Under consideration for publication in Formal Aspects of Computing

Inter-process Buffers in Separation Logic with Rely-Guarantee

Richard Bornat and Hasan Amjad
Middlesex University, London, UK

Abstract. Separation logic allows simple proofs of concurrent algorithms which use blocking mechanisms such as semaphores. It can even deal with non-blocking algorithms. With the addition of mechanisms borrowed from rely-guarantee, we can make reasonably simple proofs of some simple non-blocking algorithms. We show that it extends to proofs of some intricate algorithms, including Simpson's famous asynchronous four-slot buffer and Harris's novel three-slot algorithm, in a manner that is arguably simpler than earlier treatments, though we cannot claim that we have yet found proofs that are as simple as we would wish. Our example proofs show functional correctness but do not deal with questions of liveness.

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

1. Introduction

The message of structured programming was that we should not use programming constructs which we do not understand – that is, whose effect we cannot specify and cannot reason about. The programming community took this message to heart despite the fact that there was an execution cost to be paid when giving up assembly language and the `goto` command. The cost was worth it for the benefits gained: programmers understood their programs better and they made fewer mistakes. Structured programming won such a comprehensive victory that most programmers can hardly remember that we didn't always write sequential programs that way. But in concurrent programming the same battle has still to be fought and won.

There are concurrency constructs which we do understand, but we still find it very hard to reason about them completely formally. And, unfortunately, some of those which we least understand offer great efficiency advantages over those we understand better. In such a situation programmers will use the most efficient mechanisms because there is no reason to do anything else. It is imperative that we try to understand, to specify and to reason about those mechanisms. We are certainly not yet at a stage where we can build a compiler that could check our use of concurrency, but if we want to get there, it can only be helpful to understand how to make formal verifications of some of the concurrent algorithms that exist.

To understand concurrency properly we must begin at the beginning, and it is remarkable how close to the beginning we still are. Passing a value reliably between two concurrent processes using a shared variable

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

– a *buffer* – is a problem identified by Dijkstra [Dij65], who proposed semaphores as the basis of a solution. Despite the continuing influence of that seminal work, the problem remains.

The main difficulty is that we must transmit *coherent* values: that is, we must somehow prevent overlapping of writes and reads, so that a reader always receives a complete written value rather than a mixture of an old value and a partially-written new one. Dijkstra's solution was to use semaphores to provide *mutual exclusion* between read and write operations: a writer must wait if a reader is reading, and vice-versa. This solves the problem of coherence but it introduces a new problem: the underlying semaphore mechanism may not (and in general cannot) ensure that readers and writers get fair access to the buffer and the effect may be *starvation* of one process or the other. Worse, a faulty algorithm may provoke *deadlock*, in which each process is waiting for the other and no progress is ever made.

Deadlock and starvation are such problems that alternatives to semaphores are increasingly popular. The ideal, when synchronisation is not required, is *wait-free* communication [Her91] in which each communicating process can always make its move without waiting for the other. This can offer real efficiency advantages, but algorithms can be intricate and very difficult to understand. This paper examines two wait-free algorithms which provide an implementation of a buffer.

Shared-variable communication is peculiar. It does not offer synchronisation – a writer never knows whether a reader has taken a value, let alone when it happened if it did. The writer may write more frequently than the reader reads, so that some values transmitted are simply ignored – the communication can be *lossy* – and/or the reader may read more frequently than the writer writes, so that some values are read more than once – the communication can be *stuttering*. All the wait-free single-place buffer algorithms of which we are aware use more than one buffer 'slot' to produce their effect and this introduces a new problem: because those shared slots can contain more than one transmitted value, a reader might not always receive a *fresh* value, the one most recently written at the instant when it starts to read.

One wait-free single-place buffer algorithm is described by Simpson [Sim90]. It uses an array of four shared data-variable *slots* and two shared single-bit index variables to show where the latest written value is stored and to signal where the next should be placed. It manages without any special concurrency mechanisms and yet delivers coherence and freshness. Simpson was unable to devise an algorithm with fewer than four slots, but Tim Harris (Microsoft Research Cambridge) described to us in an email an algorithm with an array of two slots, a single index variable and an additional shared data variable. It is not as fast as Simpson's algorithm, because there must sometimes be a write to and a read from the additional variable as well as a buffer slot, but like Simpson's it is asynchronous and wait-free.

Both these algorithms are small but exasperatingly difficult to understand. Why four slots and not two, or three, for Simpson? How does Harris reliably detect non-collision? How do they manage without synchronisation mechanisms? Why their particular arrangement of index variables and slots? Simpson [Sim92] explains the evolution of his algorithm, why one or two slots necessarily require waiting (blocking), and why the author could not make a three-slot version. Despite that explanation, few understand his algorithm. Harris's algorithm, so far as we know, has no previous formal explanation.

In this paper we analyse a sequence of buffer algorithms ending with Harris's and Simpson's algorithms. We employ separation logic [ORY01, Rey02], its developments which deal with partial ownership of resource [BCOP05] and variables as resource [BCY05, PBC06], and Vafieadis and Parkinson's RGSep logic [VP07] which marries separation logic with Jones's rely-guarantee [Jon83]. So far as we are aware ours is the first demonstration of the use of RGSep on significant examples. We believe that the resulting proofs are simpler than earlier treatments, though not as simple as we had hoped.

It would be ideal if we were able to present a sequence of refinements that proceed from an abstract algorithm to Simpson's algorithm. But there is as yet no formal notion of refinement in separation logic, let alone refinement of atomicity, nor are all the algorithms related in that way.

2. Related work

In one sense the algorithms in this paper do not need further study because their state spaces are small enough for their correctness to be established by model checking. Rushby [Rus02] and indeed Simpson [Sim97b] give a correctness arguments of that kind. Harris's algorithm, too, could no doubt be model-checked. But these algorithms are the tip of an iceberg: there are plenty of concurrent mechanisms whose state-space is too large to model-check: see, for example, Michael's non-blocking stack [Mic04], dealt with formally by Parkinson et

al. [PBO07], and very many of the algorithms described by Herlihy and Shavits [HS08], which so far as we are aware have no formal treatment yet.

There have been formal treatments of Simpson's algorithm which did not use model checking, by Henderson and Paynter [HP02] and Henderson [Hen03]. That work demonstrates how the use of a particular framework simplifies the proof. Henderson's work demonstrates a rely-guarantee proof in CVS which verifies the properties of a state monad that corresponds to the algorithm. It is a mechanised formal proof, although it does not work directly on the code. In [PHA04] Paynter and Henderson discuss the difficulties of a formal treatment of Simpson's algorithm in its original hardware context, where metastability of single-bit 'variables' represented by bistables becomes an issue.

More recently Jones and Pierce [JP08] have presented a refinement development of Simpson's algorithm, providing an explanation of safety, correctness, and in particular freshness. Abrial (private communication) is preparing a treatment in event-B.

3. Separation logic and its extensions

Hoare logic [Hoa69] treats arrays as single variables. Each assignment to an array alters the whole array. This reduces problems of aliasing between array-element formulas like $a[i]$ and $a[j \div 2]$ to questions about arithmetic equality of subscripts i and $j \div 2$. But in turn it means that verifications can come to depend upon lots of intricate arithmetical reasoning dealing with aliasing.

The heap – the part of the store allocated and released by operations like Pascal's new(x)/dispose(x) or C's `z=malloc(n)`/`free(z)`– is, in principle, a large array indexed by integer addresses. But it is difficult to reason about programs which use the heap using the Hoare-logic treatment of arrays because every assignment potentially interferes with every access. That leads many to believe that pointers are a bad idea: Hoare himself opined [Hoa73] "Pointers are like jumps, leading wildly from one part of the data structure to another. Their introduction into high-level languages has been a step backward from which we may never recover."; SPARK [Bar03], a high-integrity programming environment with its roots in Hoare logic, avoids them entirely.

Burstall observed [Bur72] that lists which use separate areas of the heap can be reasoned about separately. Morris [Mor82] used the same insight to make a semi-formal proof of the Schorr-Waite garbage collector. Bornat [Bor00] proved various graph programs including Schorr-Waite using a formal notion of separation; Mehta and Nipkow [MN05] mechanised those proofs. But all laboured to show their results: as Burstall put it in the case of separated lists, it was "a great deal of work for such a simple matter".

The proofs we present in this paper all use separation logic in one of its variants. We present here a summary of the logics we use.

3.1. Separation logic

The aim of program logic development is to make difficult proofs straightforward. Reynolds [Rey00] generalised Burstall's notion to deal with heap data structures in general. O'Hearn, Reynolds and Yang [ORY01, Rey02] turned that work into separation logic by adapting O'Hearn and Pym's BI [OP99, Pym99].

Separation logic is a version of Hoare logic in which assertions describe not only the values of variables, but also the heap resources which are available for use. The triple $\{Q\}\ C\ \{R\}$ is interpreted in no-fault partial-correctness terms: command C, executed in a state and with heap described by Q, will not fault and, if it terminates, will do so in a state and with heap described by R. Absence of fault includes a guarantee that execution will not exceed the heap footprint described by Q, for example by attempting to access cells outside that footprint. Command C may acquire and/or release heap by using $x :=$ new( ) and dispose(x).

The separation-logic heap is indexed by natural numbers and contains integers plus the distinguished non-indexing value nil. Pre and postconditions use the constants, connectives and quantifiers of classical predicate calculus, plus the following predicates:

- **emp** – true of the empty heap;
- $E \mapsto E'$ – true of a singleton heap with address E and contents E';
- $E \mapsto \_$ – shorthand for $\exists k(E \mapsto k)$;
- $E \mapsto F_0, F_1, \ldots, F_{n-1}$ – shorthand for $E \mapsto F_0 \star E+1 \mapsto F_1 \star \cdots \star E+n-1 \mapsto F_{n-1}$

- $A \star B$ – true of a heap which can be separated into two parts, one described by A and the other by B;
- $A \wedge B$ – true of a heap which is described by A and at the same time by B;
- $A \mathrel{-\!\!\star} B$ – true of a heap which, if we added a separate portion satisfying A, would satisfy B.

The most important feature of separation logic is that $\star$ describes separation of resource. $E \mapsto E' \star F \mapsto F'$, for example, describes two separate single-cell heaps made into a two-cell heap: it is impossible that E and F could be the same address, so if E and F are equal, then $E \mapsto E' \star F \mapsto F'$ is false. By contrast in $E \mapsto E' \wedge F \mapsto F'$, because each conjunct describes the same heap, we know that $E = F$ and $E' = F'$. Conjunctions such as $y = k \wedge x \mapsto k$, in which only one side explicitly describes a heap, are equivalent to $(y = k \star \text{true}) \wedge (x \mapsto k)$ in which the heap on both sides is explicit.

Note that address arithmetic is built in to the logic in the treatment of $E \mapsto F_0, F_1, \ldots, F_{n-1}$.

E, E', F etc. in these definitions are 'pure expressions' which can only mention variables and constants. Heap-accessing formulae $[E]$ can appear in commands but not in assertions. Instead of $y = [x]$, for example, we must assert $\exists k(y = k \wedge x \mapsto k)$.

The logic uses only a limited range of assignment instructions. More complex accesses and mutations can be compiled into sequences of these few, and assignments are the only way to access or mutate the heap.

- $x := E$ – assignment to a variable, with the Hoare axiom $\{R[E/x]\}\ x := E\ \{R\}$.
- $[E] := E'$ – assignment to a heap cell, with axiom $\{E \mapsto \_\}\ [E] := E'\ \{E \mapsto E'\}$.
- $x := [E]$ – access to a heap cell, with axiom $\{E \mapsto E'\}\ x := [E]\ \{E \mapsto E' \wedge x = E'\}$ and side-condition that x cannot occur free in E or E'.

Because the heap assignment and access axioms work forward, in practice we usually employ a forward variable-assignment axiom $\{Q \wedge x = X\}\ x := E\ \{Q[X/x] \wedge x = E[X/x]\}$. We can use a similar axiom with the heap-access assignment if E or E' mention x.

The *frame rule* is what drives separation logic, allowing us to focus on the minimum heap that a command needs to do its work, and guaranteeing that all the rest of the heap remains unchanged:

$$\frac{\{Q\}\ C\ \{R\}}{\{P \star Q\}\ C\ \{P \star R\}}\ (\mathit{modifies}(C)\ \cap \mathit{vars}(P) = \emptyset)\ \text{frame rule} \tag{1}$$

If P is separate from Q, and C transforms Q into R then surely when C finishes we have R and — separately and therefore untouched — still P, provided that C does not modify any of the program variables free in P. The rule is a consequence of the *frame property* of the programming language: informally, if a command C is safe (does not fault) in a given heap H, then in a larger heap $H \star H'$ (a) it will not fault and (b) the execution will correspond to *some* execution in the original heap [YO02].

There are axiomatisations of new and dispose which we include for completeness, though they are not used in the examples in this paper:

- $\{\mathbf{emp}\}\ x := \mathit{new}()\ \{x \mapsto \_\}$
- $\{E \mapsto \_\}\ \mathit{dispose}(E)\ \{\mathbf{emp}\}$

– new() acquires a cell from a memory allocator and dispose(_) returns a cell. Because of the frame rule these axioms can focus on the minimum resource required to execute the instruction: nothing for new, a single cell for dispose. The address returned by new is, in principle, unpredictable – i.e. new is non-deterministic. The pointer used to return a cell by dispose still points to the cell after disposal, but the cell itself is no longer usable; a subsequent call of new might return the disposed cell.

We can read $E \mapsto F$ as expressing *ownership* of a heap cell: it is owned by the part of the program being analysed and not by the memory allocator. This idea becomes important and useful once we consider concurrent programs.

3.2. Variables as resource

In separation logic variables are separate from the heap. They model a stack or the registers of a machine, and they cannot be pointed to. But if we treat variables as resource in the same sort of way as we deal with the heap as resource, then we can achieve two goals: we can eliminate variable-use side conditions from our rules; and we can deal with algorithms, like those put forward by Dijkstra when he introduced

semaphores [Dij65], which control access to variables. The treatment of variables as resource is proposed in [BCY05] and shown to be sound in [PBC06].

