{P' \wedge \boxed{sP'}\right\}\ C2\ \left\{Q \wedge \boxed{sQ}\right\}}{\left\{P \wedge \boxed{sP}\right\}\ C1; C2\ \left\{Q \wedge \boxed{sQ}\right\}}\ \text{sequence (abbreviated)} \tag{6}$$

4.1. Atomicity refinement

Suppose we have an atomic sequence $\langle A; B \rangle$ with a verified specification

$$\left\{P \wedge \boxed{sP}\right\}\ \langle A; B \rangle\ \left\{Q \wedge \boxed{sQ}\right\} \tag{7}$$

and that we have shown the environment's assertions stable under the action $sP \rightsquigarrow sQ$.

We wish to refine the original atomic instruction into a sequence of two atomic instructions $\langle A \rangle; \langle B \rangle$. Verification of the refined sequence with the same precondition as before requires a step of stabilisation between $\langle A \rangle$ and $\langle B \rangle$:

$$\begin{array}{l} \left\{P \wedge \boxed{sP}\right\}\ \langle A \rangle\ \left\{P' \wedge \boxed{sP'}\right\} \\ sP' \Rightarrow sP''; sP'' \text{ stable under } R \\ \left\{P' \wedge \boxed{sP''}\right\}\ \langle B \rangle\ \left\{Q' \wedge \boxed{sQ'}\right\} \end{array} \tag{8}$$

The refined version generates *two* actions $sP \rightsquigarrow sP'$ and $sP'' \rightsquigarrow sQ'$, and a postcondition $Q' \wedge \boxed{sQ'}$ which may not be the same as $Q \wedge \boxed{sQ}$.

For this to be an atomicity refinement we require that the environment cannot tell the difference, and that the sequential context into which the original fitted accepts the refined version. The first of these conditions is straightforward: we must show that the environment's assertions are as stable under the two refined actions as they were under the single original. The second is more subtle: sQ and sQ' must stabilise to the same assertion, and the local precondition of anything which followed $\langle A; B \rangle$ and now follows $\langle A \rangle; \langle B \rangle$ must be implied by Q' as it was by Q.

In practice in our atomicity refinements Q and Q' will often be identical and $sP \rightsquigarrow sP'$ will often be the identity action or deal with state that isn't mentioned in the environment's assertions. All of this will make the verification of atomicity refinement straightforward.

We note that atomicity refinement is *not* local reasoning: we have to assess, in all the other threads, the interference of the refined sequence on their previously-stable assertions. In this paper that isn't much of a problem, because there is only one other thread, so not much to check, and because we use a communication invariant to simplify a lot of stability arguments.

$$\begin{array}{l}\mathsf{data}\ b = \mathit{null}\ \mathsf{in}\\ \left(\begin{array}{l}\ldots\ \text{calls of } \mathit{write}(\_)\ \ldots\\ \mathit{write}(\mathsf{data}\ w)\ \hat{=}\ \langle b := w\rangle\end{array}\ \Big\|\ \begin{array}{l}\ldots\ \text{calls of } \mathit{read}(\_)\ \ldots\\ \mathsf{data}\ \mathit{read}(\,)\ \hat{=}\ \mathsf{data}\ y\ \mathsf{in}\langle y := b\rangle; \mathsf{return}\ y\ \mathsf{ni}\end{array}\right)\\ \mathsf{ni}\end{array}$$

Fig. 2. A single-variable buffer, with angle-bracketed atomic instructions

5. The atomic single-slot algorithm

Fig. 2 is a buffer algorithm using a single shared variable b. Two processes/threads are composed in parallel: one makes calls $\mathit{write}(W)$ which writes the value W to the buffer; the other makes calls $\mathit{read}(\,)$ which retrieve the current value from the buffer. The algorithm is idealised in the sense that it depends upon an unexplained execution mechanism: the angle-bracketed buffer accesses are required to occur atomically, which ideally means instantaneously and never at the same instant. Because execution is instantaneous and instants infinitesimally small, neither operation has to wait for the other and there is no need of a scheduler. A $\mathit{write}(w)$ call on one side of the parallel composition executes $\langle b := w\rangle$, instantaneously writing to the buffer; a $\mathit{read}(\,)$ call on the other side executes $\langle y := b\rangle$, instantaneously reading. It's obvious that this is a *wait-free* algorithm: the writer can always operate without waiting for the reader, and vice-versa, and every procedure call will always terminate.

Instantaneous non-overlapping execution is impossible in practice. We might settle for interleaving of competing pseudo-atomic executions, which isn't difficult to implement: we could, for example, make each atomic instruction a critical section by bracketing it with $\mathrm{P}(m)$ and $\mathrm{V}(m)$ where m is a global binary semaphore. But such an implementation complicates the issue by introducing questions of fairness and liveness: an unfair scheduler might prevent one or other process from ever executing its critical section, and it could be difficult to prove that that never happened. We don't, therefore, want to consider how to implement atomicity. It will be the unexplained mechanism in our idealised algorithm, and removing it the goal of our refinement.

The important properties of this algorithm have to do with communication: it is potentially *lossy* and *stuttering*, which might be thought to be drawbacks, but it also exhibits *freshness* and *coherence* [Sim90], which are definitely advantages. Lossiness and stuttering are essential characteristics of wait-free single-place-buffer algorithms: the writer can overwrite the buffer at any time, so some written values may never be read (lossiness); and the reader may read the buffer at any time, so some values may be read more than once (stuttering). But each time the reader executes $\langle y := b\rangle$, it is certain to get the latest value that the writer has placed there (freshness);[2]. and the writer always writes a complete value with $\langle b := w\rangle$, so the buffer only ever contains complete values, and the reader therefore always retrieves a complete value with $\langle y := b\rangle$ (coherence).

Our description of the properties of the algorithm in fig. 2 – stuttering, lossiness, coherence, freshness – is essentially Simpson's, and we take this algorithm to be the specification of a single-place buffer. Single-place buffer algorithms must do some or all of what this algorithm does, we assert, or else they aren't single-place buffer algorithms. But there is an objection to considering this algorithm as a starting place in a refinement process. First we note that, so far as communication is concerned, it defines a sequential interleaving of $\langle b := w\rangle$ and $\langle y := b\rangle$ operations. A sequential specification may not sufficiently constrain the refined algorithm: atomicity refinement adds interactions and therefore potentially adds communication behaviours. Jones and Pierce [JP08, JP09] object that a sequential specification like ours, with single atomic actions in writer and reader, must be inadequate because it doesn't sufficiently describe behaviour of the refined algorithm in which reads and writes are sequences of actions rather than atomic actions, and therefore can overlap. We deal with that objection in §5.2, after we have annotated the specification to permit formal description and refinement.

5.1. A communication invariant

The writer in fig. 2 produces a sequence of values, and the reader consumes some or all of those values. To facilitate reasoning about the sequences of values written and read, we introduce a version of the algorithm,

[2] If it reads before the first write it will see the initial value *null*, but we treat that as if it was the first value written.

$$\begin{array}{l}\mathsf{data}\, b = \mathit{null}\, \mathsf{in}\, \textbf{data* } \boldsymbol{ws = .null, rs = .}\ \mathsf{in} \\ \left(\begin{array}{l} \ldots \text{ calls of } \mathit{write}(\_)\ \ldots \\ \mathit{write}(\mathsf{data}\, w) \mathrel{\hat{=}} \langle b := w; \boldsymbol{ws := ws.w}\rangle \end{array}\right) \Vert \left(\begin{array}{l} \ldots \text{ calls of } \mathit{read}(\,)\ \ldots \\ \\ \mathsf{data}\, \mathit{read}(\,) \mathrel{\hat{=}} \mathsf{data}\, y\, \mathsf{in} \langle y := b; \boldsymbol{rs := rs.y}\rangle; \mathsf{return}\, y\, \mathsf{ni} \end{array}\right) \\ \mathsf{ni}\, \mathsf{ni}\end{array}$$

Fig. 3. A single-variable buffer, with darkened auxiliary annotations

in fig. 3, with added assignments to auxiliary data-sequence variables ws and rs to record those sequences. Because we use angle brackets for atomic instructions, we can't use them for sequences; so '$s.x$' appends the single value x to sequence s; '$.x$' is the singleton sequence containing x; and '$.$' is the empty sequence.

We capture the first part of our requirements for communication with the assertion

$$\lfloor rs \rfloor \preccurlyeq ws \wedge ws_\Omega = b \tag{9}$$

– the read sequence destuttered is a subsequence of the written sequence, and the value in the buffer is the last element of the written sequence. We shall see that this assertion is invariant under the actions of either thread from fig. 3. The first part of the invariant deals with stuttering (we consider $\lfloor rs \rfloor$ rather than simply rs), lossiness (subsequence $\preccurlyeq$ rather than equality) and coherence (the values in rs come only from ws). The second part helps with consideration of freshness: it's obvious, given $ws_\Omega = b$, that our annotated specification algorithm always assigns the last element of ws – the latest written value – to y.

But what is obvious must be stated formally. Our specification of freshness is that if B is the initial value in the buffer when the reader begins, we require when it ends that it has assigned a fresh value to y:

$$\exists B' (y = B' \wedge \lfloor rs.B.B' \rfloor \preccurlyeq ws) \tag{10}$$

– the value in y is part of the written sequence and no earlier in that sequence than B. This is a somewhat informal specification because it requires context (B is the value in the buffer when the reader starts) before it can be interpreted. It is obviously true of the algorithm in fig. 3 (because $B = B' = ws_\Omega$ after $y := b$); in later refinements we shall see that this formulation emerges naturally from our verifications, which provide the necessary context. Note that this specification allows the writer to add to the sequence between the time that the reader starts (the instant when the reading process calls $\mathit{read}(\,)$) and the time that it retrieves a value (the instant that the $\mathit{read}(\,)$ procedure executes $y := b$ in our idealised algorithm). Again, this is essentially Simpson's definition of freshness.

When we come to refine our algorithm we shall have a multi-*variable* implementation, and read and write operations will no longer contain a single atomic action, but we still shall have a single-*place* buffer – only a single buffered latest value, and no queuing. We shall, therefore, hold on to lossiness and stuttering; and we must surely hold on to coherence and freshness.

5.2. Is it a sufficient specification?

Atomicity refinement, by converting atomic sequences into sequences of atomic actions, adds interaction behaviour: actions which were instantaneous and necessarily alternated become extended over time and can therefore overlap. By allowing reads and writes to overlap we might add communication behaviours, and then (9) and (10) might not specify the behaviour of the refined algorithm. Jones and Pierce [JP08, JP09] insist that it is necessary to constrain the behaviour of the refined algorithm as part of the specification, and that it is not enough to describe, as we do, a simple sequential interleaving of reads and writes. We assert, however, that our refinements add no communication behaviours and that therefore our specification is adequate. The basis of our argument is that all the refinements we consider (with the exception of the faulty fig. 18) are *linearisable* [HW90] and therefore every concurrent execution is equivalent to a sequential execution.

We have chosen to wrap our atomic communication operations in procedure definitions because it isolates the sites at which we have to perform refinement. Procedure calls take time, so there will be an interval between the start of a $\mathit{write}(w)$ call and execution of the atomic $\langle b := w\rangle$ operation it contains; during that interval one or more $\mathit{read}(\,)$ procedure calls might execute their $\langle y := b\rangle$ operations. The same applies to the interval between execution of $\langle b := w\rangle$ and the return of the $\mathit{write}(w)$ call, and the reverse applies to the similar intervals in the execution of $\mathit{read}(\,)$ procedure calls, during each of which there can be $\mathit{write}(w)$

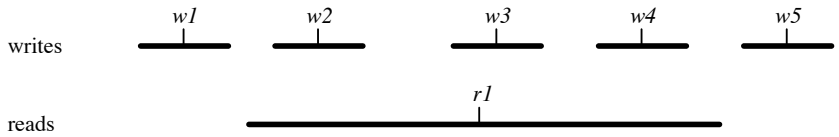


Fig. 4. Concurrent execution with write procedure calls overlapping a read

Fig. 5. Sequential execution equivalent to fig. 4

calls for various different values of w. So there is a sense in which even this simple algorithm generates concurrent overlapping executions. Fig. 4 shows an example in which several writes overlap a single read, with a horizontal line showing the duration of a procedure call and a vertical spike the execution of each atomic operation.

But the concurrency can be simplified: the *write*(_) calls which execute their atomic operation before $\langle y := b \rangle$ might as well have taken place before the *read*( ) call started, and those which execute their atomic operation after $\langle y := b \rangle$ might as well have waited until it returns. Fig. 5 shows the sequential execution equivalent to fig. 4 – note that in pushing writes aside we haven't reordered their executions.

Not much seems to have happened: the order of procedure executions in fig. 5 is just the order of the atomic executions in fig. 4, and indeed that is the point. The sequential and concurrent executions are equivalent, in the sense that reader and writer can't tell the difference between them, just because the spike-orders are the same. We don't even need to know where the atomic-execution spikes occur within the procedure call execution interval: it's enough to observe that for every concurrent (overlapping) execution there is an equivalent (non-overlapping) sequential execution which gives just the same order of spikes.

We have illustrated the principle of linearisability with an example. If concurrent procedure call executions have the character that they each contain exactly one instant – an atomic execution, called a *linearisation point* – before which the procedure call has had no effect (write) or has not fixed upon a value (read), and after which the effect is visible (write) or the value is decided (read), then they can always be rearranged into an equivalent sequential execution. The effects of all concurrent executions are contained within the effects of all sequential executions: nothing is left out.

All the algorithms we consider (with the exception once again of the faulty fig. 18) have just this character. In write procedures the linearisation point is either the buffer assignment or one of the index-variable updates which follow it, but is always annotated with an assignment to ws. The linearisation point for reads is trickier to identify, because it is often in the writer rather than in the reader. We prefer to appeal to the notion of freshness and to observe that if our freshness condition $y = B' \wedge \lfloor rs.B.B' \rfloor \preccurlyeq ws$ holds, where B is the buffer value when the reader begins, then

either $B = B'$ and the linearisation point is the instant the reader began; the write procedure call which wrote B effectively occurred before the read call; and all subsequents writes effectively occur after the read call.

or $B \neq B'$ and B' was written after B by an overlapping write procedure; the linearisation point of the read is the linearisation point of that write; that procedure call, and all the writes which precede it, effectively occurred before the read; and all subsequent writes effectively occur after the read.

We prefer not to make linearisation points explicit in our arguments, since to do so would simply repeat the argument above in local detail. What is important is that our algorithms, up to and including Simpson's and Harris's algorithms (fig. 40 and fig. 52) are linearisable refinements of fig. 3. The behaviours of the algorithm in fig. 3 encompass the behaviours of the refinements so far as communication is concerned. There is no need to say more about concurrent executions. Jones and Pierce's objection is mistaken; we have an adequate specification.

$$\left\{w = W \wedge \boxed{b = B \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws}\right\}$$
$$\left\langle \begin{array}{l} \{w = W \wedge b = B \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws\} \\ \quad b := w \\ \{w = W \wedge b = W \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws\} \\ \quad \boldsymbol{ws := ws.w} \\ \{w = W \wedge b = W \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws\} \end{array} \right\rangle$$
$$\left\{w = W \wedge \boxed{b = W \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws}\right\}$$

Fig. 6. Mid-stability verification of single-variable writer

$$\left\{\boxed{b = B \wedge rs = RS \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws}\right\}$$
$$\left\langle \begin{array}{l} \{b = B \wedge rs = RS \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws\} \\ \quad y := b \\ \{y = B \wedge b = B \wedge rs = RS \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws\} \\ \quad \boldsymbol{rs := rs.y} \\ \{y = B \wedge b = B \wedge rs = RS.B \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws\} \end{array} \right\rangle$$
$$\left\{y = B \wedge \boxed{b = B \wedge rs = RS.B \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws}\right\}$$

Fig. 7. Mid-stability verification of single-variable reader

5.3. Verification of the single-variable algorithm

Our verification of the single-variable writer is shown in fig. 6. Just before execution of $\langle b := w; \ldots\rangle$ the buffer contains a value B, the value to be written is W, and the written sequence is $WS.B$.[3] We observe that if $\lfloor X \rfloor \preccurlyeq Z.B$ then $\lfloor X \rfloor \preccurlyeq Z.B.W$, which enables us to conclude that $\lfloor rs \rfloor \preccurlyeq ws$ still holds after the update of ws. Ignoring the value of rs which doesn't change during this atomic execution and writing WS in place of $WS.B$, we extract the action

$$b = B \wedge ws = WS \rightsquigarrow b = W \wedge ws = WS.W \tag{11}$$

Mid-stability verification tells us what the atomic instruction inside a $write(w)$ procedure call does, but we might like to know the stable pre and postcondition of the procedure call itself. The reader can change rs, which is mentioned in both assertions but, as we shall see, if it does so it preserves $\lfloor rs \rfloor \preccurlyeq ws$. So both pre and postcondition of fig. 6 are stable, and they are the stable pre and postcondition of $write(w)$.