A predicate $\mathrm{Own}(x)$ can express ownership of the variable named x, in the same way as $E \mapsto \_$ expresses ownership of the heap cell addressed by E, but it is peculiar in an assertion because it is a predicate of the variable itself and not its value. Ownership is not affected by assignment, so when substituting a value-formula for x, as for example in Hoare's assignment axiom, we do not change x in $\mathrm{Own}(x)$. Therefore for clarity, and also for brevity, we separate variable ownership assertions from value assertions using a context notation with a models-turnstile $\vDash$. So $b, \boldsymbol{ws}, \boldsymbol{rs} \vDash P$, for example, asserts $\mathrm{Own}(b) \star \mathrm{Own}(\boldsymbol{ws}) \star \mathrm{Own}(\boldsymbol{rs}) \wedge P$.

Mixing variable and heap resource is subtle, and it is necessary to explain the use of $\wedge P$ rather than $\star P$ in the de-abbreviation above. In general $\mathrm{Own}(x)$ asserts a singleton 'stack' of variables but says nothing about the heap; $E \to F$ asserts a singleton heap but nothing about the stack. $\mathrm{Own}(x) \wedge x \mapsto 3$ says that we own x and that x points to a heap cell containing 3, and that is all the resource we have. When we want to talk about absence of resource, we have to split the **emp** of separation logic into $\mathbf{emp_s}$ – we have no variables – and $\mathbf{emp_h}$ – we have no heap; $\mathbf{emp_s} \wedge \mathbf{emp_h} \iff \mathbf{emp}$. (For simplicity, we shall elide the distinction between $\mathbf{emp_s}$, $\mathbf{emp_h}$ and **emp**, where we can, in our examples below.)

For legibility we normally construct our assertions so that occurrences of variables are scoped by the relevant ownership-contexts. When we cannot do this it is untidy but not a problem, because the context notation is merely a shorthand for use of the Own predicates.

Once we account for variable ownership, we can make sure that our programs use variables-as-resource as carefully as they use heap-cells-as-resource. Just as the assignment $[E] := F$ requires a precondition that asserts ownership of the cell at address E, so the assignment $x := E$ requires a precondition that asserts ownership of x. We also need to assert ownership (or permission to read – see §3.4 below) to any variables mentioned in E or in F. This enables us to remove the side-conditions on variable use that disfigures the frame rule and normally disfigures rules used to reason about concurrency. [BCY05] uses the notion to give formal proofs of some classic semaphore algorithms.

We have not shown the details of our reasoning about variables as resource in this paper, but variable-ownership transfer is at the heart of the first few of the proofs we discuss.

3.3. Concurrent separation logic

Concurrent separation logic [O'H04, O'H07, Bro07] deals with shared-variable concurrency using a mechanism based on Hoare's conditional critical regions [Hoa72].

The disjoint concurrency rule states that we can combine proofs of separate threads provided that each operates on its own separate heap.

$$\frac{\{Q_1\}\ C_1\ \{R_1\} \quad \cdots \quad \{Q_n\}\ C_n\ \{R_n\}}{\{Q_1 \star \cdots \star Q_n\}\ (C_1 \parallel \cdots \parallel C_n)\ \{R_1 \star \cdots \star R_n\}}\ \text{disjoint concurrency} \tag{2}$$

– note there is no side condition on variable use because ownership of variables is separated in $Q_1 \star \cdots \star Q_n$ and $R_1 \star \cdots \star R_n$ as sketched in §3.2 above.

As well as the heap owned by each thread there are separate heaps each associated with a *resource identifier* and described by an invariant. The conditional critical region (henceforward CCR) command $\mathsf{with}\ r\ \mathsf{when}\ G\ \mathsf{do}\ C\ \mathsf{od}$ gives mutually exclusive access to the heap (and variables) described by the invariant I_r. Execution of the command goes as follows:

1. the executing thread attempts to gain exclusive ownership of the resource identifier r. If this fails, because some other thread already has ownership, the thread simply tries again. If it succeeds, then
2. the guard G is evaluated; G may safely mention the resources (variables, heap) owned by r, because it will only be evaluated when the thread has exclusive access to those resources. If G is false then the thread releases ownership of r and starts executing the command again. If G is true, then
3. the thread executes C which, like G, may refer to both the thread's and the resource identifier's resources. When execution of C terminates, ownership of r is released and the CCR execution terminates.

The concurrent separation logic CCR rule is

$$\frac{\{(Q \star I_r) \wedge G\}\ C\ \{R \star I_r\}}{\{Q\}\ \mathsf{with}\ r\ \mathsf{when}\ G\ \mathsf{do}\ C\ \mathsf{od}\ \{R\}}\ [Q \star I_r, I_r \star R \text{ precise}]\ \text{CCR} \tag{3}$$

(note that because the consequent triple describes partial correctness, the rule does not guarantee that execution will ever terminate). Use of the rule is sound only if $Q \star I_r$ and $I_r \star R$ are *precise* resource splits, i.e. if there is only one way to divide the combined resource so as to satisfy both formulae. In practice that requires that one or other of the two formulae – Q and I_r in one case, I_r and R in the other – should pick out a unique subheap from any heap which satisfies the conjunction. If we treat variables as resource there is no need for side conditions on variable use.

The invention of CCRs postdated the invention of semaphores [Dij65]. They are more difficult to implement than semaphores, but easier to reason about, and were the first concurrency mechanism treated in separation logic [O'H07].

3.4. Permissions

The $\mapsto$ and Own operators can be read as expressions of ownership: because of the frame property we can be sure, for example, that if one thread in a program asserts $E \mapsto \_$ then no other thread can concurrently assert anything about the heap cell addressed by E. O'Hearn made this reading the basis of the first treatment of a single-place buffer in [O'H04, O'H07] by showing an example in which ownership of a heap cell is transmitted between concurrent threads, one allocating the cell with new( ), the other safely recycling it with dispose(_).

Separation is only part of the concurrency problem: another part is sharing. The readers and writers algorithm [CHP71], for example, allows many readers access to a shared variable whilst giving writers exclusive access. A formal treatment of that algorithm is sketched in [BCOP05], using the notion of partial permissions and showing it in two forms: fractional permission and counting permission. In this paper we use fractional permissions.

Total ownership of a heap cell or a variable permits reading and writing. Fractional ownership allows sharing and permits reading only. Total ownership of a heap cell is written $\underset{1}{\longmapsto}$ (equivalent to $\mapsto$), fractional ownership as, for example, $\underset{0.3}{\longmapsto}$. Total ownership allows reading and writing; partial ownership allows reading only. Permissions can always be split, and can be combined provided that the total is not greater than 1.

$$E \underset{z}{\longmapsto} E' \star E \underset{z'}{\longmapsto} E' \iff E \underset{z+z'}{\longmapsto} E' \wedge z > 0 \wedge z' > 0 \tag{4}$$

There is no zero permission, and it is an oddity that the magnitude of permissions less than 1 is irrelevant except when accounting for splitting and recombination. Because permissions can always be split and usually combined there is some infelicity in dealing with inductive definitions of data structures, but the idea is sound, as shown in [Bro04, Bro07]. There is an Ownf for fractional permission on variables, and we write $x_{0.5}$, for example, to describe half-permission to use x.

The notion of counting permissions, necessary to explain the readers and writers algorithm, is not used in this paper.

3.5. RGSep

The CCR machinery of concurrent separation logic can be pressed into service to deal with programs that communicate using atomic instructions and a single shared memory. We treat the atomic command $\langle C \rangle$ as if it were the CCR command $\mathsf{with}\ s\ \mathsf{when}\ \mathrm{true}\ \mathsf{do}\ C\ \mathsf{od}$, where s is the otherwise-unnamed resource identifier which owns all the shared store. Then we can reason for each atomic command

$$\frac{\{Q \star I_s\}\ C\ \{I_s \star R\}}{\{Q\}\ \langle C \rangle\ \{R\}}\ [Q \star I_s, I_s \star R \text{ precise}]\ \text{atomic CSL} \tag{5}$$

But this treatment requires a single invariant for all the shared store, preserved by every atomic instruction, and in practice that can make for inelegant proofs. In particular, we are often required to introduce auxiliary variables [Owi75, OG76a, OG76b] to express variations in the invariant at different program points.

Jones's rely/guarantee formalism [Jon83] allows reasoning about concurrent threads which may interfere with each others' variables in atomic instructions, but it requires non-local reasoning about every thread at the same time. As a consequence, to add a thread to a verified collection it is necessary to investigate anew the activity of each other thread. But in place of a global invariant, it allows for invariant formulae in the form of *stable assertions* as pre and postconditions, assertions which are invariant under the action of other threads.

Vafeiadis and Parkinson [VP06, Vaf07] have married rely-guarantee and concurrent separation logic to simplify that problem. We can express *temporary* invariants about shared storage in the form of *stable assertions* in their logic RGSep – assertions which other threads will not disturb. As a result we can localise some of our invariants to program position and avoid some uses of auxiliary variables.

In RGSep, as in concurrent separation logic, shared store is separated from local store. In RGSep assertions are divided accordingly. As in concurrent separation logic, atomic commands are the only way to access shared store. We write $\langle C \rangle$ to indicate atomic execution of a command. Then the triple

$$\left\{ P \star \boxed{sP} \right\} \ \langle C \rangle \ \left\{ Q \star \boxed{sQ} \right\} \tag{6}$$

declares that $\langle C \rangle$ transforms local resource P and shared resource sP into local resource Q and shared resource sQ and thereby generates an *action* – an effect on all or part of the shared store – $sP \rightsquigarrow sQ$. The actions of a thread – rather, their transitive and reflexive closure – are gathered together into its *guarantee* G, the maximum interference it can cause for other threads. The guarantees of other threads are gathered into the *rely* R, the maximum interference that the environment can generate for this thread.

As in rely-guarantee the Hoare triple $\{P\}\ C\ \{Q\}$ becomes the quintuple $C\,\mathbf{sat}(R, G, P, Q)$. The concurrent execution rule is

$$\frac{C_1\,\mathbf{sat}(R \cup G_2 \cup \cdots \cup G_n, P_1, Q_1) \quad \cdots \quad C_n\,\mathbf{sat}(R \cup G_1 \cup \cdots \cup G_{n-1}, P_n, Q_n)}{(C_1 \parallel \cdots \parallel C_n)\,\mathbf{sat}(R, G_1 \cup \cdots \cup G_n, P_1 \star \cdots \star P_n, Q_1 \star \cdots \star Q_n)}\ \text{RGSep concurrency} \tag{7}$$

– each thread must accept the global interference R of the parallel composition plus the interference generated by its partner threads; the total guarantee is the union of interference of all the threads.

For simplicity of presentation of proofs, since we are dealing with algorithms in which there are only two threads and a single parallel composition, we show triples and keep R and G implicit: the writer's R is the reader's G and vice-versa.

In dealing with atomic commands RGSep replaces the notion of a global shared-resource invariant with the notion of *stable assertions*. A stable assertion is one that is invariant under the action of the environment: this notion is defined mathematically in [VP06], but we omit the details here. When considering the triple in (6), there are four possibilities:

total stability both sP and sQ are stable under every possible action of the environment;
early stability sQ is stable but sP can describe the instantaneous state just before $\langle C \rangle$ is executed;
late stability sP is stable but sQ describes the instantaneous state just after $\langle C \rangle$ is executed;
mid stability neither sP nor sQ need be stable: both can describe instantaneous states.

The choice between these interpretations is a matter of convenience of presentation. If we consider a single isolated atomic instruction, then we can reasonably talk about its instantaneous effect (mid stability). If we want the shared-store postcondition sQ of one atomic instruction to be the precondition of another, then necessarily it must be stable (total stability or early stability) or made stable by weakening. In this paper we use late stability and weaken shared-store assertions that are not the postcondition of the whole proof.

We simplify the rules of RGSep in our presentation, in particular by making R and G implicit. The following is our rendering of the late-stability atomic instruction rule.

$$\frac{\exists \vec{x}(sP \star sF) \text{ stable under } R \quad \{P \star sP\}\ C\ \{Q \star sQ\} \quad sP \rightsquigarrow sQ \subseteq G}{\left\{ P \star \boxed{\exists \vec{x}(sP \star sF)} \right\}\ \langle C \rangle\ \left\{ \exists \vec{x} \left(Q \star \boxed{sQ \star sF} \right) \right\}}\ [sP, sQ \text{ precise}] \text{ atomic RGSep} \tag{8}$$

The action $sP \rightsquigarrow sQ$ need only describe the part of shared store essential to the action, but typically we add sF to make it easier to show that $sP \star sF$ is stable.

Because local assertions can only mention local resource, and shared-store (boxed) assertions can only mention shared resource, we can use existentially-quantified variables to allow the local store in the post-condition to contain values retrieved from the shared store by $\langle C \rangle$.

Reasoning about atomic commands is a little more subtle than we have made it seem, and properly uses a layering of deductions: stability outside, framing and action inside. We use that layering when reasoning forward: given are C, P and $sP \star sF$; we require that $sP \star sF$ is stable; then we deal with $\{P \star sP\}\ C\ \{Q \star sQ\}$; finally we check that $sP \rightsquigarrow sQ \subseteq G$. That is, we deal with the effect of C after we have shown that the precondition is stable.

$$\begin{array}{l}\mathsf{local}\, b = \mathit{null}\, \mathsf{in} \\ \left(\begin{array}{l|l} \ldots & \ldots \\ \mathit{write}(w) = \langle b := w \rangle & \mathit{read}(\,) = \mathsf{local}\, y\, \mathsf{in} \\ & \qquad\qquad \langle y := b \rangle; \\ & \qquad\qquad \mathsf{return}\, y \end{array} \right)\end{array}$$

Fig. 1. An idealised single-slot buffer, with angle-bracketed atomic instructions

4. Single-slot buffers

A shared variable is a single-*place* buffer, which can store exactly one value en route between writer and reader. It is also, following Simpson's terminology, a single-*slot* buffer, because it uses only one shared 'slot' variable to implement that single place. Our examples in this paper will all be single-place buffers, but they will not all be single-slot.

Fig. 1 shows an idealised single-slot algorithm. We suppose that the instructions writing to and reading from the buffer are atomic, which property we indicate by angle-bracketing. Ideally atomic instructions would execute instantaneously, and we can imagine that in this case they do. In this and all other examples we are interested in the *write*(w) and *read*$(\,)$ procedures by which the processes communicate, and we concern ourselves only with a single writer and a single reader.

Everything that a single-place buffer does must correspond to something that fig. 1 can do. Successive calls of *write* transmit a sequence of values by assigning them sequentially to b. The reader might miss some of the values if it does not always read quickly enough (the communication can be *lossy*); it might read the same value more than once if the writer does not overwrite quickly enough (the communication can be *stuttering*); and it might read the buffer before the first *write* (in which case it will see the distinguished value *null*). But every value it sees will be one that the writer put there (if we include the initial value *null*), so the values seen by the reader will be a *stuttering subsequence* of those written – i.e. every value in the sequence seen is in the sequence written, and in the order written, but there may be repetitions.

In fig. 1 the value in b is always the latest value written, so the value the reader retrieves is always the latest value written. In more complicated algorithms where there is more than one slot, the latest value written will be in only one of the slots, and the reader might not in every case retrieve the very latest value. It is necessary to be clear about what is acceptable. Simpson's notion of *freshness* is that when *read*$(\,)$ is called there is a latest completely written value – say B – and the value returned by that call must not be earlier than B in the written sequence (though it could be later).

Stuttering subsequencing and freshness are not enough: we need also to allow either thread to proceed without waiting for the other. In the ideal case of fig. 1 this is not a problem because we imagine that executions of atomic instructions are instantaneous. But in any realistic implementation executions will take time and will have to be forced to interleave, so that one thread may sometimes have to wait for the other to finish its business before it can start its own. Such waiting can be unfairly organised, and in the limit one thread or the other may never get a turn. Unfairness is not prohibited by the stuttering-subsequence and freshness properties: a permanently locked-out reader 'sees' the empty sequence, which is a subsequence of the values written, and a permanently locked-out writer 'produces' a singleton sequence which the reader stutters over forever. We need somehow to be sure that such starvation cannot happen.

Issues of fairness, liveness and starvation are beyond the reach of separation logic at present. We shall make informal arguments and appeal to programmer's intuition. Simpson's and Harris's algorithm, for example, avoid starvation by abolishing waiting, and thus avoid the need for any interleaving-supervision mechanism, either upfront or behind the scenes.

In the rest of this section we discuss various algorithms which are related to the algorithm of fig. 1 and use them as a means of introducing various logical techniques which we call upon later. It will be seen that as our implementations become more realistic, our proofs become more intricate.

4.1. A one-slot buffer using a conditional critical region

As an algorithm fig. 1 is unrealistic without an implementation of atomicity. It is not necessary to provide instantaneous execution for the algorithm to work: it is enough to force atomic actions to be interleaved. One way of doing that – historically, the first way to be proposed – is to use a synchronisation mechanism to

$$
\begin{array}{l}
\mathsf{local}\, b = \mathit{null}, \boldsymbol{ws} = \boldsymbol{null.}, \boldsymbol{rs} = \boldsymbol{.}\ \mathsf{in} \\
\mathsf{resource}\ \mathit{bufr} \\
\quad \mathsf{invariant}\, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = b\, \mathsf{in} \\
\left(\begin{array}{l|l}
\ldots & \ldots \\
\mathit{write}(w) = & \mathit{read}(\,) = \\
\quad \mathsf{with}\ \mathit{bufr}\ \mathsf{when}\ \mathrm{true}\ \mathsf{do} & \quad \mathsf{local}\ y; \\
\quad\quad b := w; \boldsymbol{ws} := \boldsymbol{ws.b} & \quad \mathsf{with}\ \mathit{bufr}\ \mathsf{when}\ \mathrm{true}\ \mathsf{do} \\
\quad \mathsf{od} & \quad\quad y := b; \boldsymbol{rs} := \boldsymbol{rs.b} \\
 & \quad \mathsf{od}; \\
 & \quad \mathsf{return}\ y
\end{array} \right)
\end{array}
$$

Fig. 2. A one-slot buffer without turn-taking, with darkened auxiliary annotations

$$
\begin{array}{l}
\{w, \boldsymbol{ws}_{0.5} \vDash w = W \wedge \boldsymbol{ws} = WS\} \\
\quad \mathsf{with}\ \mathit{bufr}\ \mathsf{when}\ \mathrm{true}\ \mathsf{do} \\
\quad\quad \{(w, \boldsymbol{ws}_{0.5} \vDash w = W \wedge \boldsymbol{ws} = WS) \star (b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = b)\} \ \Rightarrow \\
\quad\quad \{w, b, \boldsymbol{ws}, \boldsymbol{rs}_{0.5} \vDash w = W \wedge \boldsymbol{ws} = WS \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = b\} \\
\quad\quad\quad b := w \\
\quad\quad \{w, b, \boldsymbol{ws}, \boldsymbol{rs}_{0.5} \vDash b = w = W \wedge \boldsymbol{ws} = WS \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \exists b' (\boldsymbol{ws}_\Omega = b')\} \\
\quad\quad\quad \boldsymbol{ws} := \boldsymbol{ws.b} \\
\quad\quad \{w, b, \boldsymbol{ws}, \boldsymbol{rs}_{0.5} \vDash b = w = W \wedge \boldsymbol{ws} = WS.b \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq WS \wedge (WS.b)_\Omega = b\} \ \Rightarrow \\
\quad\quad \{(w, \boldsymbol{ws}_{0.5} \vDash w = W \wedge \boldsymbol{ws} = WS.W) \star (b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = b)\} \\
\quad \mathsf{od} \\
\{w, \boldsymbol{ws}_{0.5} \vDash w = W \wedge \boldsymbol{ws} = WS.W\}
\end{array}
$$

Fig. 3. Verification of the one-slot non-turn-taking CCR writer

provide mutual exclusion of competing concurrent executions. Our implementation, in fig. 2, uses conditional critical regions and we verify it using concurrent separation logic.

In fig. 2 we have added auxiliary variables $\boldsymbol{ws}$ and $\boldsymbol{rs}$ to keep track of the sequences written and read; because angle brackets are used for atomic instructions elsewhere, '$\mathit{null}.$' is the singleton sequence containing null, '$.$' is the empty sequence, and '$s.x$' appends the singleton x to the sequence s. The shared resource identifier is bufr, and its invariant states

- which variables it owns or partially owns: $b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \ldots$;
- that $\boldsymbol{rs}$ destuttered is a subsequence of $\boldsymbol{ws}$: $\lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws}$;
- that b contains the last element of $\boldsymbol{ws}$: $\boldsymbol{ws}_\Omega = b$.

The buffer only gets half-ownership of $\boldsymbol{ws}$ and $\boldsymbol{rs}$: half of $\boldsymbol{ws}$ goes to the writer, and half of $\boldsymbol{rs}$ to the reader, so that we can assert in pre and postconditions properties of the sequences they construct.

In fig. 2 there are no atomic actions. Instead we rely on mutual exclusion of execution of the bodies of the two CCRs. We can reason about the body of $\mathit{write}(w)$ as in fig. 3. After $\boldsymbol{ws} := \boldsymbol{ws.b}$ we still have $\boldsymbol{ws}_\Omega = (WS.b)_\Omega = b$, and since $\lfloor \boldsymbol{rs} \rfloor \preccurlyeq WS$ we have $\lfloor \boldsymbol{rs} \rfloor \preccurlyeq WS.b$. Notice how half-ownership of $\boldsymbol{ws}$ outside the resource identifier allows us to write a meaningful pre and postcondition for the CCR; notice also how the two half-ownerships combine, using the CCR rule, to give total ownership inside the body of the CCR, permitting the assignment to $\boldsymbol{ws}$.

The body of $\mathit{read}(\,)$ is hardly more difficult, and the proof is shown in fig. 4. After $\boldsymbol{rs} := \boldsymbol{rs.b}$, if $\lfloor RS \rfloor \preccurlyeq \boldsymbol{ws}$ then either RS ends with b, in which case $\lfloor RS \rfloor = \lfloor RS.b \rfloor$ and therefore $\lfloor RS.b \rfloor \preccurlyeq \boldsymbol{ws}$, or it does not end with b but $\boldsymbol{ws}$ does ($\boldsymbol{ws}_\Omega = b$), in which case also $\lfloor RS.b \rfloor \preccurlyeq \boldsymbol{ws}$. Notice, once again, the use of half-ownership to allow meaningful pre and postcondition assertions around the CCR.

All of this may look like a deal of effort for little return, proving the obvious with a special formalism (the informal reasoning about stuttering and subsequence could be formalised, though we have not done so here). But what seems obvious may not be what is true, especially in concurrency, and taking great care with resource provides an explanation of what is happening when you use a CCR that is, in principle, machine-checkable.

$\{y, \boldsymbol{rs}_{0.5} \vDash \boldsymbol{rs} = RS\}$
with *bufr* when true do
$\quad \{(y, \boldsymbol{rs}_{0.5} \vDash \boldsymbol{rs} = RS) \star (b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = b)\} \Rightarrow$
$\quad \{y, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs} \vDash \boldsymbol{rs} = RS \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = b\}$
$\quad\quad y := b$
$\quad \{y, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs} \vDash y = b \wedge \boldsymbol{rs} = RS \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = b\}$
$\quad\quad \boldsymbol{rs} := \boldsymbol{rs}.\boldsymbol{b}$
$\quad \{y, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs} \vDash y = b \wedge \boldsymbol{rs} = RS.b \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = b\} \Rightarrow$
$\quad \{(y, \boldsymbol{rs}_{0.5} \vDash \boldsymbol{rs} = RS.y) \star (b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = b)\}$
od
$\{y, \boldsymbol{rs}_{0.5} \vDash \boldsymbol{rs} = RS.y\}$

Fig. 4. Verification of the one-slot non-turn-taking CCR reader

local $full = \text{true}, b = null, \boldsymbol{ws} = null., \boldsymbol{rs} = .$ in
resource *bufr*
invariant $full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash (\neg full \wedge \boldsymbol{ws} = \boldsymbol{rs}) \vee (full \wedge \boldsymbol{ws} = \boldsymbol{rs}.b)$ in

...	‖	...
write(w) =		*read*( ) =
with *bufr* when $\neg full$ do		local y;
$b := w$; $\boldsymbol{ws} := \boldsymbol{ws}.\boldsymbol{b}$;		with *bufr* when *full* do
$full := \text{true}$		$y := b$; $\boldsymbol{rs} := \boldsymbol{rs}.\boldsymbol{b}$;
od		$full := \text{false}$
		od;
		return y

Fig. 5. A one-slot turn-taking buffer using a CCR, adapted from [O'H07]

Note that this algorithm does not require turn-taking – that is, it allows lossy and stuttering communication – but that it does use an unwieldy and potentially unfair synchronisation mechanism.

4.2. A one-slot turn-taking buffer using a CCR

The algorithm of fig. 2 does not guarantee that either reader or writer will ever manage to access the shared resource. We can guarantee each of them some progress if we force them to take turns. In fig. 5 the resource identifier *bufr* once again owns the buffer variable b as well as half the auxiliary accounting variables $\boldsymbol{ws}$ and $\boldsymbol{rs}$; in addition it owns a Boolean variable *full*. The invariant is much simpler than before: either $\boldsymbol{rs}$ and $\boldsymbol{ws}$ are in step (when $\neg full$), or $\boldsymbol{ws}$ is one step ahead (when *full*), and in the *full* case the extra value is also in b. The program starts with a full buffer, to correspond with other examples.

O'Hearn [O'H04] used this buffer mechanism to transfer ownership of a heap-cell from writer to reader. That was a compelling example of the power of the logic, but use of the heap is unnecessary to our argument. Using reasoning very similar to fig. 3 and fig. 4 it is straightforward to show that the invariant of *bufr* is maintained.

Verification of the body of the writer is shown in fig. 6. Coalescing the writer's resource, the invariant and the guard $\neg full$ resolves the disjunction in the invariant; reasoning is then straightforward until the disjunction has to be put back at the end of the CCR body.

Reasoning about the reader in fig. 7 is similar. Compared to fig. 3 and fig. 4 the reasoning about $\boldsymbol{ws}$ and $\boldsymbol{rs}$ is tame: all it needs is a bit of equality substitution. That is fine because this is a far tamer algorithm. It does not stutter because of turn taking, so $\boldsymbol{rs} \preccurlyeq \boldsymbol{ws}$, it does have freshness because $\boldsymbol{ws}_\Omega = b$, and it is fair – the threads take turns, so if one starves they both do, and it would be a very poor implementation of CCRs that provided that sort of unfairness. But it is unnaturally tame: we do not want the writer to have to wait for the reader to pick up a previous value, and we do not want the reader to block if there is no new value available. And it still uses an unwieldy synchronisation mechanism.