We verify the single-variable reader in fig. 7. We observe that if $\lfloor X \rfloor \preccurlyeq Z.B$ then $\lfloor X.B \rfloor \preccurlyeq Z.B$, which enables us once again to verify that $\lfloor rs \rfloor \preccurlyeq ws$ is invariant. Dropping irrelevant detail, this verification

[3] By convention we write quantified variables in upper case and in the absence of explicit quantification there is an implicit universal quantification over the entire verification. So B, W and WS are universally quantified in fig. 6.

$$\left\{\boxed{\exists B, WS\,(\, b = B \wedge rs = RS \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws\,)}\right\}$$
$$\left\langle \begin{array}{l} \{b = B \wedge rs = RS \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws\} \\ \quad y := b \\ \{y = B \wedge b = B \wedge rs = RS \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws\} \\ \quad \boldsymbol{rs := rs.y} \\ \{y = B \wedge b = B \wedge rs = RS.B \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws\} \Rightarrow \\ \{y = B \wedge rs = RS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge ws_\Omega = b\} \end{array} \right\rangle$$
$$\left\{\exists B \left(y = B \wedge \boxed{rs = RS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge ws_\Omega = b} \right)\right\}$$

Fig. 8. Full-stability verification of single-variable reader

$$\left\{\boxed{b = B \wedge rs = RS \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws}\right\}$$
$$\langle y := b \rangle$$
$$\left\{y = B \wedge \boxed{b = B \wedge rs = RS \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws}\right\} \;\Rightarrow\; \text{(stability)}$$
$$\left\{y = B \wedge \boxed{rs = RS \wedge \lfloor RS.B \rfloor \preccurlyeq ws \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge ws_\Omega = b}\right\}$$
$$\langle \boldsymbol{rs := rs.y} \rangle$$
$$\left\{y = B \wedge \boxed{rs = RS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge ws_\Omega = b}\right\}$$

Fig. 9. Atomicity refinement of single-variable reader

$$\begin{array}{l}\mathsf{data}\; b = null \;\mathsf{in}\; \mathbf{data*}\; \boldsymbol{ws = .null, rs = .}\; \mathsf{in} \\ \left(\begin{array}{l} \ldots \\ write(\mathsf{data}\; w) \;\hat{=}\; \left\langle \begin{array}{l} b := w; \\ \boldsymbol{ws := ws.w} \end{array} \right\rangle \end{array}\right. \;\Big\|\; \begin{array}{l} \ldots \\ \mathsf{data}\; read(\,) \;\hat{=}\; \mathsf{data}\; y\; \mathsf{in} \\ \qquad \langle y := b \rangle; \\ \qquad \langle \boldsymbol{rs := rs.y} \rangle; \\ \qquad \mathsf{return}\; y \\ \quad \mathsf{ni} \end{array} \left.\begin{array}{l}\\ \\ \\ \\ \end{array}\right) \\ \mathsf{ni}\;\mathsf{ni}\end{array}$$

Fig. 10. Atomicity-refined single-variable single-place buffer

generates the action

$$b = B \wedge rs = RS \rightsquigarrow b = B \wedge rs = RS.B \tag{12}$$

In the instantaneous postcondition y holds the same value as the last element in the written sequence – that is, the reader has retrieved the freshest possible value. This is definitely a mid-stability deduction: both pre and postcondition are unstable since the writer can alter both b and ws. The full-stability verification is shown in fig. 8, in which it is only possible to see that the reader has retrieved a fresh value if we look at the deduction inside the angle brackets. This enables us to see the stable pre and postcondition of a $read(\,)$ procedure call, for what it's worth.

5.4. Atomicity refinement of the single-variable algorithm

We can't do atomicity refinement on the writer, because if we separate the updates of b and ws we cannot keep $ws_\Omega = b$ invariant. But we can atomically refine the reader's atomic operation into $\langle y := b \rangle; \langle \boldsymbol{rs := rs.y} \rangle$. The verification in fig. 9 uses the same precondition as fig. 7. In the postcondition of the first step we see freshness just as before: $y = b = ws_\Omega$. In the stability weakening step, required in the sequence rule (6) and allowing in this case for the fact that the writer can alter b and add to ws, that association is lost. But the fact that B is a valid appendage to rs is stable under the writer's action (11), so we retain it. In the postcondition we see that the value of b at the instant we begin is the value of y at the instant we end; otherwise the postcondition is a stabilised version of the postcondition of fig. 7, which fulfils the second requirement of atomicity refinement. The action of $\langle y := b \rangle$ is the identity action, which cannot destabilise anything; the action of $\langle \boldsymbol{rs := rs.y} \rangle$ is

$$rs = RS \wedge \lfloor RS.B \rfloor \preccurlyeq ws \rightsquigarrow rs = RS.B \tag{13}$$

which does not destabilise the writer's pre or postcondition since it only writes to rs in circumstances which preserve $\lfloor rs \rfloor \preccurlyeq ws$, which fulfils the first condition of atomicity refinement. We have an atomicity refinement, shown in fig. 10.

6. Refinement to two slots

We need atomicity in the single-variable algorithm because reader and writer compete for access to the same variable. If the buffer variable could be refined by two variables to give a multi-*variable* but still single-*place*

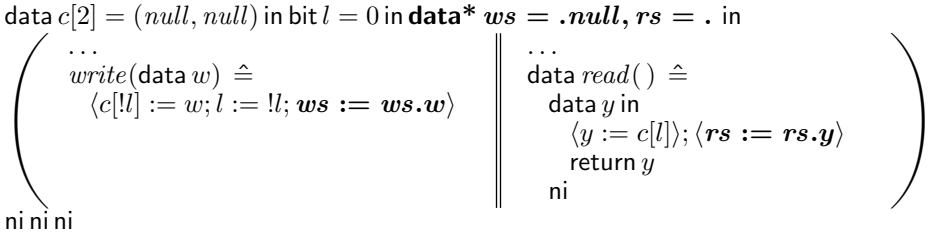


Fig. 11. Two slots, writer chooses

$$\Big\{w = W \wedge \boxed{l = L \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B}\Big\}$$
$$\langle c[!l] := w; l := !l; \boldsymbol{ws} := \boldsymbol{ws.w}\rangle$$
$$\Big\{w = W \wedge \boxed{l = !L \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B \wedge c[!L] = W}\Big\}$$

Fig. 12. Data refinement of fig. 6 to two slots (writer chooses)

buffer, we might hope that reader and writer could operate in different variables at the same time, achieving overlapping executions without compromising coherence. Following [Sim90] we use a two-element array c[4] and a single-bit index variable l to point to the position – the *slot* – containing the last-written value. In assertions we replace $b = E$ by $c[l] = E$. The refined invariant is

$$\lfloor rs \rfloor \preccurlyeq ws \wedge c[l] = ws_\Omega \tag{14}$$

We can easily refine $\langle y := b\rangle$ to $\langle y := c[l]\rangle$, but refining $\langle b := w\rangle$ isn't so immediate. There are two straightforward possibilities:[5]

$$\langle c[l] := w\rangle \tag{15}$$
$$\langle c[!l] := w; l := !l\rangle \tag{16}$$

The first writes directly into $c[l]$; the second does not write to $c[l]$ but by manipulating l it changes the referent of $c[l]$ and so achieves $c[l] = w$. The second operates in two steps, which will be important when we come to atomicity refinement. For now we note that a refinement which used only the single-step (15) would use only one of the two slots, which would seem to make the refinement pointless. We investigate first of all, therefore, an algorithm which uses only the two-step (16).

7. A two-slot algorithm – first attempt

Fig. 11 shows the two-slot algorithm which refines $b := w$ with (16). Inspired by Simpson [Sim97a] we call this algorithm 'writer chooses', because the writer decides the slot into which it will write and the reader must follow.

Because everything is atomic we retain coherence; because we have refined $y := b$ by $y := c[l]$; and because we retain $c[l] = ws_\Omega$ we retain freshness. In the writer, data refinement of the pre and postcondition from fig. 6 and introduction of a universally-quantified L so that we can track the value of l gives us the specification in fig. 12, which is easily verified (details omitted) and produces the action

$$l = L \wedge ws = WS \wedge c[L] = B \rightsquigarrow l = !L \wedge ws = WS.W \wedge c[L] = B \wedge c[!L] = W$$

This is a straightforward data refinement of (11). We have had to be explicit about the value of l because it is changed, but otherwise it changes $c[l] = B$ on the left to $c[l] = W$ on the right, just as (11) changes

[4] In our refinements we do not use the long variable names that Simpson and Harris used. Instead we use single-letter names to save space in assertions.

[5] To save space in our programs and assertions we use the C bit-inversion $!E$ rather than $1 - E$ to invert a single-bit value ($!0 = 1 - 0 = 1$, !1=1-1=0).

$$\Big\{\boxed{rs = RS \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[l] = B}\Big\}$$
$$\langle y := c[l] \rangle$$
$$\Big\{y = B \wedge \boxed{rs = RS \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[l] = B}\Big\} \;\Rightarrow\; \text{(stability)}$$
$$\Big\{y = B \wedge \boxed{rs = RS \wedge \lfloor RS.B \rfloor \preccurlyeq ws \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[l] = ws_\Omega}\Big\}$$
$$\langle \boldsymbol{rs := rs.y} \rangle$$
$$\Big\{y = B \wedge \boxed{rs = RS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[l] = ws_\Omega}\Big\}$$

Fig. 13. Data refinement of fig. 9 to two slots (writer chooses)

$$\Big\{w = W \wedge \boxed{l = L \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B}\Big\}$$
$$\langle c[!l] := w \rangle$$
$$\Big\{w = W \wedge \boxed{l = L \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B \wedge c[!L] = W}\Big\}$$
$$\langle l := !l; \boldsymbol{ws := ws.w} \rangle$$
$$\Big\{w = W \wedge \boxed{l = !L \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B \wedge c[!L] = W}\Big\}$$

Fig. 14. Atomicity refinement of two-slot writer (writer chooses)

$b = B$ to $b = W$. In this action $c[L] = B$ appears on both sides, because that slot doesn't change, but we don't need to mention parts of shared store which don't change. We simplify the action, therefore, into

$$l = L \wedge ws = WS \rightsquigarrow l = !L \wedge ws = WS.W \wedge c[!L] = W \tag{17}$$

In the reader, data refinement of assertions gives us fig. 13, which also verifies easily. Because b wasn't mentioned in the single-slot reader's actions the refined version produces the same actions as before: identity action in the first step; (13) in the second step. As before we see freshness, and we have a simply refined postcondition.

The writer is ripe for atomicity refinement. We can separate $c[!l] := w$ from $l := !l$: the reader's assertions depend on the value in $c[l]$, not $c[!l]$; so $c[!l] := w$ won't disturb any stability considerations. On the other hand we can't separate $l := !l$ and the update of ws, because together they preserve $c[l] = ws_\Omega$ but separately they do not. The verification in fig. 14 shows two actions

$$l = L \rightsquigarrow l = L \wedge c[!L] = W \tag{18}$$
$$l = L \wedge ws = WS \wedge c[!L] = W \rightsquigarrow l = !L \wedge ws = WS.W \wedge c[!L] = W \tag{19}$$

neither of which disturbs the invariant: (18) changes only $c[!l]$; and (19) updates both l and ws at the same time so as to preserve $ws_\Omega = c[l]$. Note that the linearisation point is the assignment to l: before that point the reader can't see the effect of the write; after that point it is completed.

We can go further. We shall see that it is an advantage to use atomic instructions which make only a single access to a single shared variable or array element. If we introduce in the writer a local single-bit variable wt (for *w*riter *t*emporary) we can decompose $\langle c[!l] := w \rangle$ into $\langle wt := !l \rangle$; $\langle c[wt] := w \rangle$, and then we can replace $l := !l$ by $l := wt$. The verification is shown in fig. 15. The first step generates the identity action, and the second and third generate (18) and (19) as before.

Something important has happened, indicated by the double-angle-bracketing of the first and third instructions. An instruction which makes a single access – either read or write – to a single-bit shared variable can be called *naturally atomic*. Stores serialise such accesses, and Simpson's algorithm rests on the assumption that they do, and on the assumption that competing store accesses are scheduled fairly, or at least not infinitely unfairly.[6] From now on we shall make the same assumptions, and we defer until §15 the discussion of what happens in modern processor designs with their so-called 'weak memory models'.

[6] Shared-store concurrency always depends on the assumption that the store arbiter is fair. If it isn't fair or fair-ish, there really doesn't seem much point in sharing store.

$$\Big\{w = W \wedge \boxed{l = L \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B}\Big\}$$
$$\langle\!\langle wt := !l \rangle\!\rangle$$
$$\Big\{w = W \wedge wt = !L \wedge \boxed{l = L \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B}\Big\}$$
$$\langle c[wt] := w \rangle$$
$$\Big\{w = W \wedge wt = !L \wedge \boxed{l = L \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B \wedge c[!L] = W}\Big\}$$
$$\langle\!\langle l := wt; \boldsymbol{ws := ws.w} \rangle\!\rangle$$
$$\Big\{w = W \wedge \boxed{l = !L \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B \wedge c[!L] = W}\Big\}$$

Fig. 15. Second atomicity refinement of two-slot writer (writer chooses)

$$\Big\{\boxed{l = L \wedge rs = RS \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B}\Big\}$$
$$\langle\!\langle rt := l \rangle\!\rangle$$
$$\Big\{rt = L \wedge \boxed{l = L \wedge rs = RS \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B}\Big\}$$
$$\Rightarrow \text{ (stability)}$$
$$\Big\{rt = L \wedge \boxed{rs = RS \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[l] = ws_\Omega}\Big\}$$
$$\langle y := c[rt] \rangle$$
$$\Big\{\exists B' \Big(y = B' \wedge \boxed{rs = RS \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[l] = ws_\Omega}\Big)\Big\}$$
$$\langle \boldsymbol{rs := rs.y} \rangle$$
$$\Big\{\exists B' \Big(y = B' \wedge \boxed{rs = RS.B' \wedge c[l] = ws_\Omega}\Big)\Big\}$$

Fig. 16. Unsuccessful atomicity refinement of two-slot reader (writer chooses)

$write(W)$	$\Big\{\boxed{l = L \wedge c[L] = \_}\Big\}$	$read()$
$\langle\!\langle wt := !l \rangle\!\rangle$		$\langle\!\langle rt := l \rangle\!\rangle$
$\{wt = !L\}$		$\{rt = L\}$
$\langle c[wt] := w \rangle$	$\Big\{\boxed{l = L \wedge c[L] = \_ \wedge c[!L] = W}\Big\}$	
$\langle\!\langle l := wt \rangle\!\rangle$	$\Big\{\boxed{l = !L \wedge c[L] = \_ \wedge c[!L] = W}\Big\}$	
...		
$write(W')$		
$\langle\!\langle wt := !l \rangle\!\rangle$		
$\{wt = L\}$		
$\langle c[wt] := w \rangle$	$\Big\{\boxed{l = !L \wedge c[L] = W' \wedge c[!L] = W}\Big\}$	
		$\langle y := c[rt] \rangle$
		$\{y = W'\}$
		...
		$read()$
		$\langle\!\langle rt := l \rangle\!\rangle$
		$\{rt = !L\}$
		$\langle y := c[rt] \rangle$
		$\{y = W\}$

Fig. 17. Writer-chooses reader, after atomicity refinement, can read out of sequence

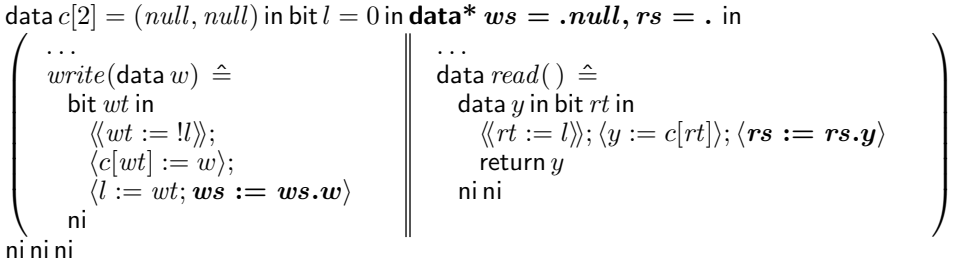


Fig. 18. Two slots, writer chooses, imperfectly refined

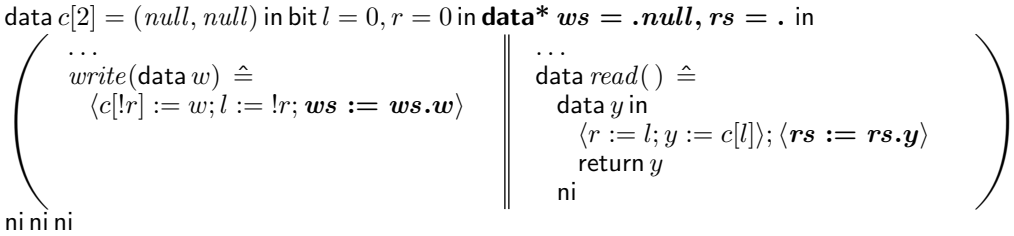


Fig. 19. Two slots, reader chooses

Next we turn to atomicity refinement in the reader. To split $\langle y := c[l]\rangle$ into a sequence of single-shared-access instructions we add a local variable rt and decompose it into $\langle\!\langle rt := l\rangle\!\rangle$; $\langle y := c[rt]\rangle$. Unfortunately we discover, in fig. 16, that this is not an atomicity refinement: after the first step the initial value of l is captured, but the writer can change l, $c[l]$ and ws before the second step. The stability weakening expresses this loss of information, so that in the second step we can't assert that y receives the latest written value: action (13) becomes

$$rs = RS \rightsquigarrow rs = RS.B \tag{20}$$

We can't be sure that this action updates rs so that $\lfloor rs \rfloor \preccurlyeq ws$. We have made the writer's stable assertions unstable. We don't have an atomicity refinement.

This is more than a failure to prove a result: the communication invariant $\lfloor rs \rfloor \preccurlyeq ws$ can actually be violated. The reader can read a value which has not yet been added to the written sequence – $c[!l]$ has been written but l and ws haven't yet been updated – and then come back to read the previous written value, as illustrated in fig. 17. The writer writes W, then writes W' but pauses before it is added to ws; the reader begins before W is added to ws, pauses until W' is about to be added to ws; then the writer pauses, the reader restarts and retrieves W', then begins again and sees W. It's possible to see this as a failure of coherence (the reader retrieves a value W' which is not yet part of the written sequence), but we describe it as an out-of-sequence read (W' before W). In either case we fail to preserve $\lfloor rs \rfloor \preccurlyeq ws$.

This is a real problem which we can't solve by doing atomicity and data refinement in some other order. However we try, we can't prevent the reader sometimes getting one step ahead of the writer-chooses writer. Simpson in [Sim97a, Sim97b] and in private communication argues that this is not a crippling problem in practice – the reader is guaranteed the latest value or at worst the one before, so there is a kind of freshness guarantee – and he goes on to derive a four-slot algorithm which retains the feature. But that's not a true descendant of the idealised algorithm and it won't do for us.

8. Two slot algorithm – second attempt

The writer in fig. 18 constantly switches slots, and therefore can overtake the reader, producing the out-of-sequence read in fig. 17. Simpson's solution is to add a shared index variable r in which the reader indicates the slot $c[r]$ it is reading from, and the writer writes to slot $c[!r]$. The writer still updates l to show which is the latest value written, and the reader still reads that latest value. The refinement/bug-fix has surprising consequences.

$$\Big\{w = W \wedge \boxed{l = L \wedge r = R \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[l] = B}\Big\}$$
$$\langle c[!r] := w; l := !r; \boldsymbol{ws := ws.w}\rangle$$
$$\Big\{w = W \wedge \boxed{l = !R \wedge r = R \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[l] = W}\Big\}$$

Fig. 20. Data refinement of fig. 6 to two slots (reader chooses)

$$\Big\{\boxed{l = L \wedge rs = RS \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B}\Big\}$$
$$\langle r := l; y := c[l]\rangle;$$
$$\Big\{y = B \wedge \boxed{l = r = L \wedge rs = RS \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B}\Big\}$$
$$\Rightarrow \text{ (stability)}$$
$$\Big\{y = B \wedge \boxed{r = L \wedge rs = RS \wedge \lfloor RS.B \rfloor \preccurlyeq ws \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[l] = ws_\Omega}\Big\}$$
$$\langle \boldsymbol{rs := rs.y}\rangle$$
$$\Big\{y = B \wedge \boxed{r = L \wedge rs = RS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[l] = ws_\Omega}\Big\}$$

Fig. 21. Data refinement of fig. 9 to two slots (reader chooses)

In fig. 19 we have backtracked, for reasons of explanation, on the atomicity refinement of the writer-chooses algorithm and gone back to atomic buffer accesses in both reader and writer. The reader reads from $c[l]$, but it also writes in a shared variable r the index of the slot it is reading from, and the writer always writes in the opposite slot. Inspired by Simpson [Sim97a] we call this algorithm 'reader chooses'. Two slots are not enough to guarantee coherent operation without using atomic instructions, so it isn't a complete solution, but it does avoid the out-of-sequence problem.

An obvious property of the algorithm is that the writer only ever alters l so as to make $l = !r$ and the reader only ever alters r so as to make $r = l$. That is, $l = r$ is stable under the reader's actions, and $l \neq r$ under the writer's. These facts are preserved in refinement to the four-slot algorithm and they are important in its proof, but they are harder to spot there than in fig. 19. It is an advantage of the refinement approach that they emerge so early in development.

Data refinement is straightforward, once we have added universally quantified L and R to track the values of the indexed variables. Refining pre and postconditions from fig. 6, the writer's specification is shown in fig. 20. We verify this specification below, but note already that it generates the action

$$l = L \wedge r = R \wedge ws = WS \wedge c[l] = B \rightsquigarrow l = !R \wedge r = R \wedge ws = WS.W \wedge c[l] = W$$

Because we don't know the relationship between l and r on the left-hand side, it isn't really clear what happens to l. We make it clearer by describing two actions: one in the case that $l = r$ initially, and the other in the case that $l \neq r$:

$$l = r = L \wedge ws = WS \wedge c[L] = B \rightsquigarrow !l = r = L \wedge ws = WS.W \wedge c[L] = B \wedge c[!L] = W$$
$$l = !r = L \wedge ws = WS \wedge c[L] = B \wedge c[!L] = B' \rightsquigarrow l = !r = L \wedge ws = WS.W \wedge c[L] = W \wedge c[!L] = B'$$

We can simplify these actions, just as in the writer-chooses case, by not mentioning the slot which doesn't change:

$$l = r = L \wedge ws = WS \rightsquigarrow !l = r = L \wedge ws = WS.W \wedge c[!L] = W \tag{21}$$
$$l = !r = L \wedge ws = WS \wedge c[L] = B \rightsquigarrow l = !r = L \wedge ws = WS.W \wedge c[L] = W \tag{22}$$

Each of these actions preserves the invariant (14). Action (21) has the effect of the two-stage refinement (16) of $\langle b := w\rangle$, writing to $c[!l]$ and inverting l; action (22) has the effect of the single-stage refinement (15), writing directly to $c[l]$. This is the second important property of this two-slot algorithm: the writer sometimes makes a two-stage update, sometimes one-stage (though at this point, before atomicity refinement, even the two-stage update takes place in a single instant).

The reader isn't so subtle. We data-refine fig. 16 with additions to track the values of l and r to give fig. 21, which verifies directly. The first step, which in the single-slot case generated the identity action, now

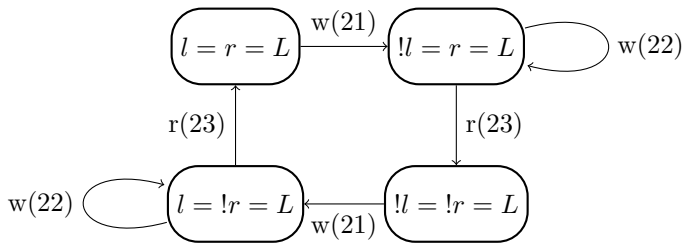


Fig. 22. Sequencing of actions in the reader-chooses algorithm

generates

$$l = L \rightsquigarrow l = r = L$$

We can clarify this by noting that when $l = r$ initially it's the identity action, and otherwise it inverts r:

$$l = !r = L \rightsquigarrow l = r = L \tag{23}$$

The second step generates (13) as before.

Another important property of the two-slot algorithm can now be seen if we consider the behaviour of the writer and reader with respect to r and l. Fig. 22 shows a finite state machine in which the actions of the writer are shown horizontally and the actions of the reader vertically. After a two-step update (21) the writer has set $l \neq r$, and its only possible action then, without intervention of the reader, is a one-step (22).

Now that we know the actions of each process, we can verify the writer. The difficulty is that it generates different actions in different starting states. In the RGSep atomic instruction rule (2) the action is calculated by the Hoare logic verification of the command inside the atomic brackets. We expect to generate two actions, one each for the disjunctive possibilities $l = r$ and $l \neq r$. We analyse each possibility separately and combine the results with Floyd's specification-disjunction rule [Flo67].

$$\frac{\{Q\}\ C\ \{R\} \quad \{Q'\}\ C\ \{R'\}}{\{Q \vee Q'\}\ C\ \{R \vee R'\}} \tag{24}$$

In the first case, when $l = r$, we have

$$\begin{array}{l} \{w = W \wedge l = r = L = R \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B\} \\ \quad c[!r] := w; l := !r; \boldsymbol{ws := ws.w} \\ \{\, w = W \wedge !l = r = L = R \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B \wedge c[!L] = W \,\} \end{array} \tag{25}$$

In the second case, when $l = !r$, we have

$$\begin{array}{l} \{w = W \wedge l = !r = L = !R \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B\} \\ \quad c[!r] := w; l := !r; \boldsymbol{ws := ws.w} \\ \{\, w = W \wedge l = !r = L = !R \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = W \,\} \end{array} \tag{26}$$

Putting these two together with the specification-disjunction rule gives the verification in fig. 23.

Data refinement is successful, then, even though the verification of the writer wasn't as simple as we might have hoped.

8.1. Atomicity refinement of reader-chooses writer

The writer's atomic action, without auxiliary annotation, is $\langle c[!r] := w; l := !r \rangle$. It's immediately clear that we can't simply separate $c[!r] := w$ from $l := !r$: updating $c[!r]$ first would allow the reader to change r so that l wouldn't point at the slot just written; putting the update of l first would allow the writer to point l at a slot not yet written. So we can't make an atomicity refinement in two stages as in the writer-chooses case. Instead we have to move immediately to a sequence which reads r into a local variable, uses that value

$$\left\{w = W \wedge \boxed{\begin{array}{l}(l = r = L = R \vee l = !r = L = !R) \wedge ws = WS.B \wedge \\ \quad \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B\end{array}}\right\}$$

$$\left\langle \left(\begin{array}{l} \left\{ \begin{array}{l} w = W \wedge (l = r = L = R \vee l = !r = L = !R) \wedge ws = WS.B \wedge \\ \quad \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B \end{array} \right\} \\ \quad c[!r] := w; l := !r; \boldsymbol{ws := ws.w} \\ \left\{ \begin{array}{l} w = W \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge \\ \quad \Big((!l = r = L = R \wedge c[L] = B \wedge c[!L] = W) \vee \\ \quad (l = !r = L = !R \wedge c[L] = W) \Big) \end{array} \right\} \Rightarrow \\ \{\, w = W \wedge l = !R \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[l] = W \,\} \end{array} \right) \right\rangle$$

$$\left\{w = W \wedge \boxed{l = !R \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[l] = W}\right\}$$

Fig. 23. Verification of two-slot writer (reader chooses)

to index c, and then updates l from the value it read in the first step:

$$\langle\!\langle wt := !r \rangle\!\rangle; \langle c[wt] := w \rangle; \langle\!\langle l := wt \rangle\!\rangle \tag{27}$$

Even then we haven't solved the problem, because it isn't clear where to put the auxiliary update to ws. Analysis of the writer has shown that it produces two actions, one if initially $l = r$, another if initially $l \neq r$. In the first case (action (21)) the writer changes $c[!l]$ and then updates l: to preserve $ws_\Omega = c[l]$ the update to ws has to go with the update to l. In the second case (action (22)) the writer updates $c[l]$ directly and then doesn't change l: to preserve $ws_\Omega = c[l]$ the update to ws has to go with the update to $c[l]$. That means that we can't simply use the specification-disjunction rule to combine verifications of the $l = r$ and $l \neq r$ cases, because in one case we analyse

$$\langle\!\langle wt := !r \rangle\!\rangle; \langle c[wt] := w \rangle; \langle\!\langle l := wt; \boldsymbol{ws := ws.w} \rangle\!\rangle$$

and in the other

$$\langle\!\langle wt := !r \rangle\!\rangle; \langle c[wt] := w; \boldsymbol{ws := ws.w} \rangle; \langle\!\langle l := wt \rangle\!\rangle$$

We could solve this problem by using conditional updates to ws, discriminating on whether $wt = l$ or not, but we'd rather not make auxiliary activity appear to be the major part of the algorithm. What we are faced with is commonly dealt with in linearisation [HW90]. We know that at some point during the execution of the writer's body an update to the refined buffer variable occurs and ws is updated to match, and we know that it can happen at different points in different circumstances. When we consider the body overall we don't care where it happens, merely that it happens somewhere.

Auxiliary assignments don't really execute: the program can't detect their effects and can't choose what to do on the basis of an auxiliary value. We propose that refinement of an atomic sequence which includes an auxiliary assignment needn't be definite about where that assignment happens, provided of course that stable assertions in other threads remain stable no matter where it actually happens. We define our problem as to verify

$$\{P\} \left(\begin{array}{c} \boldsymbol{ws := ws.w} \\ \langle\!\langle wt := !r \rangle\!\rangle; \langle c[wt] := w \rangle; \langle\!\langle l := wt \rangle\!\rangle \end{array} \right) \{Q\} \tag{28}$$

in which the floating auxiliary assignment is guaranteed, whatever the initial conditions, to land just once somewhere in the program below. This notation was inspired by relational separation logic [Yan07] but our logic and our aims are different: the two levels in our notation interact, sharing state, and produce composite actions.

The rules we use, which guarantee that the auxiliary assignment is placed somewhere and placed exactly once, are shown in table 1: if an assignment floats above an atomic command it must be placed there; if it floats above a sequence it must float either to the left or the right, whichever the proof requires. These rules are all that we need, since in our examples we are only dealing with sequences of instructions; we could if necessary add rules which deal with atomic skips so as to handle linearisation in another thread, or to deal

Table 1. Rules for floating auxiliary assignments

$$\frac{\{P\}\ \langle C; A\rangle\ \{Q\}}{\{P\}\ \left(\begin{array}{c} A \\ \langle C\rangle \end{array}\right)\ \{Q\}}\ \text{postatomic}$$

$$\frac{\{P\}\ \left(\begin{array}{c} A \\ C1 \end{array}\right); C2\ \{Q\}}{\{P\}\ \left(\begin{array}{c} A \\ C1; C2 \end{array}\right)\ \{Q\}}\ \text{presequence} \qquad \frac{\{P\}\ C1; \left(\begin{array}{c} A \\ C2 \end{array}\right)\ \{Q\}}{\{P\}\ \left(\begin{array}{c} A \\ C1; C2 \end{array}\right)\ \{Q\}}\ \text{postsequence}$$

$$\begin{array}{l} \left\{w = W \wedge \boxed{l = r = L = R \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B}\right\} \\ \quad \langle\!\langle wt := !r \rangle\!\rangle \\ \left\{w = W \wedge wt = !L \wedge \boxed{l = r = L = R \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B}\right\} \\ \quad \langle c[wt] := w \rangle \\ \left\{w = W \wedge wt = !L \wedge \boxed{l = r = L = R \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B \wedge c[!L] = W}\right\} \\ \quad \langle\!\langle l := wt; \boldsymbol{ws := ws.w} \rangle\!\rangle \\ \left\{w = W \wedge wt = !L \wedge \boxed{!l = r = L = R \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B \wedge c[!L] = W}\right\} \end{array}$$

Fig. 24. Atomicity refinement of two-slot writer (reader chooses), $l = r$ case

with conditionals and loops. We stress once again that this solution is merely a notational convenience, not a change in semantics: what is being sugared is conditional execution of auxiliary assignments.

As before, then, we consider two cases and put them together with the specification-disjunction rule. The precondition disjunction is, as before, between $l = r = L = R$ and $l = !r = L = !R$. Because we use mid stability we can consider each case directly. The case when $l = r$, noting that $l = r = L = R$ is stable under the reader's actions (23) and (13), is shown in fig. 24. The first step generates the identity action; the second and third generate

$$l = r = L \rightsquigarrow l = r = L \wedge c[!L] = W \tag{29}$$

$$l = r = L \wedge ws = WS \wedge c[!L] = W \rightsquigarrow !l = r = L \wedge ws = WS.W \wedge c[!L] = W \tag{30}$$

Action (29) has no effect on the invariant (14), because it affects only $c[!l]$; action (30) updates ws and l to preserve the invariant. Crucially, these actions can only happen when $l = r$ and can only happen in sequence: both these facts will turn out to be important.

$$\begin{array}{l} \left\{w = W \wedge \boxed{l = !r = L = !R \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B}\right\} \\ \quad \langle\!\langle wt := !r \rangle\!\rangle \\ \left\{w = W \wedge wt = L \wedge \boxed{l = !r = L = !R \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B}\right\} \\ \qquad\qquad \Rightarrow \text{ (stability)} \\ \left\{w = W \wedge wt = L \wedge \boxed{l = L = !R \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B}\right\} \\ \quad \langle c[wt] := w; \boldsymbol{ws := ws.w} \rangle \\ \left\{w = W \wedge wt = L \wedge \boxed{l = L = !R \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = W}\right\} \\ \quad \langle\!\langle l := wt \rangle\!\rangle \\ \left\{w = W \wedge wt = L \wedge \boxed{l = L = !R \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = W}\right\} \end{array}$$

Fig. 25. Atomicity refinement of two-slot writer (reader chooses), $l \neq r$ case

$$\Big\{\boxed{l = L \wedge rs = RS \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B}\Big\}$$
$$\langle r := l \rangle$$
$$\Big\{\boxed{l = r = L \wedge rs = RS \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B}\Big\} \;\Rightarrow\; \text{(stability)}$$
$$\left\{\boxed{\exists B' \left(\begin{array}{l} (l = r = L \vee !l = r = L) \wedge rs = RS \wedge \lfloor rs.B.B' \rfloor \preccurlyeq ws \wedge \\ \quad (\,(c[L] = B') \wedge (c[l] = ws_\Omega)\,) \end{array} \right)}\right\}$$
$$\langle y := c[r] \rangle$$
$$\Big\{\exists B' \Big(y = B' \wedge \boxed{r = L \wedge rs = RS \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge \lfloor rs.B.B' \rfloor \preccurlyeq ws \wedge c[l] = ws_\Omega} \Big)\Big\}$$

Fig. 26. Atomicity refinement of two-slot reader (reader chooses)

In the case when $l = !r$ initially, we have the verification in fig. 25. The first and third steps generate the identity action, and the second generates

$$l = L \wedge ws = WS \wedge c[L] = B \rightsquigarrow l = L \wedge ws = WS.W \wedge c[L] = W \tag{31}$$

– almost the same as (22) except that we don't know, because of the stability weakening, the value of r before or after. This action preserves the invariant because it updates $c[l]$ and ws consistently and atomically.