$\{w, \boldsymbol{ws}_{0.5} \vDash w = W \wedge \boldsymbol{ws} = WS\}$
with $bufr$ when $\neg full$ do
$\left\{ \left(\begin{array}{l} (w, \boldsymbol{ws}_{0.5} \vDash w = W \wedge \boldsymbol{ws} = WS) \star \\ \quad (full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash (\neg full \wedge \boldsymbol{ws} = \boldsymbol{rs}) \vee (full \wedge \boldsymbol{ws} = \boldsymbol{rs}.b)) \end{array} \right) \wedge \neg full \right\} \Rightarrow$
$\{w, full, b, \boldsymbol{ws}, \boldsymbol{rs}_{0.5} \vDash w = W \wedge \boldsymbol{ws} = WS \wedge \neg full \wedge \boldsymbol{ws} = \boldsymbol{rs}\}$
$b := w; \boldsymbol{ws} := \boldsymbol{ws}.\boldsymbol{b};\ full := \text{true}$
$\{w, full, b, \boldsymbol{ws}, \boldsymbol{rs}_{0.5} \vDash w = b = W \wedge \boldsymbol{ws} = WS.b \wedge full \wedge WS = \boldsymbol{rs}\} \Rightarrow$
$\{(w, \boldsymbol{ws}_{0.5} \vDash w = W \wedge \boldsymbol{ws} = WS.W) \star (full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash full \wedge \boldsymbol{ws} = \boldsymbol{rs}.b)\} \Rightarrow$
$\left\{ \begin{array}{l} (w, \boldsymbol{ws}_{0.5} \vDash w = W \wedge \boldsymbol{ws} = WS.W) \star \\ \quad (full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash (\neg full \wedge \boldsymbol{ws} = \boldsymbol{rs}) \vee (full \wedge \boldsymbol{ws} = \boldsymbol{rs}.b)) \end{array} \right\}$
od
$\{w, \boldsymbol{ws}_{0.5} \vDash w = W \wedge \boldsymbol{ws} = WS.W\}$

Fig. 6. Verification of the one-slot turn-taking CCR writer

$\{y, \boldsymbol{rs}_{0.5} \vDash \boldsymbol{rs} = RS\}$
with $bufr$ when $full$ do
$\left\{ \left(\begin{array}{l} (y, \boldsymbol{rs}_{0.5} \vDash \boldsymbol{rs} = RS) \star \\ \quad (full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash (\neg full \wedge \boldsymbol{ws} = \boldsymbol{rs}) \vee (full \wedge \boldsymbol{ws} = \boldsymbol{rs}.b)) \end{array} \right) \wedge full \right\} \Rightarrow$
$\{y, full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs} \vDash \boldsymbol{rs} = RS \wedge full \wedge \boldsymbol{ws} = \boldsymbol{rs}.b\}$
$y := b; \boldsymbol{rs} := \boldsymbol{rs}.\boldsymbol{b};\ full := \text{false}$
$\{y, full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs} \vDash y = b \wedge \boldsymbol{rs} = RS.b \wedge \neg full \wedge \boldsymbol{ws} = RS.b\} \Rightarrow$
$\{(y, \boldsymbol{rs}_{0.5} \vDash \boldsymbol{rs} = RS.y) \star (full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \neg full \wedge \boldsymbol{ws} = \boldsymbol{rs})\} \Rightarrow$
$\left\{ \begin{array}{l} (y, \boldsymbol{rs}_{0.5} \vDash \boldsymbol{rs} = RS.y) \star \\ \quad (full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash (\neg full \wedge \boldsymbol{ws} = \boldsymbol{rs}) \vee (full \wedge \boldsymbol{ws} = \boldsymbol{rs}.b)) \end{array} \right\}$
od
$\{y, \boldsymbol{rs}_{0.5} \vDash \boldsymbol{rs} = RS.y\}$

Fig. 7. Verification of the one-slot turn-taking CCR reader

4.3. A one-slot buffer using a split binary semaphore

CCRs make for simple reasoning, but they are not in practice much used. Semaphores [Dij65] are more popular; they and their cousins 'locks' and 'spinlocks' are used widely. Algorithms which use semaphores are subtly different from those which use CCRs, because all that is guaranteed mutually-exclusive is the actual semaphore execution, the P and V instructions themselves, not the 'critical sections' which they guard. That makes the proof argument more subtle, as we shall see.

local $b = null, \boldsymbol{ws} = null., \boldsymbol{rs} = .$ in
semaphore $full = 1$
  invariant $(full \vDash full = 0) \vee (full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash full = 1 \wedge \boldsymbol{ws} = \boldsymbol{rs}.b)$ in
semaphore $empty = 0$
  invariant $(empty \vDash empty = 0) \vee (empty, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash empty = 1 \wedge \boldsymbol{ws} = \boldsymbol{rs})$ in

...	...
$write(w) =$	$read(\,) =$
P($empty$);	local y in
$b := w; \boldsymbol{ws} := \boldsymbol{ws}.\boldsymbol{b};$	P($full$);
V($full$)	$y := b; \boldsymbol{rs} := \boldsymbol{rs}.\boldsymbol{b};$
	V($empty$);
	return y

Fig. 8. A one-slot buffer controlled by a split binary semaphore

$\{\mathbf{emp_s}\}$
with *empty* when $empty = 1$ do
$\left\{ \left(\mathbf{emp_s} \star \left(\begin{array}{l} (empty \vDash empty = 0) \vee \\ (empty, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash empty = 1 \wedge \boldsymbol{ws} = \boldsymbol{rs})) \end{array} \right) \right) \wedge empty = 1 \right\} \Rightarrow$
$\{empty, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash empty = 1 \wedge \boldsymbol{ws} = \boldsymbol{rs}\}$
$empty := 0$
$\{empty, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash empty = 0 \wedge \boldsymbol{ws} = \boldsymbol{rs}\} \Rightarrow$
$\{(b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \boldsymbol{ws} = \boldsymbol{rs}) \star (empty \vDash empty = 0)\} \Rightarrow$
$\left\{ (b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \boldsymbol{ws} = \boldsymbol{rs}) \star \left(\begin{array}{l} (empty \vDash empty = 0) \vee \\ (empty, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash empty = 1 \wedge \boldsymbol{ws} = \boldsymbol{rs})) \end{array} \right) \right\}$
od
$\{b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \boldsymbol{ws} = \boldsymbol{rs}\}$

Fig. 9. P(*empty*) releases ownership of variables

$\{w, \boldsymbol{ws}_{0.5} \vDash w = W \wedge \boldsymbol{ws} = WS\} \Rightarrow$
$\{(w, \boldsymbol{ws}_{0.5} \vDash w = W \wedge \boldsymbol{ws} = WS) \star \mathbf{emp_s}\}$
Framed: $\langle \{\mathbf{emp_s}\}\ \mathrm{P}(empty)\ \{b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \boldsymbol{ws} = \boldsymbol{rs}\} \rangle$
$\{(w, \boldsymbol{ws}_{0.5} \vDash w = W \wedge \boldsymbol{ws} = WS) \star (b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \boldsymbol{ws} = \boldsymbol{rs})\} \Rightarrow$
$\{w, b, \boldsymbol{ws}, \boldsymbol{rs}_{0.5} \vDash w = W \wedge \boldsymbol{ws} = WS \wedge \boldsymbol{ws} = \boldsymbol{rs}\}$
$b := w; \boldsymbol{ws} := \boldsymbol{ws}.\boldsymbol{b}$
$\{w, b, \boldsymbol{ws}, \boldsymbol{rs}_{0.5} \vDash w = b = W \wedge \boldsymbol{ws} = WS.b \wedge WS = \boldsymbol{rs}\} \Rightarrow$
$\{(w, \boldsymbol{ws}_{0.5} \vDash w = W \wedge \boldsymbol{ws} = WS.W) \star (b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \boldsymbol{ws} = \boldsymbol{rs}.b)\}$
Framed: $\langle \{b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \boldsymbol{ws} = \boldsymbol{rs}.b\}\ \mathrm{V}(full)\ \{\mathbf{emp_s}\} \rangle$
$\{(w, \boldsymbol{ws}_{0.5} \vDash w = W \wedge \boldsymbol{ws} = WS.W) \star \mathbf{emp_s}\} \Rightarrow$
$\{w, \boldsymbol{ws}_{0.5} \vDash w = W \wedge \boldsymbol{ws} = WS.W\}$

Fig. 10. Verification of the split-semaphore writer

Both the non-turn-taking fig. 2 and the turn-taking fig. 5 have semaphore-based counterparts. It suits our development best to analyse a version of the turn-taking one-slot buffer using a split binary semaphore (a pair of binary semaphores that are never both 1) shown in fig. 8. Once again, the program starts with a full buffer. In concurrent separation logic semaphore instructions are treated as stylised CCRs and we can treat our binary semaphores as follows:

$$\mathrm{P}(m) \triangleq \mathsf{with}\ m\ \mathsf{when}\ m = 1\ \mathsf{do}\ m := 0\ \mathsf{od} \tag{9}$$

$$\mathrm{V}(m) \triangleq \mathsf{with}\ m\ \mathsf{when}\ \mathsf{true}\ \mathsf{do}\ m := 1\ \mathsf{od} \tag{10}$$

The secret of the algorithm is that the variable b is always owned by the semaphore which is 1, as shown by the semaphore invariants. Fig. 9 shows that P(*empty*), if it terminates, releases ownership of b (as well as the auxiliary $\boldsymbol{ws}$ and $\boldsymbol{rs}$) into the executing thread. With similar reasoning we can calculate pre and postconditions for all four semaphore operations:

$$\{\mathbf{emp_s}\}\ \mathrm{P}(empty)\ \{b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \boldsymbol{ws} = \boldsymbol{rs}\} \tag{11}$$

$$\{b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \boldsymbol{ws} = \boldsymbol{rs}\}\ \mathrm{V}(empty)\ \{\mathbf{emp_s}\} \tag{12}$$

$$\{\mathbf{emp_s}\}\quad \mathrm{P}(full)\quad \{b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \boldsymbol{ws} = \boldsymbol{rs}.b\} \tag{13}$$

$$\{b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \boldsymbol{ws} = \boldsymbol{rs}.b\}\quad \mathrm{V}(full)\quad \{\mathbf{emp_s}\} \tag{14}$$

The reasoning in fig. 10, which shows that *write*(w) transmits a value, is very similar to fig. 6. A proof that *read*( ) extracts a value is similar to fig. 7, and works the other way round to fig. 10; we omit the details.

One subtly important difference between the split-semaphore algorithm of fig. 8 and the CCR algorithm of fig. 5 is in the treatment of mutual exclusion. The hardware implementation of P and V commands ensures mutual exclusion on the assignments which implement the semaphores, but the assignments $b := w$

$$
\begin{array}{l}
\mathsf{local}\, b = \mathit{null}, \mathit{full} = \mathrm{true}, \boldsymbol{ws} = \mathit{null}., \boldsymbol{rs} = . \;\mathsf{in} \\
\left(\begin{array}{l|l}
\ldots & \ldots \\
\mathit{write}(w) = & \mathit{read}(\,) = \\
\quad \mathsf{local}\, wt; & \quad \mathsf{local}\, y, rt; \\
\quad \mathsf{do}\langle\!\langle wt := \mathit{full} \rangle\!\rangle \;\mathsf{until}\, \neg wt; & \quad \mathsf{do}\langle\!\langle rt := \mathit{full} \rangle\!\rangle \;\mathsf{until}\, rt; \\
\quad b := w; \boldsymbol{ws} := \boldsymbol{ws}.\boldsymbol{b}; & \quad y := b; \boldsymbol{rs} := \boldsymbol{rs}.\boldsymbol{b}; \\
\quad \langle\!\langle \mathit{full} := \mathrm{true} \rangle\!\rangle & \quad \langle\!\langle \mathit{full} := \mathrm{false} \rangle\!\rangle; \\
 & \quad \mathsf{return}\, y
\end{array}\right)
\end{array}
$$

Fig. 11. A one-slot buffer mechanism using a condition variable, with double-angle-bracketed naturally-atomic commands

and $y := b$ are in critical sections and, though they ought to be and are in fact executed in mutual exclusion, the proof rather than the hardware must provide that guarantee.

The separation-logic explanation of how critical sections work rests on ownership. If threads start with separated resources, use CCRs to communicate, and preserve invariants as the CCR rule prescribes, then their resources stay separated. If the writer owns the b variable, as it will after P(*empty*) and before V(*full*), then the reader *cannot* own it, and so *cannot* attempt to read from it. Vice-versa, when the reader owns the b variable between P(*full*) and V(*empty*), it can read from it but the writer cannot attempt a write. Coherence is assured by alternation of ownership.

By using counting rather than binary semaphores and an array rather than a single variable, it is possible to build a bounded buffer in which the writer may deposit a queue of several values which the reader picks up in order, or an unbounded buffer which is a linked list of written values, with no waiting on either side but unbounded resource demands [O'H07]. Neither of these algorithms is in the scope of this paper: we are interested in non-queuing buffers with constant resource.

4.4. A one-slot buffer mechanism without synchronisation commands

Semaphores are essentially blocking mechanisms, because a thread which tries to P a 0-valued semaphore must wait in a busy-wait loop or in the queues of a thread manager. Simpson's algorithm is non-blocking, in that both writer and reader can always make progress, no matter what the other is doing. We need to step away from semaphores to a simpler mechanism. To do so we consider the single-slot single-place buffer in fig. 11.

This algorithm has no synchronisation instructions – no CCRs, no semaphores and no CAS instructions. Instead it depends, as many non-blocking algorithms do, on the fact that hardware stores serialise reads and writes to single store locations – typically 'words' of 32 or 64 bits in RAM stores nowadays, but even flash memory serialises single-bit accesses.[1] Each of the double-angle-bracketed commands in the program makes a single access (read or write) to a single shared variable *full* which could be implemented by a single storage bit and, because the hardware serialises those accesses, is 'naturally atomic'.

It is not hard to see how it works. The writer waits in a busy-wait loop until $\neg \mathit{full}$; the reader similarly waits until *full*. The writer only writes when $\neg \mathit{full}$ and the reader only reads when *full*; each side only enables the other's actions and not its own – the writer sets *full* and the reader $\neg \mathit{full}$. Surely their accesses to b will not overlap.

In the algorithm of fig. 11 the various possible instantaneous states of the shared resource are

S1. $\mathit{full}, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \mathit{full} \wedge \boldsymbol{ws} = \boldsymbol{rs}.b$
– the starting state and the state after the writer has written a value, with the writer one step ahead;

S2. $\mathit{full} \vDash \mathit{full}$
– after the reader has withdrawn b, $\boldsymbol{ws}$ and $\boldsymbol{rs}$;

S3. $\mathit{full}, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \neg \mathit{full} \wedge \boldsymbol{ws} = \boldsymbol{rs}$
– after the reader has consumed a value and put the variables back, with both sides in step;

S4. $\mathit{full} \vDash \neg \mathit{full}$
– after the writer has withdrawn b, $\boldsymbol{ws}$ and $\boldsymbol{rs}$.

[1] This 'fact' is not so factual as it once was. See the discussion of weak memory models in §8.

It is clear that the writer and reader cycle round these four states in order: it remains to verify that claim. Because of the sequencing of states, proof using the concurrent separation logic rule (5) would require at least one auxiliary variable. We can avoid that by making a proof in RGSep.

The writer's actions in fig. 11 – its guarantee and the rely of the reader – are

$$\mathit{full}, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \neg \mathit{full} \wedge \boldsymbol{ws} = \boldsymbol{rs} \rightsquigarrow \mathit{full} \vDash \neg \mathit{full} \tag{15}$$

$$\mathit{full} \vDash \neg \mathit{full} \rightsquigarrow \mathit{full}, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \mathit{full} \wedge \boldsymbol{ws} = \boldsymbol{rs}.b \tag{16}$$

– (15) says that it can withdraw ownership of b, $\boldsymbol{ws}$ and $\boldsymbol{rs}$ from the shared resource, transforming the state from S3 to S4, and (16) says that it can put those variables back, transforming the state from S4 to S1.