In both fig. 24 and fig. 25 the linearisation point, before which the operation of the writer has no effect on the reader and after which the update is completely visible, is marked by the auxiliary assignment to ws. The postcondition when we put these disjoint verifications together is

$$\left\{ \begin{array}{l} \left(\begin{array}{l} w = W \wedge wt = !L \wedge \\ \quad \boxed{!l = r = L = !R \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B \wedge c[!L] = W} \end{array} \right) \vee \\ \left(\begin{array}{l} w = W \wedge wt = L \wedge \\ \quad \boxed{l = L = !R \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = W} \end{array} \right) \end{array} \right\} \Rightarrow$$
$$\Big\{ w = W \wedge \boxed{l = !R \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[l] = W} \Big\} \tag{32}$$

– the same as in the pre-refinement verification fig. 23 except that we have lost $r = R$ because of stabilisation in the verification of the $l \neq r$ disjunct.

All three actions – (29), (30) and (31) – preserve stability of the asssertions in fig. 21. We have an atomicity refinement.

8.2. Atomicity refinement of the reader-chooses reader

If we split the first atomic command of the reader of fig. 19 into a sequence $\langle r := l \rangle; \langle y := c[r] \rangle$ we won't disturb the writer's stable assertions: the only effect on the shared store, as before, is to change $l \neq r$ into $l = r$. Using the same precondition as fig. 21, we have the verification in fig. 26. The important step is the stability weakening: $l = r = L$ weakens to $l = r = L \vee !l = r = L$ because of (30); $c[L]$ can be overwritten, it would seem, by either (29) or (31). The important part of the weakening step is the claim $\lfloor rs.B.B' \rfloor \preccurlyeq ws$: if the reader doesn't read B, the latest value when it starts, then it reads a later value; if the claim were more simply $\lfloor rs.B' \rfloor \preccurlyeq ws$ then we wouldn't so obviously have proved that rs is not simply a sequence of *null*s.

To justify the claim that $c[L]$ always contains a fresh value that is part of the written sequence it is illuminating to revisit the state machine behaviour of the writer. Atomicity refinement has made the writer a little more complicated, and we clarify its behaviour by annotating with assignments to an auxiliary variable ac (for $a$ction $c$ounter):

$$\begin{array}{l} \mathsf{bit}\ wt, ac = 0\ \mathsf{in} \\ \quad \langle\!\langle wt := !r; \boldsymbol{ac := (l = r\ ?\ 3 : 1)} \rangle\!\rangle; \\ \quad \left(\begin{array}{c} \boldsymbol{ws := ws.w} \\ \langle c[wt] := w; \boldsymbol{ac := ac - 1} \rangle; \langle\!\langle l := wt; \boldsymbol{ac := 0} \rangle\!\rangle \end{array} \right) \\ \mathsf{ni} \end{array} \tag{33}$$

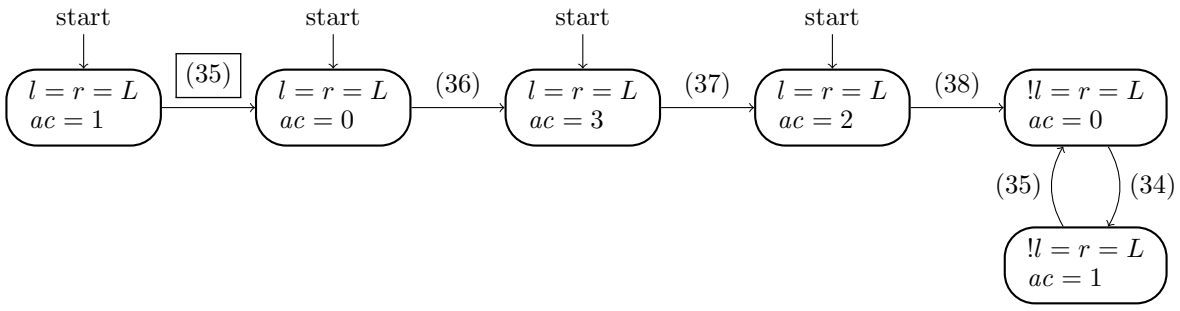


Fig. 27. Sequencing of two-slot writer's actions after $\langle r := l \rangle$

We omit the details of the verification, but there are now two auxiliary actions produced by the first step, which previously generated the identity action, and other actions now change the action counter:

$$ac = 0 \wedge l = !r = L \rightsquigarrow ac = 1 \wedge l = !r = L \tag{34}$$

$$ac = 1 \wedge l = L \wedge ws = WS \wedge c[L] = B \rightsquigarrow ac = 0 \wedge l = L \wedge ws = WS.B' \wedge c[L] = B' \tag{35}$$

$$ac = 0 \wedge l = r = L \rightsquigarrow ac = 3 \wedge l = r = L \tag{36}$$

$$ac = 3 \wedge l = r = L \wedge c[!L] = B \rightsquigarrow ac = 2 \wedge l = r = L \wedge c[!L] = B' \tag{37}$$

$$ac = 2 \wedge l = r = L \wedge ws = WS \wedge c[!L] = B \rightsquigarrow ac = 0 \wedge !l = r = L \wedge ws = WS.W \wedge c[!L] = B \tag{38}$$

To reason about stability we have to consider the writer's actions in the absence of action by the reader – in this case, what the writer can do after $\langle r := l \rangle$. Fig. 27 shows a finite state machine. When the reader starts we have $l = L$, and instantaneously after $\langle\langle r := l \rangle\rangle$ we have $l = r = L$. Reading across the diagram from left to right the boxed action (35) updates $c[L]$ but also makes it ws_Ω; (36) is auxiliary; (37) when $l = r = L$ writes to $c[!L]$; (38) updates l and ws but doesn't touch c; (34) is auxiliary; and (35) when $l = !r = L$ also writes to $c[!L]$. Only the boxed operation (35) at the left of the diagram writes to $c[L]$ (which is also $c[r]$); that can only happen once; and it updates ws at the same time.

So in fig. 26 we can assert $\lfloor rs.B.B' \rfloor \preccurlyeq ws$ after stability weakening: if B' is B, because $c[L]$ hasn't been overwritten, we have $\lfloor rs.B \rfloor \preccurlyeq ws$, as when the reader starts; and if $c[L]$ is overwritten by the writer it is with the latest value in the written sequence. It isn't the case that $c[L] = ws_\Omega$ is invariant: two actions in the diagram make $c[!L]$ the latest value, but that doesn't matter since the reader always reads a value which is at least as fresh as the latest at the instant it began.

The postcondition of $\langle y := c[r] \rangle$ in fig. 26 isn't the same as the postcondition of $\langle r := l; y := c[l] \rangle$ in fig. 21 but they have equivalent force so far as the subsequent update to rs is concerned: the value in y is a valid extension to rs and is either B or later than B in ws. We have an atomicity refinement.

The matter of linearisability isn't quite so simple as it was in the single-variable algorithm. Clearly, the linearisation points in the writer are $\langle c[wt] := w \rangle$ when $wt = l$, and $\langle\langle l := wt \rangle\rangle$, when $wt = !l$. In the reader it's more subtle. If we choose $\langle r := l \rangle$ as the linearisation point in the reader we could be mistaken: fig. 27 shows that the writer can execute action (35) and write to $c[r]$ between that point and the execution of $\langle y := c[r] \rangle$; the linearisation point must be such that the value received by the reader is produced by the write-linearisation point which immediately precedes it in the concurrent execution; so $\langle r := l \rangle$ could be too early. If we take $\langle y := c[r] \rangle$ we could be too late: after $\langle r := l \rangle$ the writer can take various actions in fig. 27 which write to $c[!r]$, none of which would write values that could be received by $\langle y := c[r] \rangle$.

The solution to the problem is part of the theory of linearisability. Sometimes the linearisation point of a procedure call is in another process in the same concurrent execution. The linearisation point of the reader is either $\langle r := l \rangle$, if the writer doesn't write to $c[r]$ between that point and $\langle y := c[r] \rangle$; it's the writer's linearisation point itself if it does. This may seem like special pleading, but all we must do is to show that there *is an instant* during the execution of each procedure call after which the actions of the environment don't have an effect. This algorithm is linearisable; for every concurrent execution there is an equivalent sequential execution.

$$\begin{array}{r||l||l||l}
 & l = !r = L & & \\
\langle\!\langle wt := !r \rangle\!\rangle & & & \\
\{wt = L\} & & & \langle r := l \rangle \\
 & l = r = L & & \\
\langle c[wt] := w \rangle & & & \langle y := c[r] \rangle
\end{array}$$

Fig. 28. Collision in atomicity-refined reader-chooses algorithm

$$\Big\{\boxed{l = L \wedge rs = RS \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge ws = WS.B \wedge c[L] = B}\Big\}$$
$$\langle\!\langle rt := l \rangle\!\rangle$$
$$\Big\{rt = L \wedge \boxed{l = L \wedge rs = RS \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge \lfloor rs.B \rfloor \preccurlyeq ws \wedge ws_\Omega = B \wedge c[L] = B}\Big\}$$
$$\Rightarrow \text{ (stability)}$$
$$\Big\{rt = L \wedge \boxed{\exists B' \left(\begin{array}{l} (l = r = L \vee l = !r = L \vee !l = r = L) \wedge rs = RS \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge \lfloor rs.B.B' \rfloor \preccurlyeq ws \wedge \\ \quad ((c[L] = B') \wedge (c[l] = ws_\Omega)) \end{array} \right)}\Big\}$$
$$\langle\!\langle r := rt \rangle\!\rangle$$
$$\Big\{rt = L \wedge \boxed{\exists B' \left(\begin{array}{l} r = L \wedge rs = RS \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge \lfloor rs.B.B' \rfloor \preccurlyeq ws \wedge \\ \quad ((c[L] = B') \wedge (c[l] = ws_\Omega)) \end{array} \right)}\Big\}$$
$$\langle y := c[rt] \rangle$$
$$\Big\{\exists B' \Big(y = B' \wedge \boxed{r = L \wedge rs = RS \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge \lfloor rs.B.B' \rfloor \preccurlyeq ws \wedge c[l] = ws_\Omega} \Big)\Big\}$$

Fig. 29. Second atomicity refinement of two-slot reader (reader chooses)

9. The essence of Simpson's algorithm

We now have almost all the information we need to see how and why Simpson's algorithm works. Writer and reader can operate in the same slot in the two-slot reader-chooses algorithm, as illustrated in fig. 28, so two slots are not a solution. But fig. 27 shows that there can be only one such collision: after $\langle r := l \rangle$ the writer can only write once in the slot $c[r]$, with a single-step update, and afterwards, until the reader intervenes to change r, it has to write repeatedly in $c[!r]$. Further atomicity refinement of the two-slot reader preserves that property, and refinement to four slots simply allows the two processes to avoid each other when the refined collision occurs.

Once again refinement has revealed an essential property of the algorithm before it is buried in implementation detail.

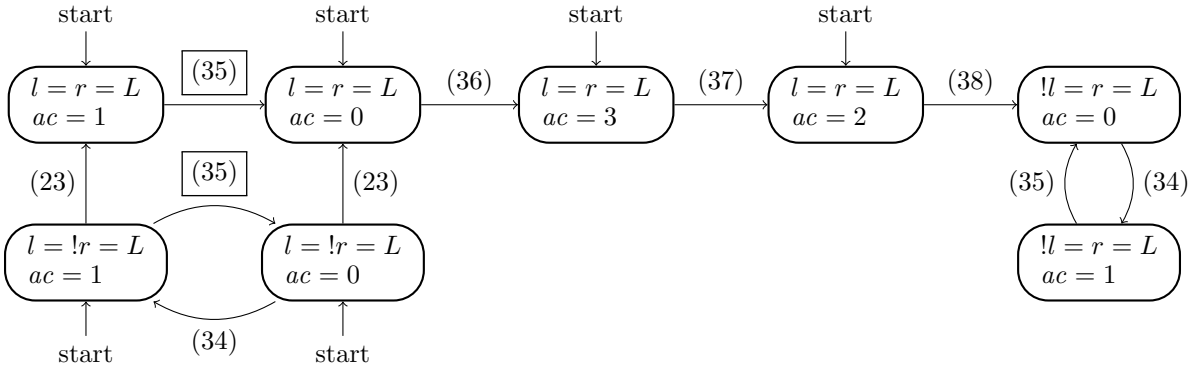


Fig. 30. Sequencing of two-slot actions after $\langle rt := l \rangle$

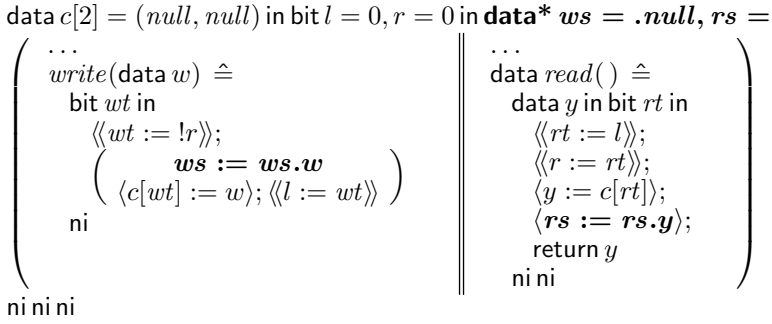


Fig. 31. Two-slot reader-chooses algorithm after atomicity refinement

10. Further refinement of the reader-chooses reader

We can make another step of atomicity refinement and move to single-access atomic instructions in the reader, using a local variable rt to refine $\langle r := l\rangle; \langle y := c[r]\rangle$ into $\langle\!\langle rt := l\rangle\!\rangle; \langle\!\langle r := rt\rangle\!\rangle; \langle y := c[rt]\rangle$. In fig. 29 the first step generates the identity action. The stability weakening is, once again, the important part of the verification. The finite-state machine in fig. 30 shows the situation after $\langle\!\langle rt := l\rangle\!\rangle$ and its transition, via action (23) of the reader (the vertical arrows to the left of the diagram) to the same machine as fig. 27 after $\langle\!\langle r := rt\rangle\!\rangle$.

Initially, as before, we have $l = L$ and either $l = r$ or $l = !r$. The reader's $\langle\!\langle rt := l\rangle\!\rangle$ doesn't change any of that, so the writer can start in any of the six indicated states. This time there are two transitions which can write to $c[L]$, both on the left of the diagram. The lower one can happen repeatedly, but each time it does ws is updated and $c[L]$ becomes the latest value. Then the reader's $\langle\!\langle r := rt\rangle\!\rangle$ moves, with action (23), from whichever of the two lower states is active to the corresponding upper state, and (as before atomicity refinement) there is only the possibility of a single further collision.

The third step in fig. 29 generates the identity action and produces the same postcondition as fig. 26. Overall we have the same actions and the same postcondition, and therefore an atomicity refinement. Linearisation in the reader is as indicated in §5.2: either at $\langle\!\langle rt := l\rangle\!\rangle$, if it picks up B, or at the last write to $c[r]$ which occurs between that point and $\langle y := c[rt]\rangle$, if it picks up a later value B'.

The refined algorithm is shown in fig. 31. It uses two slots, it always reads in sequence, and though the reader doesn't necessarily get the latest value, it gets one which is no earlier than the one which was latest when it executed its first instruction.

11. Refinement to four slots

So far we've data-refined the one-slot atomic buffer of fig. 2 into the two-slot reader-chooses atomic buffer of fig. 19, and atomicity-refined to the algorithm of fig. 31. It remains to refine the buffer-slot accesses $\langle c[wt] := w\rangle$ and $\langle y := c[rt]\rangle$.

In the two-slot reader-chooses algorithm there can be at most one collision once the reader has declared, in variable r, which slot it is about to read from. Suppose that we double each slot, replacing $c[2]$ by $d[2][2]$, introduce an array of single-bit index variables $i[2]$, and refine $c[l]$ by $d[l][i[l]]$ – i.e. l points to the latest pair, and $i[I]$ points to the latest slot within pair I.

As in the two-slot data refinement we have a choice in refining $\langle c[wt] := w\rangle$: either to use a single-step refinement $\langle d[wt][i[wt]] := w\rangle$ which writes directly to the latest slot in the pair; or a two-step refinement $\langle d[wt][!i[wt]] := w; i[wt] := !i[wt]\rangle$ which writes to the not-latest slot and then inverts the index variable. As before, the single-step choice would be pointless, but making the two-step choice got us into trouble in the writer-chooses algorithm. That trouble came about on the *second* write, and this time, as we've seen, when there is a collision there is no second write to the same pair. Writer and reader, in the refined algorithm, will collide at most once in the same pair and then go their separate ways; in that pair-collision the writer will write to $d[L][!i[L]]$ and the reader will read from $d[L][i[L]]$. So there will be no slot-collision, and therefore not even a need for atomic buffer accesses. It remains only to work out the details.

$$
\begin{array}{l}
\mathsf{data}\ d[2][2] = ((null, null), (null, null))\ \mathsf{in}\ \mathsf{bit}\ l = 0, r = 0, i[2] = (0,0)\ \mathsf{in} \\
\qquad \textbf{data* } \boldsymbol{ws = .null, rs = .}\ \mathsf{in} \\
\left(\begin{array}{l} \ldots \\ write(\mathsf{data}\ w)\ \hat{=} \\ \quad \mathsf{bit}\ wt\ \mathsf{in} \\ \quad\ \langle\!\langle wt := !r \rangle\!\rangle; \\ \quad\ \left(\begin{array}{c} \boldsymbol{ws := ws.w} \\ \left\langle \begin{array}{l} d[wt][!i[wt]] := w; \\ i[wt] := !i[wt] \end{array} \right\rangle ; \langle\!\langle l := wt \rangle\!\rangle \end{array} \right) \\ \quad \mathsf{ni} \end{array} \right. \\
\qquad \Big\| \\
\left. \begin{array}{l} \ldots \\ \mathsf{data}\ read(\,)\ \hat{=} \\ \quad \mathsf{data}\ y\ \mathsf{in}\ \mathsf{bit}\ rt\ \mathsf{in} \\ \quad\ \langle\!\langle rt := l \rangle\!\rangle; \\ \quad\ \langle\!\langle r := rt \rangle\!\rangle; \\ \quad\ \langle y := d[rt][i[rt]] \rangle; \\ \quad\ \langle \boldsymbol{rs := rs.y} \rangle; \\ \quad\ \mathsf{return}\ y \\ \quad \mathsf{ni}\ \mathsf{ni} \end{array} \right) \\
\mathsf{ni}\ \mathsf{ni}\ \mathsf{ni}
\end{array}
$$

Fig. 32. Reader-chooses algorithm refined to four slots

$$
\begin{array}{l}
\left\{ w = W \wedge \boxed{l = r = L = R \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge i[L] = I \wedge d[L][I] = B \wedge i[!L] = I'} \right\} \\
\quad \langle\!\langle wt := !r \rangle\!\rangle \\
\left\{ w = W \wedge wt = !L \wedge \boxed{l = r = L = R \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge i[L] = I \wedge d[L][I] = B \wedge i[!L] = I'} \right\} \\
\quad \langle d[wt][!i[wt]] := w; i[wt] := !i[wt] \rangle \\
\left\{ w = W \wedge wt = !L \wedge \boxed{\begin{array}{l} l = r = L = R \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge \\ \quad i[L] = I \wedge d[L][I] = B \wedge i[!L] = !I' \wedge d[!L][!I'] = W \end{array}} \right\} \\
\quad \langle\!\langle l := wt; \boldsymbol{ws := ws.w} \rangle\!\rangle \\
\left\{ w = W \wedge \boxed{\begin{array}{l} !l = r = L = R \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge \\ \quad i[L] = I \wedge d[L][I] = B \wedge i[!L] = !I' \wedge d[!L][!I'] = W \end{array}} \right\}
\end{array}
$$

Fig. 33. Data refinement of writer to four slots ($l = r$ case from fig. 24)

Fig. 32 shows the data-refined algorithm; the refined communication invariant is

$$\lfloor rs \rfloor \preccurlyeq ws \wedge d[l][i[l]] = ws_\Omega \tag{39}$$

We verify the data-refined writer in two cases, as before. First when $l = r$ we have fig. 33. The first step generates the identity action, as before; the second and third generate

$$l = r = L \wedge i[!l] = I \rightsquigarrow l = r = L \wedge i[!l] = !I \wedge d[!l][!I] = W \tag{40}$$

$$l = r = L \wedge ws = WS \wedge i[!l] = I \wedge d[!l][I] = W \rightsquigarrow !l = r = L \wedge ws = WS.W \wedge i[l] = I \wedge d[l][I] = W \tag{41}$$

which are obvious data refinements of (29) and (30). Verification is straightforward.

In the case that $l \neq r$, we have fig. 34. The first and third steps generate the identity action, and the second step generates

$$l = L \wedge ws = WS \wedge i[l] = I \rightsquigarrow l = L \wedge ws = WS.W \wedge i[l] = !I \wedge d[l][!I] = W \tag{42}$$

which is a data refinement of (31) in the same sense that (17) was a data refinement of (11): if we include $d[l][I] = B$ on both sides, since it is unchanged by the second step, it clearly acts to change $d[l][i[l]] = B$ into $d[l][i[l]] = W$. Verification is once again straightforward.

$$\Big\{w = W \wedge \boxed{l = !r = L = !R \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge i[L] = I \wedge d[L][I] = B}\Big\}$$
$$\langle\!\langle wt := !r \rangle\!\rangle$$
$$\Big\{w = W \wedge wt = L \wedge \boxed{l = !r = L = !R \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge i[L] = I \wedge d[L][I] = B}\Big\}$$
$$\Rightarrow \text{ (stability)}$$
$$\Big\{w = W \wedge wt = L \wedge \boxed{l = L = !R \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge i[L] = I \wedge d[L][I] = B}\Big\}$$
$$\langle d[wt][!i[wt]] := w; i[wt] := !i[wt]; \boldsymbol{ws := ws.w} \rangle$$
$$\Big\{w = W \wedge wt = L \wedge \boxed{l = L = !R \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge i[L] = !I \wedge d[L][I] = B \wedge d[L][!I] = W}\Big\}$$
$$\langle\!\langle l := wt \rangle\!\rangle$$
$$\Big\{w = W \wedge wt = L \wedge \boxed{l = L = !R \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge i[L] = !I \wedge d[L][I] = B \wedge d[L][!I] = W}\Big\}$$

Fig. 34. Data refinement of writer to four slots ($l = !r$ case from fig. 25)

$$\Big\{\boxed{l = L \wedge rs = RS \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge ws_\Omega = B \wedge i[L] = I \wedge d[L][I] = B}\Big\}$$
$$\langle\!\langle rt := l \rangle\!\rangle$$
$$\Big\{rt = L \wedge \boxed{l = L \wedge rs = RS \wedge \lfloor rs.B \rfloor \preccurlyeq ws \wedge ws_\Omega = B \wedge i[L] = I \wedge d[L][I] = B}\Big\}$$
$$\Rightarrow \text{ (stability)}$$
$$\left\{rt = L \wedge \boxed{\exists B' \left(\begin{array}{l} (l = r = L \vee l = !r = L \vee !l = r = L) \wedge rs = RS \wedge \lfloor rs.B.B' \rfloor \preccurlyeq ws \wedge \\ \quad \exists I' (i[L] = I' \wedge d[L][I'] = B') \wedge \exists I'' (i[l] = I'' \wedge d[l][I''] = ws_\Omega) \end{array} \right)}\right\}$$
$$\langle\!\langle r := rt \rangle\!\rangle$$
$$\left\{rt = L \wedge \boxed{\exists B' \left(\begin{array}{l} r = L \wedge rs = RS \wedge \lfloor rs.B.B' \rfloor \preccurlyeq ws \wedge \\ \quad \exists I' (i[L] = I' \wedge d[L][I'] = B') \wedge \exists I'' (i[l] = I'' \wedge d[l][I''] = ws_\Omega) \end{array} \right)}\right\}$$
$$\langle y := d[rt][i[rt]] \rangle$$
$$\Big\{\exists B' \Big(y = B' \wedge \boxed{r = L \wedge rs = RS \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge \lfloor rs.B.B' \rfloor \preccurlyeq ws \wedge \exists I'' (i[l] = I'' \wedge d[l][I''] = ws_\Omega)}\Big)\Big\}$$

Fig. 35. Data refinement of reader to four slots (from fig. 29)

Combining the postconditions of fig. 33 and fig. 34 gives a data refinement of (32):

$$\left\{ \begin{array}{l} \left(\begin{array}{l} w = W \wedge wt = !L \wedge \\ \quad \boxed{\begin{array}{l} !l = r = L = R \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge \\ \quad i[L] = I \wedge d[L][I] = B \wedge \\ \quad i[!L] = !I' \wedge d[!L][!I'] = W \end{array}} \end{array} \right) \vee \\ \left(\begin{array}{l} w = W \wedge wt = L \wedge \\ \quad \boxed{\begin{array}{l} l = L = !R \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge \\ \quad i[L] = !I \wedge d[L][I] = B \wedge d[L][!I] = W \end{array}} \end{array} \right) \end{array} \right\} \Rightarrow \tag{43}$$
$$\left\{w = W \wedge \boxed{\begin{array}{l} l = !R \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge \\ \quad \exists I'' (i[l] = I'' \wedge d[l][I''] = W) \end{array}}\right\}$$

So far as the writer is concerned, then, we have a data refinement. Its actions preserve the invariant (39) so we shouldn't have any stability issues in the reader; the reader still only alters r and rs, both of which were allowed for in stability arguments before data refinement.

In the reader, data refinement of fig. 29 gives fig. 35. The stability argument carries across the refinement: (40) writes to $i[!L]$ and $d[!L][\_]$; (41) writes to l and ws; (42) can write to $i[L]$ and $d[L][\_]$ but, when it does so, it updates ws so that $d[L][i[L]]$ is always a valid update to rs. The action generated by the second step is

$$\left\{w = W \wedge wt = !L \wedge \boxed{\begin{array}{l} l = r = L = R \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge \\ i[L] = I \wedge d[L][I] = B \wedge i[!L] = I' \end{array}}\right\}$$
$$\langle\langle wi := !i[wt] \rangle\rangle$$
$$\left\{w = W \wedge wt = !L \wedge wi = !I' \wedge \boxed{\begin{array}{l} l = r = L = R \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge \\ i[L] = I \wedge d[L][I] = B \wedge i[!L] = I' \end{array}}\right\}$$
$$\langle d[wt][wi] := w \rangle$$
$$\left\{w = W \wedge wt = !L \wedge wi = !I' \wedge \boxed{\begin{array}{l} l = r = L = R \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge \\ i[L] = I \wedge d[L][I] = B \wedge i[!L] = I' \wedge d[!L][!I'] = W \end{array}}\right\}$$
$$\langle\langle i[wt] := wi \rangle\rangle$$
$$\left\{w = W \wedge wt = !L \wedge \boxed{\begin{array}{l} l = r = L = R \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge \\ i[L] = I \wedge d[L][I] = B \wedge i[!L] = !I' \wedge d[!L][!I'] = W \end{array}}\right\}$$

Fig. 36. Atomicity refinement of four-slot writer, $l = r$ case

$$\left\{w = W \wedge wt = L \wedge \boxed{\begin{array}{l} l = L = !R \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge \\ i[L] = I \wedge d[L][I] = B \wedge i[!L] = I' \end{array}}\right\}$$
$$\langle\langle wi := !i[wt] \rangle\rangle$$
$$\left\{w = W \wedge wt = L \wedge wi = !I \wedge \boxed{\begin{array}{l} l = L = !R \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge \\ i[L] = I \wedge d[L][I] = B \wedge i[!L] = I' \end{array}}\right\}$$
$$\langle d[wt][wi] := w \rangle$$
$$\left\{w = W \wedge wt = L \wedge wi = !I \wedge \boxed{\begin{array}{l} l = L = !R \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge \\ i[L] = I \wedge d[L][I] = B \wedge d[L][!I] = W \wedge i[!L] = I' \end{array}}\right\}$$
$$\langle\langle i[wt] := wi; \boldsymbol{ws := ws.w} \rangle\rangle$$
$$\left\{w = W \wedge wt = L \wedge \boxed{\begin{array}{l} l = L = !R \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge \\ i[L] = !I \wedge d[L][I] = B \wedge d[L][!I] = W \wedge i[!L] = I' \end{array}}\right\}$$

Fig. 37. Atomicity refinement of four-slot writer, $l \neq r$ case

(23) when $l = !r = L$, as before. The postcondition provides all that is needed for the subsequent update to rs to generate (13) (details omitted). Linearisation is just as for the pre-refinement fig. 31, either at $\langle\langle rt := l \rangle\rangle$ if $B' = B$ or at the writer which produces B', otherwise.

So we have a data refinement to four slots.

11.1. Atomicity refinement of the four-slot writer

To atomicity-refine the writer we have to verify the $l = r$ case from fig. 33 and the $l = !r$ case from fig. 34. In each case we introduce a new local variable wi and replace $d[wt][!i[wt]] := w; i[wt] := !i[wt]$ by $wi := !i[wt]; d[wt][wi] := w; i[wt] := wi$.

Extracting the precondition of the second step in fig. 33 and splitting up the atomic sequence we have fig. 36. The postcondition is just the postcondition of the second step in fig. 33. But now the single action (40) has become two actions generated by the second and third steps:

$$l = r = L \wedge i[!l] = I \rightsquigarrow l = r = L \wedge i[!l] = I \wedge d[!l][!I] = W \tag{44}$$
$$l = r = L \wedge i[!l] = I \rightsquigarrow l = r = L \wedge i[!l] = !I \tag{45}$$

Nothing in the stability assertions in the reader is disturbed: (44) only operates on $d[!rt][\_]$, and (45) only operates on $i[!rt]$. This is an atomicity refinement.

In the other case, taking the precondition of the second step from fig. 34 and splitting up the atomic sequence – note that we keep the updates of $i[wt]$ and ws together in a single atomic step – we have fig. 37.

$$\left\{ rt = L \wedge \boxed{\exists B' \left(\begin{array}{l} r = L \wedge rs = RS \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge \lfloor rs.B.B' \rfloor \preccurlyeq ws \wedge \\ \exists I'(i[L] = I' \wedge d[L][I'] = B') \wedge \exists I''(i[l] = I'' \wedge d[l][I''] = ws_\Omega) \end{array} \right)} \right\}$$

$$\langle\!\langle ri := i[rt] \rangle\!\rangle$$

$$\left\{ \exists I' \left(\begin{array}{l} rt = L \wedge ri = I' \wedge \\ \boxed{\exists B' \left(\begin{array}{l} r = L \wedge rs = RS \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge \lfloor rs.B.B' \rfloor \preccurlyeq ws \wedge \\ \left(\begin{array}{l} \exists I'(i[L] = I' \wedge d[L][I'] = B') \wedge \\ \exists I''(i[l] = I'' \wedge d[l][I''] = ws_\Omega) \end{array} \right) \end{array} \right)} \end{array} \right) \right\}$$

$$\Rightarrow \text{ (stability)}$$

$$\left\{ \exists I' \left(\begin{array}{l} rt = L \wedge ri = I' \wedge \\ \boxed{\exists B' \left(\begin{array}{l} r = L \wedge rs = RS \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge \lfloor rs.B.B' \rfloor \preccurlyeq ws \wedge \\ \left(\begin{array}{l} (d[L][I'] = B') \wedge \\ \exists I''(i[l] = I'' \wedge d[l][I''] = ws_\Omega) \end{array} \right) \end{array} \right)} \end{array} \right) \right\}$$

$$\langle y := d[rt][ri] \rangle$$

$$\left\{ \exists B' \left(y = B' \wedge \boxed{\begin{array}{l} r = L \wedge rs = RS \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge \lfloor rs.B.B' \rfloor \preccurlyeq ws \wedge \\ (\, \exists I''(i[l] = I'' \wedge d[l][I''] = ws_\Omega) \,) \end{array}} \right) \right\}$$

Fig. 38. Atomicity refinement of four-slot reader

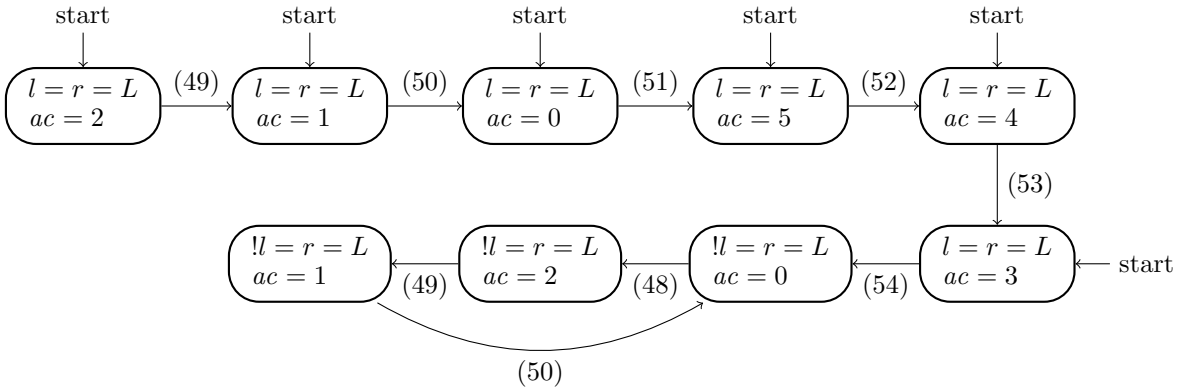


Fig. 39. Sequencing of four-slot writer's actions after $\langle\!\langle r := rt \rangle\!\rangle$

Again, the postcondition is just the postcondition of the second step in fig. 34. Again the single action (42) has been split into two:

$$l = L \wedge i[l] = I \rightsquigarrow l = L \wedge i[l] = I \wedge d[l][!I] = W \tag{46}$$

$$l = L \wedge ws = WS \wedge i[l] = I \wedge d[l][!I] = W \rightsquigarrow l = L \wedge ws = WS.W \wedge i[l] = !I \wedge d[l][!I] = W \tag{47}$$

Again, this doesn't disturb stability in the reader: (46) can write to the $d[rt]$ pair, but only to the $!i[rt]$ slot; when (47) alters $i[rt]$ it updates ws consistently. We have an atomicity refinement.

Linearisation points in each of these verifications are marked by assignment to ws. Note that this means assignment to an element of d – a buffer slot – is no longer a linearisation point in either of the two cases $l = r$ (fig. 33, linearisation point $\langle\!\langle l := wt \rangle\!\rangle$) or $l \neq r$ (fig. 37, linearisation point $\langle\!\langle i[wt] := wi \rangle\!\rangle$). This fact will become important when we consider removing atomicity from buffer accesses.