The reader's actions mesh with the writer's:

$$\mathit{full}, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs} \vDash \mathit{full} \wedge \boldsymbol{ws} = \boldsymbol{rs}.b \rightsquigarrow \mathit{full} \vDash \mathit{full} \tag{17}$$

$$\mathit{full} \vDash \mathit{full} \rightsquigarrow \mathit{full}, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \neg \mathit{full} \wedge \boldsymbol{ws} = \boldsymbol{rs} \tag{18}$$

– (17) says that the reader withdraws what the writer put there in (16), transforming the shared resource from S1 to S2, and (18) puts back what the writer needs in (15), going from S2 to S3.

The requirements of stability make things rather more complex than this simple state-machine picture. To reason about $\mathit{write}(w)$ we must first consider the states in which it might be called. If $\neg \mathit{full}$ it will be in state S3, because the reader has finished processing the last value which it transmitted. The reader will not transform that state into anything (neither of its actions can fire if $\neg \mathit{full}$), so it is stable. But if full, because the writer has transmitted a value, then the shared resource could be in state S1 or S2. So the state of shared resource at the beginning of $\mathit{write}(w)$ satisfies

$$(\mathit{full}, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \neg \mathit{full} \wedge \boldsymbol{ws} = \boldsymbol{rs}) \vee (\mathit{full}, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \mathit{full} \wedge \boldsymbol{ws} = \boldsymbol{rs}.b) \vee (\mathit{full} \vDash \mathit{full}) \tag{19}$$

This is stable: the reader will not do anything to the first disjunct, and it can only transform the second disjunct into the third (by (17)) or the third into the first (by (18)). It is not a precise formula, because the second and third disjuncts together are not precise – but no matter, because the formulae in our actions are precise, and therefore the resource splits in the rule (8) are precise. Because true matches any resource, we abbreviate (19) to

$$(\mathit{full}, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \neg \mathit{full} \wedge \boldsymbol{ws} = \boldsymbol{rs}) \vee ((\mathit{full} \vDash \mathit{full}) \star \mathrm{true}) \tag{20}$$

– also imprecise, and indeed more obviously so, but no more detailed than we need.

In RGSep only atomic instructions can access the shared store and so make actions. So in the writer we must show that $\langle\!\langle wt := \mathit{full} \rangle\!\rangle$ gets us from S3 to S4 with action (15) and then $\langle\!\langle \mathit{full} := \mathrm{true} \rangle\!\rangle$ gets us to state S1 with action (16). We can show what the first of these commands does if the shared resource matches the first disjunct of (20) – here sF is **emp**:

$$\begin{array}{l} \{(wt, w, \boldsymbol{ws}_{0.5} \vDash w = W \wedge \boldsymbol{ws} = WS) \star (\mathit{full}, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \neg \mathit{full} \wedge \boldsymbol{ws} = \boldsymbol{rs})\} \Rightarrow \\ \{wt, w, \mathit{full}, b, \boldsymbol{ws}, \boldsymbol{rs}_{0.5} \vDash w = W \wedge \boldsymbol{ws} = WS \wedge \neg \mathit{full} \wedge \boldsymbol{ws} = \boldsymbol{rs}\} \\ \quad wt := \mathit{full} \\ \{wt, w, \mathit{full}, b, \boldsymbol{ws}, \boldsymbol{rs}_{0.5} \vDash \neg wt \wedge w = W \wedge \boldsymbol{ws} = WS \wedge \neg \mathit{full} \wedge \boldsymbol{ws} = \boldsymbol{rs}\} \Rightarrow \\ \{(wt, w, b, \boldsymbol{ws}, \boldsymbol{rs}_{0.5} \vDash \neg wt \wedge w = W \wedge \boldsymbol{ws} = WS = \boldsymbol{rs}) \star (\mathit{full} \vDash \neg \mathit{full})\} \end{array} \tag{21}$$

This is a subtle argument. The command $wt := \mathit{full}$ does not do anything to the values of variables in the shared resource, but in action (15) $\mathit{full}, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \neg \mathit{full} \wedge \boldsymbol{ws} = \boldsymbol{rs} \rightsquigarrow \mathit{full} \vDash \neg \mathit{full}$ something happens to ownership of resource. In O'Hearn's phrase [O'H04] "ownership is in the eye of the prover": i.e. we arrange our assertions of ownership to suit our proof. Ownership transfer is not about the values of variables or heap cells, it is something that happens only in the proof.

It is easy to see that $wt := \mathit{full}$ does nothing at all to the shared resource – not even ownership transfer – if it matches the second disjunct of (20). Here sF is true:

$$\begin{array}{l} \{(wt, w, \boldsymbol{ws}_{0.5} \vDash w = W \wedge \boldsymbol{ws} = WS) \star (\mathit{full} \vDash \mathit{full})\} \Rightarrow \\ \{wt, w, \boldsymbol{ws}_{0.5}, \mathit{full} \vDash w = W \wedge \boldsymbol{ws} = WS \wedge \mathit{full}\} \\ \quad wt := \mathit{full} \\ \{wt, w, \boldsymbol{ws}_{0.5}, \mathit{full} \vDash wt \wedge w = W \wedge \boldsymbol{ws} = WS \wedge \mathit{full}\} \Rightarrow \\ \{(wt, w, \boldsymbol{ws}_{0.5} \vDash wt \wedge w = W \wedge \boldsymbol{ws} = WS) \star (\mathit{full} \vDash \mathit{full})\} \end{array} \tag{22}$$

The action in (21) is in the writer's G – it is (15) – and the action in (22) is the identity action

$\mathit{full} \vDash \mathit{full} \rightsquigarrow \mathit{full} \vDash \mathit{full}$, which is trivially included in G. By the specification-disjunction rule of Floyd-Hoare logic [Flo67]

$$\frac{\{Q\}\ C\ \{R\} \quad \{Q'\}\ C\ \{R'\}}{\{Q \vee Q'\}\ C\ \{R \vee R'\}} \tag{23}$$

we may conclude

$$\begin{array}{l} \left\{ \begin{array}{l} (wt, w, \boldsymbol{ws}_{0.5} \vDash w = W \wedge \boldsymbol{ws} = WS) \star \\ \quad ((\mathit{full}, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \neg \mathit{full} \wedge \boldsymbol{ws} = \boldsymbol{rs}) \vee ((\mathit{full} \vDash \mathit{full}) \star \mathrm{true})) \end{array} \right\} \\ \quad wt := \mathit{full} \\ \left\{ \begin{array}{l} ((wt, w, b, \boldsymbol{ws}, \boldsymbol{rs}_{0.5} \vDash \neg wt \wedge w = W \wedge \boldsymbol{ws} = WS = \boldsymbol{rs}) \star (\mathit{full} \vDash \neg \mathit{full})) \vee \\ \quad ((wt, w, \boldsymbol{ws}_{0.5} \vDash wt \wedge w = W \wedge \boldsymbol{ws} = WS) \star ((\mathit{full} \vDash \mathit{full}) \star \mathrm{true})) \end{array} \right\} \end{array} \tag{24}$$

and by the inclusion rules of RGSep, because each of the disjoint actions is included in G, we may conclude that the action of (24) is included in G.

Now (21) produces a shared-store postcondition that is stable under the reader's actions, but (22) does not, and it is not even stable when it is extended into $(\mathit{full} \vDash \mathit{full}) \star \mathrm{true}$: the reader may transform that by (17), if necessary, into $\mathit{full} \vDash \mathit{full}$ and then again by (18) into $\mathit{full}, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \neg \mathit{full} \wedge \boldsymbol{ws} = \boldsymbol{rs}$. $\langle\!\langle wt := \mathit{full} \rangle\!\rangle$ is certainly not the last atomic command in the body of $\mathit{write}(w)$, so we must weaken its postcondition to make it stable for the next command:

$$\begin{array}{l} \left\{ \begin{array}{l} (wt, w, \boldsymbol{ws}_{0.5} \vDash w = W \wedge \boldsymbol{ws} = WS) \star \\ \quad \boxed{(\mathit{full}, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \neg \mathit{full} \wedge \boldsymbol{ws} = \boldsymbol{rs}) \vee ((\mathit{full} \vDash \mathit{full}) \star \mathrm{true})} \end{array} \right\} \\ \quad \langle\!\langle wt := \mathit{full} \rangle\!\rangle \\ \left\{ \begin{array}{l} \Big((wt, w, b, \boldsymbol{ws}, \boldsymbol{rs}_{0.5} \vDash \neg wt \wedge w = W \wedge \boldsymbol{ws} = WS = \boldsymbol{rs}) \star \boxed{\mathit{full} \vDash \neg \mathit{full}} \Big) \ \vee \\ \quad \Big((wt, w, \boldsymbol{ws}_{0.5} \vDash wt \wedge w = W \wedge \boldsymbol{ws} = WS) \star \boxed{(\mathit{full} \vDash \mathit{full}) \star \mathrm{true}} \Big) \end{array} \right\} \Rightarrow \text{(stability)} \\ \left\{ \begin{array}{l} \Big((wt, w, b, \boldsymbol{ws}, \boldsymbol{rs}_{0.5} \vDash \neg wt \wedge w = W \wedge \boldsymbol{ws} = WS = \boldsymbol{rs}) \star \boxed{\mathit{full} \vDash \neg \mathit{full}} \Big) \ \vee \\ \quad \Big(\begin{array}{l} (wt, w, \boldsymbol{ws}_{0.5} \vDash wt \wedge w = W \wedge \boldsymbol{ws} = WS) \star \\ \quad \boxed{(\mathit{full}, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \neg \mathit{full} \wedge \boldsymbol{ws} = \boldsymbol{rs}) \vee ((\mathit{full} \vDash \mathit{full}) \star \mathrm{true})} \end{array} \Big) \end{array} \right\} \end{array} \tag{25}$$

Clearly, $\langle\!\langle wt := \mathit{full} \rangle\!\rangle$ picks out just the case we are interested in when wt is false and does nothing at all when wt is true. The second disjunct of the postcondition is the precondition, so we have a loop invariant when we find that wt holds, but we can be sure that the shared resource is $\mathit{full} \vDash \neg \mathit{full}$ when $\neg wt$.

Now we can see the effect of the body of $\mathit{write}(w)$, shown in fig. 12. In the treatment of $\langle\!\langle \mathit{full} := \mathrm{true} \rangle\!\rangle$ the shared-resource precondition $\mathit{full} \vDash \neg \mathit{full}$ is stable (the reader's actions require that full be true) and the action is (16). The instantaneous state after $\langle\!\langle \mathit{full} := \mathrm{true} \rangle\!\rangle$ is S1 and unstable, but the action of the reader can merely turn it into the precondition of the procedure call.

We can do the same sort of thing with the body of $\mathit{read}(\,)$, shown in fig. 13. The action of $\langle\!\langle rt := \mathit{full} \rangle\!\rangle$ is either (17) or the identity action $\mathit{full} \vDash \neg \mathit{full} \rightsquigarrow \mathit{full} \vDash \neg \mathit{full}$, and the shared-resource precondition is the dual of (20). The shared-resource precondition of $\mathit{full} :=$ false is $\mathit{full} \vDash \mathit{full}$, which is stable, and the action is (18). The final state, which is S3, is unstable, but the action of the writer can only transform it into the precondition of the procedure call.

This algorithm, then, can be understood as doing what the split-semaphore algorithm does: transferring ownership of the shared buffer variable to each thread in turn, so that each can access it locally and there can be no collisions. It has turn-taking, but it does not use a heavyweight synchronisation mechanism.

4.5. Doing without variable-ownership transfer

The reasoning in fig. 12 and fig. 13 depends on passing ownership of the buffer variable b in and out of the shared resource. But ownership transfer is a large sledgehammer for this concurrent nut. Surely in fig. 11 the writer writes to a shared variable b when $\neg \mathit{full}$, and the reader reads from the same variable when full, and

$$
\left\{\begin{array}{l}(wt, w, \boldsymbol{ws}_{0.5} \vDash w = W \wedge \boldsymbol{ws} = WS) \star \\ \boxed{(full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \neg full \wedge \boldsymbol{ws} = \boldsymbol{rs}) \vee ((full \vDash full) \star \text{true})}\end{array}\right\}
$$

do

$$
\left\{\begin{array}{l}(wt, w, \boldsymbol{ws}_{0.5} \vDash w = W \wedge \boldsymbol{ws} = WS) \star \\ \boxed{(full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \neg full \wedge \boldsymbol{ws} = \boldsymbol{rs}) \vee ((full \vDash full) \star \text{true})}\end{array}\right\}
$$

$\langle\!\langle wt := full \rangle\!\rangle$

$$
\left\{\begin{array}{l}\Big((wt, w, b, \boldsymbol{ws}, \boldsymbol{rs}_{0.5} \vDash \neg wt \wedge w = W \wedge \boldsymbol{ws} = WS = \boldsymbol{rs}) \star \boxed{full \vDash \neg full}\Big) \vee \\ \Big((wt, w, \boldsymbol{ws}_{0.5} \vDash wt \wedge w = W \wedge \boldsymbol{ws} = WS) \star \boxed{(full \vDash full) \star \text{true}}\Big)\end{array}\right\} \Rightarrow \text{(stability)}
$$

$$
\left\{\begin{array}{l}\Big((wt, w, b, \boldsymbol{ws}, \boldsymbol{rs}_{0.5} \vDash \neg wt \wedge w = W \wedge \boldsymbol{ws} = WS = \boldsymbol{rs}) \star \boxed{full \vDash \neg full}\Big) \vee \\ \left(\begin{array}{l}(wt, w, \boldsymbol{ws}_{0.5} \vDash wt \wedge w = W \wedge \boldsymbol{ws} = WS) \star \\ \boxed{(full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \neg full \wedge \boldsymbol{ws} = \boldsymbol{rs}) \vee ((full \vDash full) \star \text{true})}\end{array}\right)\end{array}\right\}
$$

until $\neg wt$

$$
\Big\{(wt, w, b, \boldsymbol{ws}, \boldsymbol{rs}_{0.5} \vDash \neg wt \wedge w = W \wedge \boldsymbol{ws} = WS = \boldsymbol{rs}) \star \boxed{full \vDash \neg full}\Big\}
$$