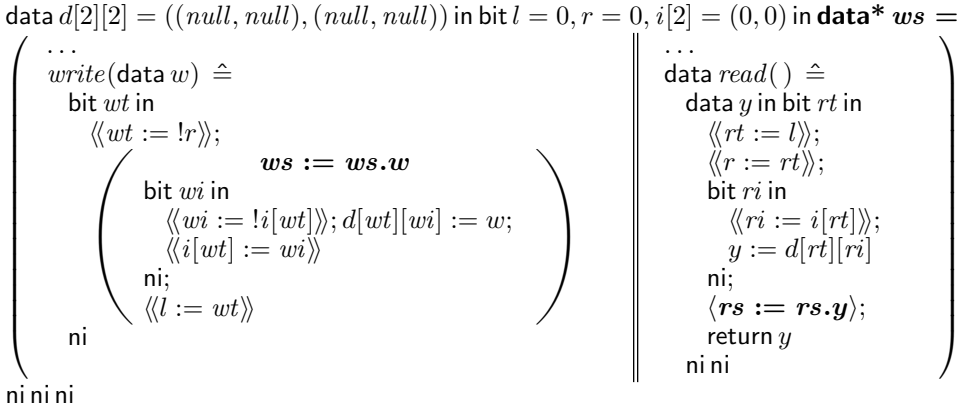


Fig. 40. Simpson's four-slot algorithm

11.2. Atomicity refinement of the four-slot reader

Atomicity refinement of the second step of fig. 35 is verified in fig. 38. Both steps generate the identity action but, as before, it is the stability step that requires most attention. Essentially, the writer can alter $i[L]$ but it can't alter $d[L][I']$. To understand this we appeal, as before, to a finite-state machine diagram of the writer's actions after $\langle\langle r := rt \rangle\rangle$. Fig. 39 is the refinement of fig. 30 after the reader action eliminates the two bottom-left states. We have produced this figure by using once again the technique of (33) (details omitted) to show that the writer's actions are

$$ac = 0 \wedge l = !r = L \rightsquigarrow ac = 2 \wedge l = !r = L \tag{48}$$

$$ac = 2 \wedge l = L \wedge i[l] = I \rightsquigarrow ac = 1 \wedge l = L \wedge i[l] = I \wedge d[l][!I] = W \tag{49}$$

$$ac = 1 \wedge l = L \wedge ws = WS \wedge i[l] = I \wedge d[l][!I] = W \rightsquigarrow ac = 0 \wedge l = L \wedge ws = WS.W \wedge i[l] = !I \wedge d[l][!I] = W \tag{50}$$

$$ac = 0 \wedge l = r = L \rightsquigarrow ac = 5 \wedge l = r = L \tag{51}$$

$$ac = 5 \wedge l = r = L \wedge i[!l] = I \rightsquigarrow ac = 4 \wedge l = r = L \wedge i[!l] = I \wedge d[!l][!I] = W \tag{52}$$

$$ac = 4 \wedge l = r = L \wedge i[!l] = I \rightsquigarrow ac = 3 \wedge l = r = L \wedge i[!l] = !I \tag{53}$$

$$ac = 3 \wedge l = r = L \wedge ws = WS \wedge i[!l] = I \wedge d[!l][I] = W \rightsquigarrow ac = 0 \wedge !l = r = L \wedge ws = WS.W \wedge i[l] = I \wedge d[l][I] = W \tag{54}$$

– more actions than before because some have been split into two stages. *None* of the actions in the diagram alter $d[L][i[L]]$. Actions (49) and (50) correspond to (35) in the two-slot algorithm, which wrote to $c[L]$ at the top left of fig. 30 and could produce a collision; but (49) writes to $d[L][!i[L]]$ and (50) inverts $i[L]$. The collision has gone and we have stability: after the writer has selected a pair and a slot within the pair – i.e. after $\langle\langle r := rt \rangle\rangle; \langle\langle ri := i[rt] \rangle\rangle$ – its value is stable. We have an atomicity refinement.

The linearisation point in the reader is at the earliest $\langle\langle rt := l \rangle\rangle$ and at the latest $\langle\langle ri := i[rt] \rangle\rangle$: after the second point writes no longer have any effect on the value returned. But as usual we have to say that it is the write which provided B', if $B' \neq B$; all we care about is that the linearisation point exists.

11.3. Four slots with non-atomic slot access

Atomicity refinement in the four-slot reader has produced an algorithm in which, once the reader has selected a slot, the writer *cannot* write to the selected slot. We no longer need atomic slot accesses, and fig. 40 is Simpson's classic four-slot algorithm [Sim90]. All we have done since atomicity refinement is to remove atomicity brackets round $d[wt][wi] := w$ in the writer and $y := d[rt][ri]$ in the reader. Linearisation points remain unchanged.

So far we have required that access to shared store take place only within an atomic command, so that we can associate actions with atomic executions. We suppose that non-atomic reads and writes take place as

$$\left\{ w = W \wedge wt = !L \wedge wi = !I' \wedge \boxed{\begin{array}{l} l = r = L = R \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge \\ i[L] = I \wedge d[L][I] = B \wedge i[!L] = I' \end{array}} \right\}$$

— $\{d[wt][wi] = X$ stable under R

$d[wt][wi] := w$

$$\left\{ w = W \wedge wt = !L \wedge wi = !I' \wedge \boxed{\begin{array}{l} l = r = L = R \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge \\ i[L] = I \wedge d[L][I] = B \wedge \\ i[!L] = I' \wedge d[!L][!I'] = W) \end{array}} \right\}$$

Fig. 41. Verification of four-slot non-atomic write, $l = r$ case

$$\left\{ \exists I' \left(rt = L \wedge ri = I' \wedge \boxed{\exists B' \left(\begin{array}{l} r = L \wedge rs = RS \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge \lfloor rs.B.B' \rfloor \preccurlyeq ws \wedge \\ \left(\begin{array}{l} (d[L][I'] = B') \wedge \\ \exists I''(i[l] = I'' \wedge d[l][I''] = ws_\Omega) \wedge \end{array} \right) \\ (i \mapsto (\_, \_) \wedge d \mapsto (\_, \_, \_, \_)) \end{array} \right)} \right) \right\}$$

— $\{d[rt][ri] = B'$ stable under R

$y := d[rt][ri]$

$$\left\{ \exists B' \left(y = B' \wedge \boxed{\begin{array}{l} r = L \wedge rs = RS \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge \lfloor rs.B.B' \rfloor \preccurlyeq ws \wedge \\ (\exists I''(i[l] = I'' \wedge d[l][I''] = ws_\Omega)) \end{array}} \right) \right\}$$

Fig. 42. Verification of four-slot non-atomic read

a series of atomic mini-writes, serialised by the store. So a non-atomic write alters its destination piecewise, a non-atomic read assembles its results piecewise; a non-atomic read will retrieve a non-coherent value if its mini-reads overlap with a sequence of mini-writes; and two writers can produce an incoherent result if their sequences of mini-writes overlap.

Vafeiadis's non-atomic write rule [Vaf07] applies to an assignment which copies a value computed from local variables into shared heap, and deals with cells which might go into and out of the heap. In our case we write from local store to a shared array element which cannot disappear. Specialising our concerns to two-dimensional arrays, we suppose that the non-atomic instruction $a[I][J] := E$ is executed as a sequence of k naturally-atomic assignments $\langle\!\langle a[I][J]_1 := E_1 \rangle\!\rangle; \langle\!\langle a[I][J]_2 := E_2 \rangle\!\rangle; \ldots; \langle\!\langle a[I][J]_k := E_k \rangle\!\rangle$. Adapting Vafeiadis's rule by replacing heap addressing with array indexing we derive the the rule

$$\frac{P \wedge I = X, P \wedge J = Y, P \wedge a[I][J] = Z \text{ stable under } R \quad P \rightsquigarrow P \wedge a[I][J] = Z \subseteq G}{\left\{E = F \wedge \boxed{P}\right\}\ a[I][J] := E\ \left\{E = F \wedge \boxed{P \wedge a[I][J] = F}\right\}}\ \text{non-atomic write} \tag{55}$$

There must be no competing writers to the array element ($P \wedge a[I][J] = Z$ stable under R); nothing can alter the index formulale I and J ($P \wedge I = X, P \wedge J = Y$ stable under R); E must be a local formula (though it would be enough for it to be stable); and the writer must declare its action to the environment ($P \rightsquigarrow P \wedge a[I][J] = Z \subseteq G$). We have all four conditions when we consider $d[wt][wi] := w$: the reader doesn't write to the array at all; it cannot alter wt or wi because they are local to the writer; w is local; and the effect of the assignment is included in the guarantee in actions (44) and (46). In the case that $wt = !L$ (fig. 41) we have the same pre and postcondition and the same action as in fig. 36. The other case, when $wt = L$, is equally straightforward (details omitted).

A non-atomic read rule which copies a value from a shared array element to a local variable is equally straightforward to derive

$$\frac{P \wedge I = X, P \wedge J = Y, P \wedge a[I][J] = Z \text{ stable under } R}{\left\{\boxed{\exists E(P \wedge a[I][J] = E)}\right\}\ x := a[I][J]\ \left\{\exists E\left(x = E \wedge \boxed{P \wedge a[I][J] = E}\right)\right\}}\ \text{non-atomic read} \tag{56}$$

– provided that the indices I and J and the value of the array element are stable under the actions of the environment, we are guaranteed to be able to read coherently. In the four-slot reader (fig. 42) we have the same pre and postcondition as in fig. 38 and the same identity action. We have completed our refinement.

The most important steps in the refinement process were all made in the two-slot case. We showed that the writer only moves l to make $l \neq r$ and the reader only moves r to make $r = l$; we showed that the writer does one thing when $l = r$ initially and another when $l \neq r$; we showed that the writer only collides with the reader briefly, once the reader has selected a slot. The rest was more or less mechanical.

The algorithm is evidently wait-free – there are no loops in either *write* or *read* procedure body – and it provably emulates the single-slot idealised original. It doesn't depend on hardware concurrency mechanisms, provided only that the store serialises single-bit reads and writes. That is a larger assumption than it may seem, and it is discussed further in §15. But, for the moment, we have achieved what we set out to do.

12. Dialectical refinement?

Conventional refinement, of whatever flavour, starts with a more or less abstract specification and a more or less magical algorithm and proceeds by refinement steps, reducing abstraction in the specification and/or magic in the algorithm but always preserving the truth of their correspondence, to reach an implementation.

In our refinement so far that is what we have done. There was a little dead end development to produce fig. 18, but we backtracked, re-refined to two slots in fig. 19, and proceeded by logical refinement to produce the four-slot algorithm of fig. 40.

But something else happened in that dead end. We discovered a bug – reading out of sequence – and constructed an algorithm to avoid it. That has a little of the dialectical in it: thesis (simple two-slot refinement) plus antithesis (observable bug / proof failure) yields synthesis (cleverer two-slot refinement). In producing Harris's three-slot algorithm we don't backtrack: we modify the faulty algorithm of fig. 18 to avoid the same bug, producing a radically different algorithm which isn't wait-free; then we modify that algorithm again to produce a wait-free algorithm. This has something of the quality, if not the elegance, of the dialectical developments described by Lakatos [Lak76]. We seem to proceed by a kind of monster-barring, and we certainly make steps which aren't truth-preserving.

In our experience, straying into the false in the course of a development is part of the algorithm-discovery process, and we feel it ought to be one of the ways in which they are explained and refined. We put forward our refinement of Harris's algorithm as an example.

13. Refinement to two slots – third attempt

The problem with fig. 18 is out-of-sequence reading. As we observed above in the refinement from two slots (reader chooses) to four slots, the problem arises in the *second* write after the reader has chosen a slot: the reader must read the current value of $l = L$, the writer must invert l and write to the $c[!l] = c[L]$ slot, and then the reader can retrieve the partially-written value – coherently, because buffer reads and writes in that algorithm are atomic, but not yet part of the written sequence and therefore out-of-sequence. Simpson's reader-chooses algorithm avoids the problem by signalling between reader and writer to alter the writer's behaviour, so that it doesn't always invert l. Harris's solution is to signal between writer and reader to alter the reader's behaviour, so that it doesn't rely on a value written after the writer inverts l, the first step in the sequence of its actions which can set up an out-of-sequence read.

In fig. 43, which is the atomically-misrefined writer-chooses algorithm of fig. 18 changed to make use of a signalling variable ok, the writer sets $\neg ok$ after it has inverted l. The reader sets ok when it starts, and if it manages to read before the writer sets $\neg ok$, it returns a value; if not, it simply tries again.[7] This is now an extremely 'racy' algorithm: the reader can retrieve possibly-incoherent or possibly-out-of-sequence values but if so it rejects them and tries again. The signalling is over-cautious – the writer sends the signal immediately it has inverted l, which may be long before it begins to write a new value to the buffer – but it is safe. If the reader retrieves a value before $\langle\langle ok := \text{false}\rangle\rangle$ then it is a fresh member of the written sequence. All this assumes, as in the refinement of Simpson's algorithm, that the writer's writes to the shared store happen in the sequence they are executed, and likewise the reader's reads and writes, and that the store

[7] In our development we access shared store only in atomic instructions or in carefully-checked non-atomic instructions. ok is in shared store so to treat it properly we fastidiously do not write $\mathsf{do} \ldots \mathsf{until}\ \neg ok$. Instead we read its value into a local variable in an atomic command, and loop on the value of that variable. It's a minor point but it keeps our development scrupulously honest.

```
data c[2] = (null, null) in bit l = 0, ok in data* ws = .null, rs = . in
(  ...                                  ||  ...
   write(data w) ≙                      ||  data read( ) ≙
     bit wt in                          ||    data y in bit rt in
       ⟨⟨wt := !l⟩⟩;                    ||      do
       ⟨c[wt] := w⟩;                    ||        ⟨⟨ok := true⟩⟩;
       ⟨⟨l := wt; ws := ws.w⟩⟩;          ||        ⟨⟨rt := l⟩⟩;
       ⟨⟨ok := false⟩⟩                  ||        ⟨y := c[rt]⟩;
     ni                                 ||        ⟨⟨rt := ok⟩⟩
                                        ||      until rt;
                                        ||      ⟨⟨rs := rs.y⟩⟩;
                                        ||      return y
                                        ||    ni ni
) ni ni ni
```

Fig. 43. A danger-signalling two-slot algorithm with a changed specification

serialises single-bit shared-memory accesses. These are not trivial assumptions: see §15 for a discussion of what might happen on a modern processor.

This algorithm has the same invariant (14) as previous two-slot algorithms. But in one respect it is distinctly inferior to the reader-chooses algorithm, because it is no longer wait-free. Previously the $write(w)$ and $read(\,)$ procedure calls were guaranteed to terminate, because their procedure bodies were sequences of straightforward certain-to-terminate assignment instructions. Now we have a loop in the reader, and there is no guarantee that it will ever terminate: the writer may write so often that the reader never has a chance to read a value before the writer sets $\neg ok$. We can fix that problem to produce a wait-free algorithm, but for the time being we explore this version, in which $y := b$ has been refined by a loop, to explain how the signalling works.

The actions of the writer in fig. 43 are (18) and (19) from its second and third steps, just as in the writer-chooses algorithm, plus from the fourth step either the identity action, when $\neg ok$, or

$$ok \rightsquigarrow \neg ok \tag{57}$$

Each of these actions preserves the invariant (14): (57) because the invariant doesn't mention ok; (18), as before, because the invariant doesn't mention $c[!l]$; (19), as before, extends ws and alters l to preserve $c[l] = ws_\Omega$. We omit most of the verification of the writer, which is identical to fig. 15. The final step is

$$\begin{array}{l} \Big\{ w = W \wedge \boxed{l = !L \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B \wedge c[!L] = W} \Big\} \\ \quad \langle\langle ok := \text{false} \rangle\rangle \\ \Big\{ w = W \wedge \boxed{\neg ok \wedge l = !L \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B \wedge c[!L] = W} \Big\} \end{array} \tag{58}$$

which clearly generates (57).

We shall see that the actions of the reader are (13), as in the atomicity-refined single-variable buffer, plus

$$\neg ok \rightsquigarrow ok \tag{59}$$

These actions preserve (14), and they certainly don't affect stability arguments in verifications fig. 15 and (58). It remains to show that these truly are the actions of the reader.

Action sequencing is at the heart of this design. It's easy to show by adding an action-sequencing auxiliary variable (details omitted) that after $\langle\langle ok := \text{true} \rangle\rangle$ in the reader the writer's actions change the shared store according to the finite state machine shown in fig. 44. The dangerous action, the one which can cause an out-of-sequence read, is (18) when $l = !L$. But it's clear from the diagram that this can only happen when $\neg ok$. As long as ok holds, the value in $c[L]$ is a member of the written sequence.