$b := w; \boldsymbol{ws} := \boldsymbol{ws}.\boldsymbol{b}$

$$
\Big\{(wt, w, b, \boldsymbol{ws}, \boldsymbol{rs}_{0.5} \vDash \neg wt \wedge w = b = W \wedge \boldsymbol{ws} = WS.b \wedge WS = \boldsymbol{rs}) \star \boxed{full \vDash \neg full}\Big\}
$$

$\langle\!\langle full := \text{true} \rangle\!\rangle$

$$
\Big\{(wt, w, \boldsymbol{ws}_{0.5} \vDash w = W \wedge \boldsymbol{ws} = WS.W) \star \boxed{full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash full \wedge \boldsymbol{ws} = \boldsymbol{rs}.b}\Big\}
$$

Fig. 12. Verification of the one-slot condition-variable writer

$$
\left\{\begin{array}{l}(y, rt, \boldsymbol{rs}_{0.5} \vDash \boldsymbol{rs} = RS) \star \\ \boxed{(full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash full \wedge \boldsymbol{ws} = \boldsymbol{rs}.b) \vee ((full \vDash \neg full) \star \text{true})}\end{array}\right\}
$$

do

$$
\left\{\begin{array}{l}(y, rt, \boldsymbol{rs}_{0.5} \vDash \boldsymbol{rs} = RS) \star \\ \boxed{(full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash full \wedge \boldsymbol{ws} = \boldsymbol{rs}.b) \vee ((full \vDash \neg full) \star \text{true})}\end{array}\right\}
$$

$\langle\!\langle rt := full \rangle\!\rangle$

$$
\left\{\begin{array}{l}\Big((y, rt, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs} \vDash rt \wedge \boldsymbol{rs} = RS \wedge \boldsymbol{ws} = \boldsymbol{rs}.b) \star \boxed{full \vDash full}\Big) \vee \\ \Big((y, rt \vDash \neg rt \wedge \boldsymbol{rs} = RS) \star \boxed{(full \vDash \neg full) \star \text{true}}\Big)\end{array}\right\} \Rightarrow \text{(stability)}
$$

$$
\left\{\begin{array}{l}\Big((y, rt, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs} \vDash rt \wedge \boldsymbol{rs} = RS \wedge \boldsymbol{ws} = \boldsymbol{rs}.b) \star \boxed{full \vDash full}\Big) \vee \\ \left(\begin{array}{l}(y, rt \vDash \neg rt \wedge \boldsymbol{rs} = RS) \star \\ \boxed{(full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash full \wedge \boldsymbol{ws} = \boldsymbol{rs}.b) \vee ((full \vDash \neg full) \star \text{true})}\end{array}\right)\end{array}\right\}
$$

until rt

$$
\Big\{(y, rt, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs} \vDash \boldsymbol{rs} = RS \wedge \boldsymbol{ws} = \boldsymbol{rs}.b) \star \boxed{full \vDash full}\Big\}
$$

$y := b; \boldsymbol{rs} := \boldsymbol{rs}.\boldsymbol{b}$

$$
\Big\{(y, rt, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs} \vDash y = b \wedge \boldsymbol{rs} = RS.b \wedge \boldsymbol{ws} = RS.b) \star \boxed{full \vDash full}\Big\}
$$

$\langle\!\langle full := \text{false} \rangle\!\rangle$

$$
\Big\{(y, rt, \boldsymbol{rs}_{0.5} \vDash \boldsymbol{rs} = RS.y) \star \boxed{full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \neg full \wedge \boldsymbol{ws} = \boldsymbol{rs}}\Big\}
$$

Fig. 13. Verification of the one-slot condition-variable reader

$$
\begin{array}{l}
\mathsf{local}\, b = \mathit{null}, \mathit{full} = \mathrm{true}, \boldsymbol{ws} = \mathit{null}., \boldsymbol{rs} = . \ \mathsf{in} \\
\left(\begin{array}{l|l}
\ldots & \ldots \\
\mathit{write}(w) = & \mathit{read}(\,) = \\
\quad \mathsf{local}\, wt; & \quad \mathsf{local}\, y, rt; \\
\quad \mathsf{do}\langle\!\langle wt := \mathit{full} \rangle\!\rangle \ \mathsf{until}\ \neg wt; & \quad \mathsf{do}\langle\!\langle rt := \mathit{full} \rangle\!\rangle \ \mathsf{until}\ rt; \\
\quad b := w; & \quad y := b; \\
\quad \langle\!\langle \mathit{full} := \mathrm{true}; \boldsymbol{ws} := \boldsymbol{ws}.\boldsymbol{b} \rangle\!\rangle & \quad \langle\!\langle \mathit{full} := \mathrm{false}; \boldsymbol{rs} := \boldsymbol{rs}.\boldsymbol{b} \rangle\!\rangle; \\
 & \quad \mathsf{return}\ y
\end{array}\right)
\end{array}
$$

Fig. 14. A one-slot buffer mechanism using a condition variable, not needing ownership transfer

those activities therefore simply cannot overlap. A proof based on that explanation should not need memory transfer.

We have noted that *full* can be implemented by a single bit, and supposed therefore that the hardware will serialise accesses to it. We do not want to suppose anything similar for b, or there would be nothing to prove. We suppose instead that accesses to b must be implemented as a sequence of smaller word-sized (or perhaps bit-sized) atomic accesses.

If we are certain that the value of a shared variable does not change whilst it is being read, then we can safely read it *non*-atomically. By observing that it will be implemented by the store as a sequence of atomic fragment reads, Vafeiadis [Vaf07] derives a rule to deal with a non-atomic read of a shared variable, which we state as follows:

$$
\frac{y = Y \star sF \text{ stable under } R}{\left\{(x \vDash x = \_) \star \boxed{\Gamma, y \vDash y = Y \star sF}\right\}\ x := y\ \left\{(x \vDash x = Y) \star \boxed{\Gamma, y \vDash y = Y \star sF}\right\}}\ \text{non-atomic read} \tag{26}
$$

– if $y = Y$ holds throughout the sequence of actions, then each fragment read is a fragment of Y and we will finish with $x = Y$.

Non-atomic writes are treated in the same way but require more side conditions. There must be no competing writers, G must declare to the environment that the writer can transform the shared variable arbitrarily (as each fragment is written), and the stable assertion cannot depend on the value of the variable being written.

$$
\frac{y = Y \star sF \text{ stable under } R \quad y = Y \star sF \rightsquigarrow y = Y' \star sF \subseteq G \quad y \notin \mathrm{fvs}(sF)}{\left\{(x \vDash x = X) \star \boxed{\Gamma, y \vDash y = \_ \star sF}\right\}\ y := x\ \left\{(x \vDash x = X) \star \boxed{\Gamma, y \vDash y = X \star sF}\right\}}\ \text{non-atomic write} \tag{27}
$$

If, for simplicity, we combine the auxiliary updates of $\boldsymbol{ws}$ and $\boldsymbol{rs}$ with the atomic updates of *full*[2] to give the algorithm in fig. 14, there are only two instantaneous states of the shared resource:

T1. $\mathit{full}, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \mathit{full} \wedge \boldsymbol{ws} = \boldsymbol{rs}.b$ – full, the writer one step ahead;
T2. $\mathit{full}, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \neg\mathit{full} \wedge \boldsymbol{ws} = \boldsymbol{rs}$ – empty, both sides in step.

The writer's actions are

$$
\mathit{full}, b \vDash \neg\mathit{full} \wedge b = B \rightsquigarrow \mathit{full}, b \vDash \neg\mathit{full} \wedge b = B' \tag{28}
$$

$$
\begin{array}{l}\mathit{full}, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \\ \quad \neg\mathit{full} \wedge \boldsymbol{ws} = \boldsymbol{rs} = RS \wedge b = B\end{array} \rightsquigarrow \begin{array}{l}\mathit{full}, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \\ \quad \mathit{full} \wedge \boldsymbol{ws} = \boldsymbol{rs}.b \wedge \boldsymbol{rs} = RS \wedge b = B\end{array} \tag{29}
$$

– (28) declares that when $\neg\mathit{full}$ it can overwrite b, fully or partially, and (29) says that it can simultaneously set *full* to true and extend $\boldsymbol{ws}$ with the value in b.

The reader's only action is

$$
\begin{array}{l}\mathit{full}, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \\ \quad \mathit{full} \wedge \boldsymbol{ws} = \boldsymbol{rs}.b = WS \wedge b = B\end{array} \rightsquigarrow \begin{array}{l}\mathit{full}, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \\ \quad \neg\mathit{full} \wedge \boldsymbol{ws} = \boldsymbol{rs} = WS \wedge b = B\end{array} \tag{30}
$$

– it can simultaneously set $\neg\mathit{full}$ and extend $\boldsymbol{rs}$.

With that preparation we can show the effect of $\mathit{write}(w)$ in fig. 15. The reasoning around $\langle\!\langle wt := \mathit{full} \rangle\!\rangle$

[2] It is legitimate to do this, because auxiliary assignments do not really happen and so take no real time.

$$\left\{ \begin{array}{l} (wt, w, \boldsymbol{ws}_{0.5} \vDash w = W \wedge \boldsymbol{ws} = WS) \star \\ \boxed{(full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \neg full \wedge \boldsymbol{ws} = \boldsymbol{rs}) \vee (full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash full \wedge \boldsymbol{ws} = \boldsymbol{rs}.b)} \end{array} \right\}$$

do

$$\left\{ \begin{array}{l} (wt, w, \boldsymbol{ws}_{0.5} \vDash w = W \wedge \boldsymbol{ws} = WS) \star \\ \boxed{(full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \neg full \wedge \boldsymbol{ws} = \boldsymbol{rs}) \vee (full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash full \wedge \boldsymbol{ws} = \boldsymbol{rs}.b)} \end{array} \right\}$$

$\langle\!\langle wt := full \rangle\!\rangle$

$$\left\{ \begin{array}{l} \left(\begin{array}{l} (wt, w, \boldsymbol{ws}_{0.5} \vDash \neg wt \wedge w = W \wedge \boldsymbol{ws} = WS) \star \\ \boxed{full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \neg full \wedge \boldsymbol{ws} = \boldsymbol{rs}} \end{array} \right) \vee \\ \left(\begin{array}{l} (wt, w, \boldsymbol{ws}_{0.5} \vDash wt \wedge w = W \wedge \boldsymbol{ws} = WS) \star \\ \boxed{full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash full \wedge \boldsymbol{ws} = \boldsymbol{rs}.b} \end{array} \right) \end{array} \right\} \Rightarrow \text{(stability)}$$

$$\left\{ \begin{array}{l} \left((wt, w, \boldsymbol{ws}_{0.5} \vDash \neg wt \wedge w = W \wedge \boldsymbol{ws} = WS) \star \boxed{full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \neg full \wedge \boldsymbol{ws} = \boldsymbol{rs}} \right) \vee \\ \left(\begin{array}{l} (wt, w, \boldsymbol{ws}_{0.5} \vDash wt \wedge w = W \wedge \boldsymbol{ws} = WS) \star \\ \boxed{(full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \neg full \wedge \boldsymbol{ws} = \boldsymbol{rs}) \vee (full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash full \wedge \boldsymbol{ws} = \boldsymbol{rs}.b)} \end{array} \right) \end{array} \right\}$$

until $\neg wt$

$$\left\{ (wt, w, \boldsymbol{ws}_{0.5} \vDash \neg wt \wedge w = W \wedge \boldsymbol{ws} = WS) \star \boxed{full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \neg full \wedge \boldsymbol{ws} = \boldsymbol{rs}} \right\}$$

— $\left\{ \begin{array}{l} full, b \vDash \neg full \wedge b = B \text{ stable under } R \\ full, b \vDash \neg full \wedge b = B \rightsquigarrow full, b \vDash \neg full \wedge b = B' \subseteq G \end{array} \right.$

$b := w$

$$\left\{ (wt, w, \boldsymbol{ws}_{0.5} \vDash \neg wt \wedge w = W \wedge \boldsymbol{ws} = WS) \star \boxed{full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \neg full \wedge \boldsymbol{ws} = \boldsymbol{rs} \wedge b = W} \right\}$$

$\langle\!\langle full := \text{true}; \boldsymbol{ws} := \boldsymbol{ws}.\boldsymbol{b} \rangle\!\rangle$

$$\left\{ (wt, w, \boldsymbol{ws}_{0.5} \vDash \neg wt \wedge w = W \wedge \boldsymbol{ws} = WS.W) \star \boxed{(full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash full \wedge \boldsymbol{ws} = \boldsymbol{rs}.b)} \right\}$$

Fig. 15. Verification of the one-slot non-atomic writer

and $\langle\!\langle full := \text{true}; \boldsymbol{ws} := \boldsymbol{ws}.\boldsymbol{b} \rangle\!\rangle$ is almost exactly as in fig. 12, except that there is no need of ownership transfer apart from the combination and splitting of permissions for $\boldsymbol{ws}$. We can similarly reason about the effect of *read*( ) in fig. 16.

In the arguments of fig. 12 and fig. 13 it was necessary to hypothesise an ownership-transfer action which was triggered by $\langle\!\langle wt := full \rangle\!\rangle$ and it was necessary, as a result, to be careful about precision. That was rather subtle – an 'action' that does not do anything visible to the shared resource? – but then accesses to b were done safely out of reach of the environment. In fig. 15 and fig. 16 the reasoning about the atomic actions is more straightforward, but the reasoning about the assignments, which are done in full visibility of an impotent environment, is more subtle.

It is difficult to decide which proof to prefer. We feel that the non-atomic version is nicer because it depends less on proof magic behind the scenes and corresponds more closely to a programmer's understanding of how the algorithm does its work. It also is more 'mechanical', in the sense that the reasoning does not depend on human insight about ownership, and that may one day make it more amenable to automation.

4.6. A one-slot buffer where the writer does not wait

Although it does not depend on hardware synchronisation beyond serialisation of single-bit accesses, fig. 11 still requires turn-taking and waiting: the reader must wait till the writer has finished, and vice-versa. If we are prepared to drop our standards, we can adapt it to an algorithm in which the writer does not wait, shown in fig. 17. The writer does not write if *full*, but instead simply discards a value. So the sequence $\boldsymbol{ws}$ written is a subsequence of the sequence $\boldsymbol{ps}$ produced. The reader still waits for *full*, though: it would be

$$
\left\{ \begin{array}{l} (y, rt \vDash \boldsymbol{rs} = RS) \star \\ \quad \boxed{(full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \neg full \wedge \boldsymbol{ws} = \boldsymbol{rs}) \vee (full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash full \wedge \boldsymbol{ws} = \boldsymbol{rs}.b)} \end{array} \right\}
$$

do

$$
\left\{ \begin{array}{l} (y, rt \vDash \boldsymbol{rs} = RS) \star \\ \quad \boxed{(full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \neg full \wedge \boldsymbol{ws} = \boldsymbol{rs}) \vee (full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash full \wedge \boldsymbol{ws} = \boldsymbol{rs}.b)} \end{array} \right\}
$$

$\langle\!\langle rt := full \rangle\!\rangle$

$$
\left\{ \begin{array}{l} \left((y, rt \vDash \neg rt \wedge \boldsymbol{rs} = RS) \star \boxed{full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \neg full \wedge \boldsymbol{ws} = \boldsymbol{rs}} \right) \vee \\ \left((y, rt \vDash rt \wedge \boldsymbol{rs} = RS) \star \boxed{full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash full \wedge \boldsymbol{ws} = \boldsymbol{rs}.b} \right) \end{array} \right\} \Rightarrow \text{(stability)}
$$

$$
\left\{ \begin{array}{l} \left((y, rt \vDash \neg rt \wedge \boldsymbol{rs} = RS) \star \boxed{full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \neg full \wedge \boldsymbol{ws} = \boldsymbol{rs}} \right) \vee \\ \left(\begin{array}{l} (y, rt \vDash \boldsymbol{rs} = RS) \star \\ \quad \boxed{(full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \neg full \wedge \boldsymbol{ws} = \boldsymbol{rs}) \vee (full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash full \wedge \boldsymbol{ws} = \boldsymbol{rs}.b)} \end{array} \right) \end{array} \right\}
$$

until rt

$$
\left\{ (y, rt \vDash \boldsymbol{rs} = RS) \star \boxed{full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash full \wedge \boldsymbol{ws} = \boldsymbol{rs}.b \wedge b = B} \right\}
$$

— $\{full, b \vDash full \wedge b = B$ stable under R

$y := b$

$$
\left\{ (y, rt \vDash \boldsymbol{rs} = RS \wedge y = B) \star \boxed{full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash full \wedge \boldsymbol{ws} = \boldsymbol{rs}.b \wedge b = B} \right\}
$$

$\langle\!\langle full := \text{false}; \boldsymbol{rs} := \boldsymbol{rs}.\boldsymbol{b} \rangle\!\rangle$

$$
\left\{ (y, rt \vDash \boldsymbol{rs} = RS.y) \star \boxed{full, b, \boldsymbol{ws}_{0.5}, \boldsymbol{rs}_{0.5} \vDash \neg full \wedge \boldsymbol{ws} = \boldsymbol{rs}} \right\}
$$

Fig. 16. Verification of the one-slot non-atomic reader

```
local b = null, full = true, ws = null., rs = . in
(  ...                          ||  ...
   write(w) =                   ||  read( ) =
     local wt;                  ||    local y, rt;
     ⟨⟨wt := full⟩⟩;             ||    do⟨⟨rt := full⟩⟩ until rt;
     if ¬wt then                ||    y := b;
       b := w;                  ||    ⟨⟨full := false; rs := rs.b⟩⟩;
       ⟨⟨full := true; ws := ws.b⟩⟩ ||  return y
     else skip fi               ||
)
```

Fig. 17. A one-slot lossy buffer

hard to see what to do if it did not (it might return a ⟨success,value⟩ pair, but that would make no real difference to an analysis).

Reasoning about this algorithm would be very similar to reasoning about fig. 11 or fig. 14, and in fact the reader and writer take turns to read and write even in this example. So the sequence written and the sequence read are the same. We omit the details.

4.7. The idealised algorithm revisited

The algorithm of fig. 1 explains, to a programmer, what single-place buffer communication means. We now have the logical machinery to specify and verify what it does. If we augment the algorithm to show the sequence of values transmitted and received, the writer's action is simply

$$
b = B \wedge \boldsymbol{ws} = WS \rightsquigarrow b = B' \wedge \boldsymbol{ws} = WS.B' \tag{31}
$$

$$\left\{ (w, \boldsymbol{ws}_{0.5} \vDash w = W \wedge \boldsymbol{ws} = WS) \star \boxed{b, \boldsymbol{rs}_{0.5}, \boldsymbol{ws}_{0.5} \vDash \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \wedge b = B} \right\}$$
$$\langle b := w; \boldsymbol{ws} := \boldsymbol{ws}.w \rangle$$
$$\left\{ (w, \boldsymbol{ws}_{0.5} \vDash w = W \wedge \boldsymbol{ws} = WS.W) \star \boxed{b, \boldsymbol{rs}_{0.5}, \boldsymbol{ws}_{0.5} \vDash \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = W \wedge b = W} \right\}$$

$$\left\{ (w, \boldsymbol{rs}_{0.5} \vDash \boldsymbol{rs} = RS) \star \boxed{\exists B(b, \boldsymbol{rs}_{0.5}, \boldsymbol{ws}_{0.5} \vDash \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \wedge b = B)} \right\}$$
$$\langle y := b; \boldsymbol{rs} := \boldsymbol{rs}.y \rangle$$
$$\left\{ \exists B \left(\begin{array}{l} (w, \boldsymbol{rs}_{0.5} \vDash y = B \wedge \boldsymbol{rs} = RS.B) \star \\ \boxed{b, \boldsymbol{rs}_{0.5}, \boldsymbol{ws}_{0.5} \vDash \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \wedge b = B} \end{array} \right) \right\} \Rightarrow \text{(stability)}$$
$$\left\{ \left(\exists B(w, \boldsymbol{rs}_{0.5} \vDash y = B \wedge \boldsymbol{rs} = RS.B) \star \boxed{\exists B'(b, \boldsymbol{rs}_{0.5}, \boldsymbol{ws}_{0.5} \vDash \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B' \wedge b = B')} \right) \right\}$$

Fig. 18. Verification of the idealised single-place buffer

– the writer can always alter the value in b and append the new value to the sequence written – and the reader's action is

$$b = B \wedge \boldsymbol{rs} = RS \rightsquigarrow b = B \wedge \boldsymbol{rs} = RS.B \tag{32}$$

– it can always add the value in b to the sequence read.

The verifications in fig. 18 show that these are indeed the actions of the algorithm. In the specification of the writer, B and W are universally quantified, and both pre and postconditions are stable. In the reader, B must be existentially quantified because the writer can alter b at any time; in the instantaneous postcondition that quantification expands to include both local and shared store, and when the postcondition is weakened for stability we break that link.

The algorithm communicates coherently because all accesses to shared store are in atomic instructions. It shows freshness because the value assigned to y in *read*( ) is at that instant the latest value written. It is fair because, magically, the atomic instructions execute instantaneously and so do not compete for execution time.

4.8. One-slot algorithms summarised

The ideal algorithm of fig. 1 magically provides coherent communication, albeit lossy and stuttering, with freshness and fairness. In showing a number of attempts to implement this ideal we have introduced and demonstrated the logical mechanisms, in particular the use of single-bit shared signalling variables in RGSep, which will enable us to storm the citadel of Simpson's algorithm. But our most 'realistic' algorithms, those which need no synchronisation mechanisms, have depended essentially on turn taking.

5. Two slots

Suppose that in place of a single shared variable we have a shared array $b[2]$ of two 'slots'.[3] We might hope to avoid waiting if the latest value is available to the reader in one slot whilst the other slot is left for the writer to fill: the reader can read from $b[L]$ while the writer can at the same time write to $b[1 - L]$.

It turns out that this idea needs clever implementation, and that it we cannot find a way to do it which completely eliminates waiting.

[3] In the examples which follow, we systematically gloss over a detail. In separation logic an array is a pointer to a sequence of heap cells; so $b[2]$ is shorthand for $b \mapsto \_, \_$, and accesses such as $b[i]$ are shorthand for $[b + i]$. In our examples the pointer value in b is shared but is not changed by either thread, so we do not count accesses to that variable when deciding whether an instruction is properly atomic. We might deal formally with the issue by distributing permission for b, one third each to reader, writer and shared resource, thereby showing that no thread can ever accumulate enough permission to change the pointer. But to do so would clutter our proofs without adding to understanding.

$$\mathsf{local}\, b[2] = (null, null), l = 0 \,\mathsf{in} \left(\begin{array}{l|l} \ldots & \ldots \\ write(w) = & read(\,) = \\ \quad \mathsf{local}\, wt \,\mathsf{in} & \quad \mathsf{local}\, y, rt \,\mathsf{in} \\ \quad\quad \langle\!\langle wt := 1 - l \rangle\!\rangle; & \quad\quad \langle\!\langle rt := l \rangle\!\rangle; \\ \quad\quad b[wt] := w; & \quad\quad y := b[rt]; \\ \quad\quad \langle\!\langle l := wt \rangle\!\rangle & \quad\quad \mathsf{return}\, y \end{array} \right)$$

(a) Writer chooses, but no slot is safe for the reader

$$\mathsf{local}\, b[2] = (null, null), l = 0, r = 0 \,\mathsf{in} \left(\begin{array}{l|l} \ldots & \ldots \\ write(w) = & read(\,) = \\ \quad \mathsf{local}\, wt \,\mathsf{in} & \quad \mathsf{local}\, y, rt \,\mathsf{in} \\ \quad\quad \langle\!\langle wt := 1 - r \rangle\!\rangle; & \quad\quad \langle\!\langle rt := l \rangle\!\rangle; \\ \quad\quad b[wt] := w; & \quad\quad \langle\!\langle r := rt \rangle\!\rangle; \\ \quad\quad \langle\!\langle l := wt \rangle\!\rangle & \quad\quad y := b[rt]; \\ & \quad\quad \mathsf{return}\, y \end{array} \right)$$

(b) Reader announces, but the writer cannot avoid it

Fig. 19. Incoherent two-slot buffers

$$\left(\begin{array}{r|c|l} writer & shared & reader \\ \hline & l = L & \langle\!\langle rt := l \rangle\!\rangle \\ & & \{rt = L\} \\ \langle\!\langle wt := 1 - l \rangle\!\rangle & & \\ \{wt = 1 - L\} & & \\ b[wt] := w & & \\ \langle\!\langle l := wt \rangle\!\rangle & & \\ & l = 1 - L & \\ \langle\!\langle wt := 1 - l \rangle\!\rangle & & \\ \{wt = L\} & & \\ b[wt] := w & & y := b[rt] \end{array} \right)$$

Fig. 20. Incoherence in fig. 19(a), with grey-background colliding instructions

5.1. Naive two-slot algorithms do not work

Fig. 19 shows two unsuccessful attempts to implement a two-slot algorithm, one adapted from [Sim90] and the other induced from [Sim97a]. In fig. 19(a) there is a shared one-bit variable l which the writer uses to signal which slot it has written the latest value into; the reader tries to read from $b[l]$ and the writer tries to write to $b[1 - l]$. But the writer inverts the value of l after it has written ($l := wt$ is in effect $l := 1 - l$), so if initially $l = L$, it will write first to $b[L]$, then to $b[1 - L]$, then to $b[L]$ ... and so on. There would be no slot that the reader could safely read from, and fig. 20 illustrates the collisions which can occur, with consequent possible incoherence.

The problem with the algorithm of fig. 19(a) is that the writer moves the goalposts after the reader has run up to shoot. But suppose the reader were to show in a second shared single-bit integer r which slot it intends to read from; then the writer might always write to $b[1 - r]$, as in fig. 19(b). This does not work either: there is an interleaving of atomic commands which can produce a collision and therefore incoherence if initially $r \neq l$, shown in fig. 21. It is easy to see that an execution of $write(w)$, without interference from a parallel execution of $read(\,)$, will produce $r \neq l$ so this is by no means an impossible state of affairs, and this time it is the reader which moves the goalposts after the writer has decided to shoot to miss.

$$\left(\begin{array}{l|l|l} & l = L \wedge r = 1 - L & \\ \langle\!\langle wt := 1 - r \rangle\!\rangle & & \\ \{wt = L\} & & \langle\!\langle rt := l \rangle\!\rangle \\ & & \{rt = L\} \\ & & \langle\!\langle r := rt \rangle\!\rangle \\ & l = L \wedge r = L & \\ b[wt] := w & & y := b[rt] \end{array} \right)$$

Fig. 21. Incoherence in fig. 19(b), with grey-background colliding instructions

$$
\begin{array}{l}
\mathsf{local}\, b[2] = (null, null), l = 0, r = 0, \boldsymbol{ws} = null., \boldsymbol{rs} = . \ \mathsf{in} \\
\left(\begin{array}{l|l}
\ldots & \ldots \\
write(w) = & read(\,) = \\
\quad \mathsf{local}\, wt, wt' \,\mathsf{in} & \quad \mathsf{local}\, y, rt \,\mathsf{in} \\
\quad\quad \langle\!\langle wt := 1 - l \rangle\!\rangle; & \quad\quad \langle\!\langle rt := l; \boldsymbol{rs} := \boldsymbol{rs}.\boldsymbol{b}[\boldsymbol{rt}] \rangle\!\rangle; \\
\quad\quad \mathsf{do} \langle\!\langle wt' := r \rangle\!\rangle \ \mathsf{until}\ wt \neq wt'; & \quad\quad \langle\!\langle r := rt \rangle\!\rangle; \\
\quad\quad b[wt] := w; & \quad\quad y := b[rt]; \\
\quad\quad \langle\!\langle l := wt; \boldsymbol{ws} := \boldsymbol{ws}.\boldsymbol{b}[\boldsymbol{wt}] \rangle\!\rangle & \quad\quad \mathsf{return}\, y
\end{array}\right)
\end{array}
$$

Fig. 22. A two-slot buffer in which the reader does not wait

5.2. Two slots where the writer waits

We introduced multiple slots in the hope that we could avoid waiting. It turns out that we cannot entirely avoid waiting with a two slot algorithm, but we can reduce it somewhat. In fig. 22 there are two shared index variables as in fig. 19(b), the writer manipulates l to make $l \neq r$, the reader manipulates r to make $r = l$, the writer waits until the reader has established $r = l$, but the reader runs free. It seems that this is the only way a two-slot buffer can work if we are to avoid collisions. The placing of the update to $\boldsymbol{rs}$ in the reader is perhaps surprising, with the first atomic instruction rather than with the assignment to y: it is in that position because it helps the argument about freshness by emphasising the fact that the value the reader will receive is determined as soon as it executes the first of its atomic instructions.