We verify the body of the loop in the reader, starting with the precondition from fig. 16, in fig. 45. After the first step ok holds instantaneously. In the stability weakening step we have to forget the unstable values of ok, ws l and $c[l]$, but so long as ok holds the writer has not overwritten $c[L]$ whether or not l has changed; also we record that B is a valid appendage to rs. After the second step we have fixed on the value of l, which

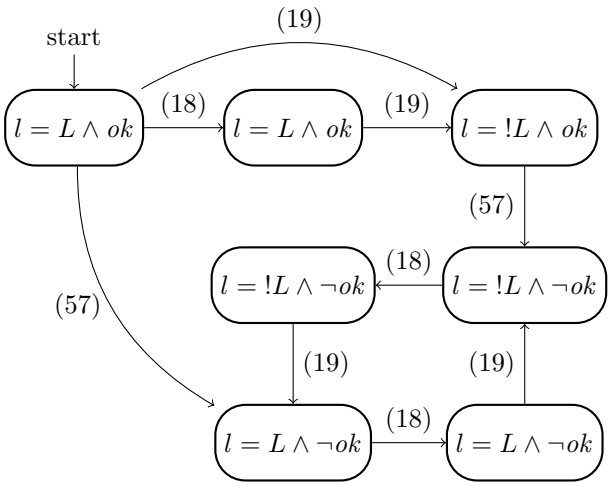


Fig. 44. Sequencing of danger-signalling writer's actions

$$\left\{\boxed{l = L \wedge rs = RS \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[l] = B}\right\}$$
$$\langle\!\langle ok := \text{true} \rangle\!\rangle$$
$$\left\{\boxed{l = L \wedge ok \wedge rs = RS \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[l] = B}\right\} \;\Rightarrow\; \text{(stability)}$$
$$\left\{\boxed{\exists B', B'' \left(\begin{array}{c} (ok \Rightarrow (l = L \wedge ws = WS.B) \vee (l = !L \wedge ws = WS.B.B' \wedge B = B'')) \wedge \\ rs = RS \wedge \lfloor rs.B \rfloor \preccurlyeq ws \wedge ws_\Omega = B' \wedge c[l] = B' \wedge c[!l] = B'' \end{array} \right)}\right\}$$
$$\langle\!\langle rt := l \rangle\!\rangle$$
$$\left\{\exists L' \left(rt = L' \wedge \boxed{\exists B', B'' \left(\begin{array}{c} (ok \Rightarrow (l = L \wedge ws = WS.B) \vee (l = !L \wedge ws = WS.B.B' \wedge B = B'')) \wedge \\ l = L' \wedge rs = RS \wedge \lfloor rs.B \rfloor \preccurlyeq ws \wedge ws_\Omega = B' \wedge c[l] = B' \wedge c[!l] = B'' \end{array} \right)} \right)\right\}$$
$$\Rightarrow \text{(stability)}$$
$$\left\{\exists L' \left(rt = L' \star \boxed{\exists B', B'' \left(\begin{array}{c} (ok \Rightarrow !l = L' \Rightarrow ws = WS.B.B' \wedge B = B'') \wedge \\ rs = RS \wedge \lfloor rs.B \rfloor \preccurlyeq ws \wedge ws_\Omega = B' \wedge c[l] = B' \wedge c[!l] = B'' \end{array} \right)} \right)\right\}$$
$$\langle y := c[rt] \rangle$$
$$\left\{\exists B''' \left(y = B''' \wedge \boxed{(ok \Rightarrow \lfloor RS.B''' \rfloor \preccurlyeq ws) \wedge rs = RS \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[l] = ws_\Omega} \right)\right\}$$
$$\langle\!\langle rt := ok \rangle\!\rangle$$
$$\left\{ \exists B''' \left(y = B''' \wedge \left(\begin{array}{l} \left(rt \wedge \boxed{ok \wedge rs = RS \wedge \lfloor RS.B''' \rfloor \preccurlyeq ws \wedge c[l] = ws_\Omega} \right) \;\vee \\ \left(\neg rt \wedge \boxed{\neg ok \wedge rs = RS \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[l] = ws_\Omega} \right) \end{array} \right) \right) \right\}$$

Fig. 45. Verification of two-slot reader (danger-signalling)

can still change, but, so long as ok holds, the value in either slot of the buffer array is a valid appendage to rs. After the third step either we have the latest value, if $ok \wedge l = L'$, or B, if $ok \wedge !l = L'$, or some other value, if $\neg ok$ – so a valid appendage to rs if ok still holds. After the fourth step the information about freshness is transferred to the local state.

The only action generated by the reader is (59) from the first step. We can use this verification to produce a specification of the loop in fig. 46. This gives us a loop postcondition which shows that $\langle\!\langle \boldsymbol{rs := rs.y} \rangle\!\rangle$ produces the action (13), as in the single-variable verification fig. 9.

$$\left\{\boxed{rs = RS \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[l] = ws_\Omega}\right\}$$
do
$$\left\{\boxed{\exists L, WS, B\,(\, l = L \wedge rs = RS \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[l] = B\,)}\right\}$$
$$\langle\!\langle ok := \text{true} \rangle\!\rangle; \langle\!\langle rt := l \rangle\!\rangle; \langle y := c[rt] \rangle; \langle\!\langle rt := ok \rangle\!\rangle$$
$$\left\{ \exists B''' \left(y = B''' \wedge \left(\begin{array}{l} \left(rt \wedge \boxed{ok \wedge rs = RS \wedge \lfloor RS.B''' \rfloor \preccurlyeq ws \wedge c[l] = ws_\Omega} \right) \vee \\ \left(\neg rt \wedge \boxed{\neg ok \wedge rs = RS \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[l] = ws_\Omega} \right) \end{array} \right) \right) \right\}$$
$$\Rightarrow \text{ (stability)}$$
$$\left\{ \exists B''' \left(y = B''' \wedge \left(\begin{array}{l} \left(rt \wedge \boxed{rs = RS \wedge \lfloor RS.B''' \rfloor \preccurlyeq ws \wedge c[l] = ws_\Omega} \right) \vee \\ \left(\neg rt \wedge \boxed{\neg ok \wedge rs = RS \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[l] = ws_\Omega} \right) \end{array} \right) \right) \right\}$$
until rt
$$\left\{ \exists B''' \left(y = B''' \wedge \boxed{rs = RS \wedge \lfloor RS.B''' \rfloor \preccurlyeq ws \wedge c[l] = ws_\Omega} \right) \right\}$$

Fig. 46. Specification of loop in two-slot reader (danger-signalling)

The linearisation point in the writer is clearly $\langle\!\langle l := wt \rangle\!\rangle$. The linearisation point in the reader is actually the last execution of $\langle\!\langle rt := l \rangle\!\rangle$ before it returns: after that point nothing the writer did had any effect.

We have repaired fig. 18: fig. 43 shows freshness at the expense of losing the wait-free property. Wait-freeness is important: we cannot be sure in this algorithm that a *read*( ) procedure call will ever terminate. We note, however, that this may not be a crippling problem in practice. The Ethernet communication algorithm, for example, isn't wait-free – there can be collisions between competing transmitters, and if there is, both transmitters loop – but it's certainly practically useful. The danger-signalling algorithm, like the Ethernet algorithm, is *obstruction-free*; if the writer is passive, the reader's loop is guaranteed to terminate. Obstruction-free algorithms can be made practical by *back-off* schemes: if there is a collision, both sides pause for an interval, likely to be different on each side, and then try again. We don't explore back-off in this case, because it is possible to restore wait-freeness by a further refinement.

13.1. Non-atomic buffer accesses in the danger-signalling algorithm

The stability argument in fig. 45 relies on the fact that so long as ok holds, the value in slot $c[rt]$ won't be overwritten. That is, there is no collision so long as ok holds, and therefore we can read and write buffer slots *non*-atomically so long the reader ignores the result if $\neg ok$ – which it does. This won't affect linearisation points, which aren't on the buffer access instructions.

The writer is straightforward using the non-atomic write rule (55)

$$\begin{array}{l} \left\{ w = W \wedge wt = !L \wedge \boxed{l = L \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B} \right\} \\ \quad — \{c[wt] = X \text{ stable under } R \\ c[wt] := w \\ \left\{ w = W \wedge wt = !L \wedge \boxed{l = L \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B \wedge c[!L] = W} \right\} \end{array} \tag{60}$$

In the reader the non-atomic read rule (56) is inadequate because $c[rt] = B$ is not stable. To derive an adequate rule[8] we note again that a non-atomic read will be executed as a sequence of atomic mini-reads. Focussing this time on single-dimensioned array access, we suppose that x and $a[I]$ are made up of k separate pieces, and a non-atomic read $x := a[I]$ is executed as the sequence $\langle\!\langle x_1 := a[I]_1 \rangle\!\rangle; \langle\!\langle x_2 := a[I]_2 \rangle\!\rangle; \ldots; \langle\!\langle x_k :=$

[8] The derivation of rule (62) and the rule itself were suggested to us by Vafeiadis and Parkinson (private communication).

$a[I]_k\rangle\!\rangle$, and that there is some formula Q which guarantees that $a[I] = V$. Then for any particular element of the sequence of atomic mini-reads we may deduce, using the specification-disjunction rule

$$\begin{array}{l}\left\{\boxed{(\neg Q \wedge c[L] = \_) \vee (Q \wedge c[L] = V)}\right\} \\ \quad \langle\!\langle x_i := a[I]_i \rangle\!\rangle \\ \left\{\exists V_i' \left(x_i = V_i' \wedge \boxed{(\neg Q \wedge c[L] = \_) \vee (Q \wedge V_i' = V_i \wedge c[L] = V)}\right)\right\}\end{array} \tag{61}$$

– we have read a segment of $a[I]$ and if Q holds, a segment of V. If $\neg Q$ is stable, so that we cannot have first Q then $\neg Q$ then Q, then if Q holds at the end of the sequence of atomic reads it must have held throughout, and we can conclude that we have successfully read all k segments of V, the value in $a[I]$.

The following rule is thus derived to deal with a non-atomic read from a shared array element to a local variable, splitting Q into $P \wedge OK$:

$$\frac{\exists V((OK \Rightarrow V = X) \wedge P(V) \wedge a[I] = V),\ \neg OK \text{ stable under } R}{\left\{\boxed{\exists V(P(V) \wedge a[I] = V)}\right\}\ x := a[I]\ \left\{\exists V' \left(x = V' \wedge \boxed{\exists V((OK \Rightarrow V = V') \wedge P(V) \wedge a[I] = V)}\right)\right\}}\ \text{non-atomic read} \tag{62}$$

- the value of the array element must be stable throughout the read: $\exists V(P(V) \wedge a[I] = V)$ stable under R;
- when OK holds, the value in the array element is fixed: $OK \Rightarrow V = X$ stable under R;
- OK does not change value from false to true during the read: $\neg OK$ stable under R.

In the reader we have

$$\begin{array}{l}\left\{\exists L' \left(rt = L' \wedge \boxed{\exists B', B'' \left(\begin{array}{l}(ok \Rightarrow\, !l = L' \Rightarrow ws = WS.B.B' \wedge B = B'') \wedge \\ \quad rs = RS \wedge \lfloor rs.B \rfloor \preccurlyeq ws \wedge ws_\Omega = B' \wedge c[l] = B' \wedge c[!l] = B''\end{array}\right)}\right)\right\} \\ \quad — \ \{\neg ok, ok \Rightarrow c[rt] = X \text{ stable under } R \\ \quad y := c[rt] \\ \left\{\exists B''' \left(y = B''' \wedge \boxed{(ok \Rightarrow \lfloor RS.B''' \rfloor \preccurlyeq ws) \wedge rs = RS \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[l] = ws_\Omega}\right)\right\}\end{array} \tag{63}$$

We have a two-slot algorithm which communicates according to (14). It has coherence and freshness.

14. Refinement to three slots

The writer in fig. 43 signals when a write which sets up an out-of-sequence read might be its next action. If it signalled also the latest value it has written then the reader, when it receives the signal, could use the signalled value instead of trying to read again from the c array. We use a shared variable c', in effect a third buffer slot, for this auxiliary communication.

It is tempting to replace $\langle\!\langle ok := \text{false} \rangle\!\rangle$ in the writer by $\langle c' := w; ok := \text{false} \rangle$ but that isn't a refinement because the original generates the identity action when $\neg ok$. We need to wrap the signalling instruction in a conditional, as shown in fig. 47. Since our rules don't let us access shared memory in the guard of a conditional, we fastidiously read the value of ok into a local variable and test that variable.

Using the precondition of $\langle\!\langle ok := \text{false} \rangle\!\rangle$ from (58), we produce the verification in fig. 48. This gives the same postcondition as (58), except that we have forgotten $\neg ok$ for stability in the second arm of the disjunction; this postcondition would fit into any sequential context as well as the pre-refinement version. Action (57) is replaced by

$$ok \wedge ws = WS.W \rightsquigarrow \neg ok \wedge ws = WS.W \wedge c' = W \tag{64}$$

– clearly a data refinement because it changes ok as before and otherwise only modifies the new variable c'.

We verify the reader in fig. 49. We can assert, in the stability step which follows $\langle\!\langle ok := \text{true} \rangle\!\rangle$, not only what ok implies, but also, because of (64), which is the only writer action to set $\neg ok$, $\neg ok \Rightarrow \lfloor rs.c' \rfloor \preccurlyeq ws$. We have the same postcondition (modulo alpha conversion) as in fig. 46, but without using a loop. Every step except for the first generates the identity action; the first still generates (59). We have recovered a wait-free algorithm.

```
data c[2] = (null, null), c′ in bit l = 0, ok in data* ws = .null, rs = . in
(  ...                                  ||  ...
   write(data w) ≙                      ||  data read( ) ≙
     bit wt in                          ||    data y in bit rt in
       ⟪wt := !l⟫;                      ||      ⟪ok := true⟫;
       c[wt] := w;                      ||      ⟪rt := l⟫;
       ⟪l := wt; ws := ws.w⟫;           ||      y := c[rt];
       ⟪wt := ok⟫;                      ||      ⟪rt := ok⟫;
       if wt then                       ||      if ¬rt then⟨y := c′⟩ else skip fi;
          ⟨c′ := w; ok := false⟩        ||      ⟪rs := rs.y⟫;
       else skip fi                     ||      return y
     ni                                 ||    ni ni
)
ni ni ni
```

Fig. 47. A value-signalling three-slot algorithm

14.1. Atomicity refinement in the three-slot algorithm

Because the writer only writes to c' when ok and the reader only reads when $\neg ok$ we can refine away all the remaining non-hardware atomicity in our algorithm. First we refine the writer, replacing $\langle c' := w; ok := \text{false}\rangle$ with $c' := w; \langle\langle ok := \text{false}\rangle\rangle$. In fig. 50 we have the same pre and postcondition as in fig. 48. The verification uses a non-atomic write rule just like (55) but with fewer sideconditions because assignment is from a shared variable rather than an array element. The two steps in this verification generate the actions

$$ok \wedge c' = C \rightsquigarrow ok \wedge c' = C' \tag{65}$$

$$ok \wedge ws = WS.W \wedge c' = W \rightsquigarrow \neg ok \wedge ws = WS.W \wedge c' = W \tag{66}$$

(65) doesn't affect stability in the writer, whose assertions when ok don't mention c', and (66) is included in (64), which is already allowed for.

In the reader we can use a version of (62) with fewer sideconditions because we are reading from a shared variable rather than an array element. Verification in fig. 51, shows the same pre and postcondition as in fig. 49, and the same identity action. We have an atomicity refinement.

14.2. Not quite Harris's algorithm

Fig. 52 is almost Harris's three-slot algorithm as communicated to us. The conditional assignments to c' and ok were at the beginning of Harris's writer; we think our placing fits explanation by refinement better, and there may even be some very minor efficiency advantage (but we wouldn't dare defend that hypothesis: measuring the efficiency of concurrent algorithms is well beyond our expertise). The only significant difference is in the reader: instead of skip as the alternative to $y := c'$, Harris had $\langle\langle ok := \text{false}\rangle\rangle$. Any programmer would recognise the motivation: the reader can cancel its signal once it's read an acceptable value, and the writer may then be able to avoid writing unnecessarily to c'.

It is easy for a programmer to see that this would be a trivial change, and possible to verify the changed algorithm, but we don't do that here. It requires a little action-sequencing in the reader, and we don't think that to do so would add to our explanation by refinement. We have a three-slot algorithm to set against Simpson's four-slot original.

Linearisation is simpler than it was for Simpson's algorithm. In the writer the linearisation point is $\langle\langle l := wt\rangle\rangle$, annotated with an assignment to ws. In the reader the linearisation point depends on what the reader does. If the reader sees true when it executes $\langle\langle rt := ok\rangle\rangle$ then the linearisation point was $\langle\langle rt := l\rangle\rangle$ as in the danger-signalling two-slot algorithm. If it sees false then the linearisation point is when the writer executed $\langle\langle ok := \text{false}\rangle\rangle$: after that point it was determined that the reader would have to take the value in c'; no subsequent calls of the write procedure have any effect. So this algorithm, like Simpson's, is linearisable and the original idealised single-variable buffer is a specification of its communication behaviour.

$$\left\{w = W \wedge \boxed{l = !L \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B \wedge c[!L] = W}\right\}$$
$$\langle\!\langle wt := ok \rangle\!\rangle$$
$$\left\{\begin{array}{l}\left(\begin{array}{l} w = W \wedge wt \wedge \\ \boxed{ok \wedge l = !L \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B \wedge c[!L] = W}\end{array}\right) \vee \\ \left(\begin{array}{l} w = W \wedge \neg wt \wedge \\ \boxed{\neg ok \wedge l = !L \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B \wedge c[!L] = W}\end{array}\right)\end{array}\right\}$$
$$\Rightarrow \text{ (stability)}$$
$$\left\{\begin{array}{l}\left(\begin{array}{l} w = W \wedge wt \wedge \\ \boxed{ok \wedge l = !L \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B \wedge c[!L] = W}\end{array}\right) \vee \\ \left(\begin{array}{l} w = W \wedge \neg wt \wedge \\ \boxed{l = !L \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B \wedge c[!L] = W}\end{array}\right)\end{array}\right\}$$
$$\textsf{if } wt \textsf{ then}$$
$$\left\{w = W \wedge wt \wedge \boxed{ok \wedge l = !L \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B \wedge c[!L] = W}\right\}$$
$$\langle c' := w; ok := \text{false} \rangle$$
$$\left\{w = W \wedge wt \wedge \boxed{c' = W \wedge \neg ok \wedge l = !L \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B \wedge c[!L] = W}\right\}$$
$$\Rightarrow \text{ (stability)}$$
$$\left\{w = W \wedge wt \wedge \boxed{c' = W \wedge l = !L \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B \wedge c[!L] = W}\right\}$$
$$\textsf{else}$$
$$\left\{w = W \wedge \neg wt \wedge \boxed{l = !L \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B \wedge c[!L] = W}\right\}$$
$$\textsf{skip}$$
$$\left\{w = W \wedge \neg wt \wedge \boxed{l = !L \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B \wedge c[!L] = W}\right\}$$
$$\textsf{fi}$$
$$\left\{w = W \wedge \boxed{l = !L \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B \wedge c[!L] = W}\right\}$$

Fig. 48. Verification of three-slot writer, signalling phase

15. Weak memory

Computing practice always runs ahead of computer science. Hardware always has more capability than it is possible to describe formally in any particular programming language, and programmers have always found ways to exploit that capability. Our proofs have relied on an assumption that single-bit writes and reads – or, more realistically, writes and reads of single-bit values stored in single-word variables – are serialised by a computer store and visible to all threads as soon as they have occurred. This is a reasonable assumption at the level of the store, but unreasonable when we consider modern shared store technology.

Programs in a modern processor do not access the store directly but via a storage buffer, which records pending writes, and a cache, which records completed reads. The processor can do hundreds or thousands of operations in the time it takes to read a single data item from the main store. Data read from memory is held in the cache and can be re-accessed in a few processor cycles; data to be written is held in the storage

$$\left\{\boxed{\exists C'\,(\,l = L \wedge rs = RS \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c' = C' \wedge c[l] = B\,)}\right\}$$
$$\langle\!\langle ok := \text{true} \rangle\!\rangle$$
$$\left\{\boxed{l = L \wedge ok \wedge rs = RS \wedge ws = WS.B \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[l] = B}\right\} \;\Rightarrow\; \text{(stability)}$$
$$\left\{\boxed{\exists B', B'' \left(\begin{array}{l}(ok \Rightarrow (l = L \wedge ws = WS.B) \vee (l = !L \wedge ws = WS.B.B' \wedge B = B'')) \wedge \\ \quad (\neg ok \Rightarrow \lfloor rs.c' \rfloor \preccurlyeq ws) \wedge \\ \qquad rs = RS \wedge \lfloor rs.B \rfloor \preccurlyeq ws \wedge ws_\Omega = B' \wedge c[l] = B' \wedge c[!l] = B''\end{array}\right)}\right\}$$
$$\langle\!\langle rt := l \rangle\!\rangle; \langle y := c[rt] \rangle; \langle\!\langle rt := ok \rangle\!\rangle$$
$$\left\{\exists B'''\left(y = B''' \wedge \left(\begin{array}{l}\left(rt \wedge \boxed{rs = RS \wedge \lfloor rs.B''' \rfloor \preccurlyeq ws \wedge c[l] = ws_\Omega}\right) \vee \\ \left(\neg rt \wedge \boxed{\neg ok \wedge rs = RS \wedge \lfloor rs.c' \rfloor \preccurlyeq ws \wedge c[l] = ws_\Omega}\right)\end{array}\right)\right)\right\}$$
$$\textsf{if } \neg rt \textsf{ then}$$
$$\left\{\exists B'''\left(y = B''' \wedge \boxed{\neg ok \wedge rs = RS \wedge \lfloor rs.c' \rfloor \preccurlyeq ws \wedge c[l] = ws_\Omega}\right)\right\}$$
$$\langle y := c' \rangle$$
$$\left\{\exists C'\left(y = C' \wedge \boxed{\neg ok \wedge c = C' \wedge rs = RS \wedge \lfloor rs.C' \rfloor \preccurlyeq ws \wedge c[l] = ws_\Omega}\right)\right\}$$
$$\textsf{else}$$
$$\left\{\exists B'''\left(y = B''' \wedge \boxed{rs = RS \wedge \lfloor RS.B''' \rfloor \preccurlyeq ws \wedge c[l] = ws_\Omega}\right)\right\}$$
$$\textsf{skip}$$
$$\left\{\exists B'''\left(y = B''' \wedge \boxed{rs = RS \wedge \lfloor RS.B''' \rfloor \preccurlyeq ws \wedge c[l] = ws_\Omega}\right)\right\}$$
$$\textsf{fi}$$
$$\left\{\exists Y\left(y = Y \wedge \boxed{rs = RS \wedge \lfloor RS.Y \rfloor \preccurlyeq ws \wedge c[l] = ws_\Omega}\right)\right\}$$

Fig. 49. Verification of three-slot reader

$$\left\{\begin{array}{l}w = W \wedge wt \wedge \\ \quad \boxed{ok \wedge l = !L \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B \wedge c[!L] = W}\end{array}\right\}$$
$$\text{— } \{ok \wedge c' = C' \text{ stable under } R; ok \wedge c' = X \rightsquigarrow ok \wedge c' = X' \subseteq G$$
$$c' := w$$
$$\left\{\begin{array}{l}w = W \wedge wt \wedge \\ \quad \boxed{c' = W \wedge ok \wedge l = !L \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B \wedge c[!L] = W}\end{array}\right\}$$
$$\langle\!\langle ok := \text{false} \rangle\!\rangle$$
$$\left\{\begin{array}{l}w = W \wedge wt \wedge \\ \quad \boxed{c' = W \wedge \neg ok \wedge l = !L \wedge ws = WS.B.W \wedge \lfloor rs \rfloor \preccurlyeq ws \wedge c[L] = B \wedge c[!L] = W}\end{array}\right\}$$

Fig. 50. Atomicity refinement of signalling operation in three-slot writer

$$\left\{\exists B'''\left(y = B''' \wedge \boxed{\neg ok \wedge rs = RS \wedge \lfloor rs.c' \rfloor \preccurlyeq ws \wedge c[l] = ws_\Omega}\right)\right\}$$
$$\text{— } \{\neg ok \wedge c' = C, \neg ok \text{ stable under } R$$
$$y := c'$$
$$\left\{\exists C'\left(y = C' \wedge \boxed{\neg ok \wedge c = C' \wedge rs = RS \wedge \lfloor rs.C' \rfloor \preccurlyeq ws \wedge c[l] = ws_\Omega}\right)\right\}$$

Fig. 51. Atomicity refinement of value-signal assignment in three-slot reader

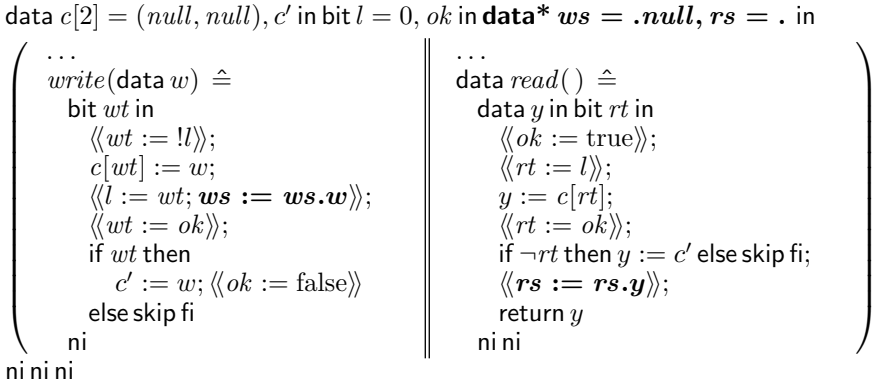


Fig. 52. Harris's three-slot algorithm, minus a pacifying signal

buffer so that it can be re-read or overwritten very rapidly. From the point of view of a program running on a single processor, nothing has changed: values are read and written just as if the cache and buffer were not there. But from the point of view of an external observer, the store does not reflect the writes made by the program unless and until the buffer is empty. So-called *memory barrier* or *fence* instructions force the processor to wait until its storage buffer is empty, but they are expensive, costing many thousands of processor cycles in practice. To add to the complexity, reads and writes may be re-ordered by the processor, and there can be corresponding barriers/fences to prevent one kind of reordering or another.

If Simpson's reader and writer are running on separate processors things therefore become more complicated than our development suggests. The reader's $\langle\langle r := rt \rangle\rangle$, which is crucial for signalling the fact that the reader is working in the rt pair, will not have an effect immediately. That means that we cannot conclude in the shared-memory postcondition that $r = L$ immediately and say that the writer will not write twice in the L pair. We cannot even say that $d[r][i[r]] = B$ is stable, because the writer's last update of one or other of the array elements might not yet have been flushed from its buffer.

Processors are intricate concurrent machines in themselves. They are nowadays designed with so-called *weak memory models* which cover their behaviour when updating or reading from memory and describe how they interact with cache and buffer. Those models do not always correspond to processor behaviour, and correct models are a matter of current research [AFI+09, OSS09, SSZN+09, SSA+11, SSO+10]. The consequence for the program prover is not only that semicolon has ceased to be a sequencing operator – writes and reads can occur out of sequence from the point of view of other processors – but that we do not and at present cannot have a formalism which deals with the problem.

Suppose that the writes of each processor are performed in the order they are generated, that the read caches of both processors remain consistent with the store, but that if there is a pending write in the store buffer, the processor reads the value of that write rather than the value in the cache. This could in fact be the memory model of the x86 processor [OSS09]. Under these assumptions, after the reader executes $\langle\langle ok := \text{true} \rangle\rangle$ then when it later executes $\langle\langle rt := ok \rangle\rangle$ it may see true because its own write is still pending, even though the store may contain false, and therefore it can't be sure that the writer hasn't yet overwritten the buffer slot it read from.

It seems that with this memory model both Simpson's and Harris's algorithms would work properly if we placed a memory barrier instruction after each write to the single-bit signalling variables (l, r and $i[\_]$ in Simpson's algorithm, l and ok in Harris's algorithm), forcing the processor to wait until all its pending writes had been completed. But without a clear semantics of the memory model and a formal argument sound against those semantics, how could we be sure? Researchers who devise concurrent algorithms cope with this situation by trial and error, and lots and lots of soak testing, and for the time being it seems to the state of the art –though recently some significant progress has been made in reasoning about programs under one weak memory model [Owe10]).

Effort expended on verifying concurrent algorithms without a formal weak memory model is not, in our opinion, wasted. We are finding out how to prove that such algorithms correspond to their specification under certain assumptions. When the assumptions change the algorithms and/or the proofs have to be remade. Algorithms like Simpson's and Harris's will perhaps turn out to be unsuited to the modern hardware world because they need too many memory barrier instructions, stopping the processor too often and for too long.

Or perhaps that won't be a problem, and they will turn out to be useful. We did not choose to study these algorithms because we thought they are practical, but because we were sure that they were difficult. Our aim is to extend our techniques so that we can deal with the algorithms that are actually used, and they are at least as tricky as these ones. We still cannot do as well as we need to, and that must be remedied.

16. Summary and Conclusion

We started with the obvious but magically-atomic algorithm of fig. 2; refined it to the two-slot but still magically-atomic writer-chooses fig. 18; noted that there's an out-of-sequence bug; retreated to the reader-chooses algorithm fig. 19; and refined that to the four-slot algorithm of fig. 40. The proofs along the way were intricate, and we had to hide some complexities of action-sequencing and auxiliary assignment placement to make them at all palatable. In an alternative development we made more radical programming interventions to fix the problems of the writer-chooses algorithm, producing the danger-signalling two-slot algorithm of fig. 43; noted that this is not wait-free; made it use atomic buffer accesses nevertheless; and finally refined it to the three-slot algorithm of fig. 52

Our original intention was to test our formalism against the problem of Simpson's 1990 algorithm and to try to develop a beautiful explanatory proof. We think that we have produced an explanatory refinement, which brings out the essential characteristics of the algorithm more clearly than earlier attempts, but our refinement proofs are surely still too tedious to be beautiful, and we believe that the tedium may say something about the problem. Simpson's algorithm works like a state machine, and that comes out in all those formal verifications that we have seen or attempted. The decision to do without auxiliary action-sequencing variables in our proof forced us to rely informally on finite-state diagrams. Simpson's and Harris's algorithms have few enough states that they can be checked by a model-checker. It would be reasonable to conclude that model-checking rules in this case, but it would be just as reasonable to say that the essential character of Simpson's algorithm lies in the sequencing of actions and in the outlawing of certain dangerous sequences of actions. Our approach brings out that essence, we believe.

Acknowledgements

Jon Burton introduced us to the problem of Simpson's algorithm. We're grateful to Matthew Parkinson, Viktor Vafeiadis and Mike Dodds for indulging our requests for illumination about RGSep, and to Peter O'Hearn for setting us on the refinement path. Viktor, in particular, suggested the approach to linearisation of auxiliary commands which we used. Although this isn't a competition, and although we have taken our own approach, we are grateful to Cliff Jones and Jean-Raymond Abrial for showing us their own draft treatments of Simpson's algorithm, which were inspiring and awe-inspiring in equal measure. Thanks to Neil Henderson for pointing out to us that the bug we thought we had found in Simpson's writer-chooses algorithm is in fact a well-known and well-documented feature. Thanks to Hugo Simpson, who read an earlier version of this paper and corrected our misconceptions about the origins of his algorithm and the freshness problem. And finally we are grateful to the referees, in particular for forcing us to recognise that we weren't exploiting separation logic and should recast our proof without it.

References

[Abr08] Jean-Raymond Abrial. Formal development of Simpson's 4-slot algorithm. Private communication, July 2008. 2

[AC06] Jean-Raymond Abrial and Dominique Cansell. Formal development of Simpson's 4-slot algorithm. Private communication, March 2006. 2

[AFI+09] Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. The semantics of Power and ARM multiprocessor machine code. In *DAMP 2009: Workshop on Declarative Aspects of Multicore Programming*, January 2009. 37

[BA10] Richard Bornat and Hasan Amjad. Inter-process Buffers in Separation Logic with Rely-Guarantee. *Formal Aspects of Computing*, 22(6):735–722, November 2010. 2, 3, 39

[Bri02] P. Brinch Hansen, editor. *The Origin of Concurrent Programming*. Springer-Verlag, 2002. 38

[Dij65] Edsger W. Dijkstra. Cooperating Sequential Processes. Technical Report EWD-123, Technical University, Eindhoven, 1965. Reprinted in [Gen68] and [Bri02]. 2

[Flo67] Robert W. Floyd. Assigning Meanings to Programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society. 16

[Gen68] F. Genuys, editor. *Programming Languages*. Academic Press, 1968. 38

[Har04] Tim Harris. A non-blocking three-slot buffer. Private communication; described in [BA10], 2004. 2

[Hen03] Neil Henderson. Proving the Correctness of Simpson's 4-slot ACM using an Assertional Rely-Guarantee Proof Method. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME*, volume 2805 of *Lecture Notes in Computer Science*, pages 244–263. Springer, 2003. 2

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991. 2

[HP02] Neil Henderson and S. Paynter. The Formal Classification and Verification of Simpson's 4-Slot Asynchronous Communication Mechanism. In *FME 2002:Formal Methods—Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 121–132, Berlin / Heidelberg, 2002. Springer. 2

[HW90] Maurice Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. 7, 17

[Jon83] Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983. 3

[JP08] Cliff B. Jones and Ken G. Pierce. Splitting atoms with rely/guarantee conditions coupled with data reification. In *ABZ '08: Proceedings of the 1st international conference on Abstract State Machines, B and Z*, pages 360–377, Berlin, Heidelberg, 2008. Springer-Verlag. 2, 6, 7

[JP09] Cliff B. Jones and Ken G. Pierce. Elucidating concurrent algorithms via layers of abstraction and reification. Technical Report CS-TR-1166, Newcastle University, Computing Science, 2009. 2, 6, 7

[Lak76] Imre Lakatos. *Proofs and Refutations: the logic of mathematical discovery*. Cambridge University Press, 1976. 29

[O'H07] Peter O'Hearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1-3):271–307, 2007. 3

[OSS09] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *TPHOLs 2009: Theorem Proving in Higher Order Logics, LNCS 5674*, pages 391–407, 2009. 37

[Owe10] Scott Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In Theo D'Hondt, editor, *ECOOP 2010 — Object-Oriented Programming, 24th European Conference*, volume 6183 of *Lecture Notes in Computer Science*, pages 478–503. Springer, June 2010. 37

[PHA04] S. E. Paynter, Neil Henderson, and J. M. Armstrong. Ramifications of metastability in bit variables explored via Simpson's 4-slot mechanism. *Formal Aspects of Computing*, 16(4):332–351, 2004. 2

[Rus02] John Rushby. Model checking Simpson's four-slot fully asynchronous communication mechanism. Technical report, Computer Science Laboratory, SRI International, July 2002. 2

[Sim86] Hugo Simpson. The mascot method. *Software Engineering Journal*, 1(3):103–120, may 1986. 2

[Sim90] H.R. Simpson. Four-Slot Fully Asynchronous Communication Mechanism. *IEE Proceedings*, 137(1):17–30, January 1990. 2, 6, 11, 27

[Sim92] H.R. Simpson. Correctness analysis for class of asynchronous communication mechanisms. *IEE Proceedings*, 139(1):35–49, January 1992. 2

[Sim97a] H.R. Simpson. New algorithms for asynchronous communication. *IEE Proceedings - Computers and Digital Techniques*, 144:227–231, July 1997. 2, 11, 14, 15

[Sim97b] H.R. Simpson. Role Model Analysis of an Asynchronous Communication Mechanism. *IEE Proceedings - Computers and Digital Techniques*, 144(4):232–240, July 1997. 14

[SJ79] H. R. Simpson and K. Jackson. Process synchronisation in MASCOT. *The Computer Journal*, 22(4):332–345, 1979. 2

[SSA+11] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. In *Proc. PLDI*, 2011. 37

[SSO+10] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, July 2010. (Research Highlights). 37

[SSZN+09] S. Sarkar, P. Sewell, F. Zappa Nardelli, S. Owens, T. Ridge, T. Braibant, M. Myreen, and J. Alglave. The semantics of x86-CC multiprocessor machine code. In *Proc. POPL 2009*, January 2009. 37

[Vaf07] Viktor Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2007. 3, 28

[VP07] Viktor Vafeiadis and Matthew J. Parkinson. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR 2007 – Concurrency Theory*, volume 4037 of *LNCS*, pages 256–271, August 2007. 3

[Yan07] Hongseok Yang. Relational separation logic. *Theoretical Computer Science*, 375(1-3):308–334, May 2007. 17