The writer's actions (omitting variable-ownership contexts for simplicity, and writing $b[E] = F$ as shorthand for $b + E \mapsto F$) are

$$r = l = R \star b[1 - l] = B \rightsquigarrow r = l = R \star b[1 - l] = B' \tag{33}$$

$$r = l = R \wedge \boldsymbol{ws} = WS \star b[1 - l] = B \rightsquigarrow r = R \wedge l = 1 - R \wedge \boldsymbol{ws} = WS.B \star b[l] = B \tag{34}$$

(33) changes $b[1 - l]$, but only when $r = l$; (34) in the same circumstances updates $\boldsymbol{ws}$ and switches l to point to the newly-written value.

The reader's actions (again omitting ownership context and using array-indexing notation) are

$$l = L \wedge \boldsymbol{rs} = RS \star b[l] = B \rightsquigarrow l = L \wedge \boldsymbol{rs} = RS.B \star b[l] = B \tag{35}$$

$$l = L \wedge r = 1 - L \rightsquigarrow r = l = L \tag{36}$$

(35) says that at any time the reader can add the value in $b[l]$ to $\boldsymbol{rs}$; (36) says that it can change r to make $r = l$.

In this algorithm the read sequence can stutter because the reader does not wait for a new value to be written before reading $b[rt]$. It will not necessarily read every value written, either: if it is slow the writer can switch l and get two steps ahead. But at every stage we have invariant $\lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = b[l]$. At the start of the program we have $r = l$, which is stable under the action of the reader, but the writer can change that to $l = 1 - r$.

We reason in fig. 23 about the writer. The atomic action $\langle\!\langle wt := 1 - l \rangle\!\rangle$ is easy to deal with, since the reader does not alter l, and generates the identity action. The action $\langle\!\langle wt' := r \rangle\!\rangle$ is not quite so easy: it instantaneously reads the value of r (and generates the identity action again), but that value is stable under the reader's actions only if $r = l$. We must weaken the postcondition for stability, and there are three possibilities:

- $l = L = r = R$: stable under the reader's actions;
- $l = L \neq r = R$: unstable under the reader's actions, which can change it into
- $l = L = r \neq R$: stable under the reader's actions.

Of these three only the first is selected by the loop guard $wt \neq wt'$; the others imply $l = L$, which is part of the loop invariant. The non-atomic write $b[wt] := w$ is then straightforward because the reader never alters either element of b. As in previous examples, the final shared-resource state is not stable but can only evolve into the proof precondition.

The reader is verified in fig. 24. Because the writer can alter the values in b and alter l we stabilise the precondition with an existential quantifier. The first atomic action $\langle\!\langle rt := l; \boldsymbol{rs} := \boldsymbol{rs}.\boldsymbol{b}[\boldsymbol{rt}] \rangle\!\rangle$ reads the value of l and adds $b[l]$ to $\boldsymbol{rs}$; instantaneously we have $\boldsymbol{ws}_\Omega = \boldsymbol{rs}_\Omega = b[l]$. We must weaken the postcondition for

$$\left\{w = W \wedge \boldsymbol{ws} = WS \star \boxed{l = L \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star b[l] = B \star b[1-l] = \_}\right\}$$
$$\langle\!\langle wt := 1 - l \rangle\!\rangle;$$
$$\left\{\begin{array}{l} w = W \wedge \boldsymbol{ws} = WS \wedge wt = 1 - L \star \\ \boxed{l = L \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star b[l] = B \star b[1-l] = \_} \end{array}\right\}$$
do
$$\left\{\begin{array}{l} w = W \wedge \boldsymbol{ws} = WS \wedge wt = 1 - L \star \\ \boxed{l = L \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star b[l] = B \star b[1-l] = \_} \end{array}\right\}$$
$$\langle\!\langle wt' := r \rangle\!\rangle$$
$$\left\{\begin{array}{l} w = W \wedge \boldsymbol{ws} = WS \wedge wt = 1 - L \wedge wt' = R \star \\ \boxed{l = L \wedge r = R \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star b[l] = B \star b[1-l] = \_} \end{array}\right\} \Rightarrow \text{(stability)}$$
$$\left\{\begin{array}{l} w = W \wedge \boldsymbol{ws} = WS \wedge wt = 1 - L \wedge wt' = R \star \\ \boxed{\begin{array}{l} ((l = L = r = R) \vee (l = L = r \neq R) \vee (l = L \neq r = R)) \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star \\ \quad b[l] = B \star b[1-l] = \_ \end{array}} \end{array}\right\}$$
until $wt \neq wt'$
$$\left\{\begin{array}{l} w = W \wedge \boldsymbol{ws} = WS \wedge wt = 1 - L \star \\ \boxed{r = l = L \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star b[l] = B \star b[1-l] = \_} \end{array}\right\}$$
$$\text{—} \left\{\begin{array}{l} b[1-L] = B \text{ stable under } R; \\ r = l = L \star b[1-L] = B \rightsquigarrow r = l = L \star b[1-L] = B' \subseteq G \end{array}\right.$$
$$b[wt] := w$$
$$\left\{w = W \wedge \boldsymbol{ws} = WS.W \star \boxed{r = l = L \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star b[l] = B \star b[1-l] = W}\right\}$$
$$\langle\!\langle l := wt; \boldsymbol{ws} := \boldsymbol{ws}.\boldsymbol{b}[\boldsymbol{wt}] \rangle\!\rangle$$
$$\left\{w = W \wedge \boldsymbol{ws} = WS.W \star \boxed{r = L = 1 - l \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = W \star b[1-l] = B \star b[l] = W}\right\}$$

Fig. 23. Verification of the two-slot writer

stability: once again we need a three-way disjunction, but this time it involves $\boldsymbol{ws}$ and the slots of b as well. Note, however, that the action of the writer preserves $\lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws}$ in each disjunct.

The action $\langle\!\langle r := rt \rangle\!\rangle$ reduces uncertainty by converting the third disjunct into the first. The non-atomic read $y := b[rt]$ is now straightforward: we have $rt = L$ locally; in each arm of the stable shared-store precondition we have $b[L] = B$; the writer can write when $r = l = L$ but only to $b[1-L]$; and it cannot write when $r \neq l$; so we have the condition for a coherent non-atomic read.

With this example we begin to have some of the properties we need to make a wait-free buffer. The reader can read faster than the writer, so can stutter, but it is guaranteed, when it begins by taking the value of l, that the slot it has thereby chosen (a) holds the then-latest element in the write sequence and (b) will not be overwritten before it has been read. The writer can get a little ahead of the reader, but only by two elements, and then it has to wait. Even though there are two slots, the reader is guaranteed to read the latest value written *at the instant it starts*, when it executes $\langle\!\langle rt := l \rangle\!\rangle$, so we have the freshness property. When the reader finishes the writer may have put a later value in the array and pointed l at it, but this does not matter: freshness is determined at the beginning of the read procedure call.

The proofs in fig. 15 and fig. 16 of the one-slot algorithm in fig. 14 required disjunction-juggling when showing how the loops distinguished one case from another. This time the proofs have throughout had to consider disjunctive assertions about the shared resource for the sake of stability. It seems an essential complication if we are to understand a multi-slot algorithm.

$$\left\{ \boldsymbol{rs} = RS \star \boxed{\exists B, L(l = L \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star b[L] = B \star b[1-L] = \_)} \right\}$$
$$\langle\!\langle rt := l; \boldsymbol{rs} := \boldsymbol{rs}.\boldsymbol{b}[\boldsymbol{rt}] \rangle\!\rangle$$
$$\left\{ \exists B, L \left(\begin{array}{l} \boldsymbol{rs} = RS.B \wedge rt = L \star \\ \boxed{l = L \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star b[L] = B \star b[1-l] = \_} \end{array} \right) \right\} \Rightarrow \text{(stability)}$$
$$\left\{ \exists B, L \left(\begin{array}{l} \boldsymbol{rs} = RS.B \wedge rt = L \star \\ \boxed{\begin{array}{l} (r = l = L \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star b[L] = B \star b[1-L] = \_) \vee \\ \exists B'(r = L \wedge l = 1 - L \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B' \star b[L] = B \star b[1-L] = B') \vee \\ (r = 1 - L \wedge l = L \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star b[L] = B \star b[1-L] = \_) \end{array}} \end{array} \right) \right\}$$
$$\langle\!\langle r := rt \rangle\!\rangle$$
$$\left\{ \exists B, L \left(\begin{array}{l} \boldsymbol{rs} = RS.B \wedge rt = L \star \\ \boxed{\begin{array}{l} (r = l = L \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star b[L] = B \star b[1-L] = \_) \vee \\ \exists B'(r = L \wedge l = 1 - L \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B' \star b[L] = B \star b[1-L] = B') \end{array}} \end{array} \right) \right\}$$
— $\{b[L] = B$ stable under R
$y := b[rt]$
$$\left\{ \exists B, L \left(\begin{array}{l} \boldsymbol{rs} = RS.y \wedge y = B \star \\ \boxed{\begin{array}{l} (r = l = L \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star b[l] = B \star b[1-l] = \_) \vee \\ \exists B'(r = L \wedge l = 1 - L \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B' \star b[L] = B \star b[l] = B') \end{array}} \end{array} \right) \right\}$$

Fig. 24. Verification of the two-slot reader

$\mathsf{local}\, c, l = 0, b[2] = (null, null), \boldsymbol{wa} = \text{false}, \boldsymbol{ws} = null., \boldsymbol{rs} = .\ \mathsf{in}$

```
( ...                                  ||  ...
  write(w) =                           ||  read( ) =
    local wt in                        ||    local rt in
      ⟨⟨c := true; wa := true⟩⟩;        ||    do  ⟨⟨c := false⟩⟩;
      ⟨⟨wt := 1 − l⟩⟩;                  ||        ⟨⟨rt := l⟩⟩;
      b[wt] := w;                      ||        y := b[rt];
      ⟨⟨l := wt; wa := false;          ||        ⟨⟨rt := c⟩⟩
        ws := ws.w⟩⟩;                  ||    until ¬rt;
                                       ||    ⟨⟨rs := rs.y⟩⟩;
                                       ||    return y
)
```

Fig. 25. Minimal colliding two-slot with auxiliary annotation

5.3. A colliding two-slot algorithm

So far, as in Simpson's developments [Sim90, Sim97a], each of our algorithms has allowed only mutually exclusive reads and writes to a buffer slot. Non-exclusive accesses – collisions – produce incoherence. But if a collision could be detected, perhaps the reader could try again.[4] This is a classic 'optimistic' concurrency strategy of non-blocking algorithms: try to perform an action; test to see if it worked and if so, exit; if it did not work or might not have worked, try again. Note that such a mechanism is non-*blocking* in that a thread never has to stop executing, but neither wait-free nor fair, because there is no guarantee of eventual success.

Tim Harris (private communication) proposed a wait-free non-blocking algorithm, the three-slot buffer discussed in §6 below, which inspired us to consider the two-slot mechanism in fig. 25. As well as the $\boldsymbol{ws}$ and $\boldsymbol{rs}$ required to account for the sequences written and read, we have had to add an auxiliary $\boldsymbol{wa}$ ('*w*riter *a*ctive') to help reasoning about sequencing of the writer's actions – even in RGSep we must use auxiliary variables to describe program position when we need to reason about sequencing of actions in R.

[4] We assume that reads do not interfere with simultaneous writes, but writes may partially overwrite a slot and thus allow an incoherent read.

$$\left(\begin{array}{r|c|l}
\langle\!\langle c := \text{true} \rangle\!\rangle & c \wedge l = L & \\
\{wt = 1 - L\} & & \\
 & & \langle\!\langle c := \text{false} \rangle\!\rangle \\
 & \neg c \wedge l = L & \\
 & & \langle\!\langle rt := l \rangle\!\rangle \\
 & & \{rt = L\} \\
\langle\!\langle l := wt \rangle\!\rangle & & \\
 & \neg c \wedge l = 1 - L & \\
\ldots & & \\
\langle\!\langle c := \text{true} \rangle\!\rangle & & \\
 & c \wedge l = 1 - L & \\
\langle\!\langle wt := 1 - l \rangle\!\rangle & & \\
\{wt = L\} & & \\
b[wt] := w & & y := b[rt]
\end{array}\right)$$

Fig. 26. Incoherence in fig. 25

The algorithm is based on the writer-chooses algorithm of fig. 19(a), with a single index variable l. The shared single-bit variable c (for '$c$ollide') detects the combination of circumstances shown in fig. 20 in which the reader picks a slot, while it is active the writer safely writes to the other slot, but then the writer changes l and starts a second transfer into the slot which the reader picked. So far as the reader is concerned, the writer is dangerous when it executes $\langle\!\langle l := wt \rangle\!\rangle$ (effectively $\langle\!\langle l := 1 - l \rangle\!\rangle$) followed by $\langle\!\langle wt := 1 - l \rangle\!\rangle$ and then $b[wt] := w$ – but the algorithm of fig. 25 cannot produce that sequence of actions without executing $\langle\!\langle c := \text{true} \rangle\!\rangle$ along the way, and we can see an example in fig. 26 in which c is true when the collision happens.

One example does not make a proof but it may point the way to a proof. The writer's actions (with variable context omitted and using array-indexing notation) are

$$\neg \boldsymbol{wa} \rightsquigarrow c \wedge \boldsymbol{wa} \tag{37}$$

$$\boldsymbol{wa} \wedge l = L \star b[1 - l] = B \rightsquigarrow \boldsymbol{wa} \wedge l = L \star b[1 - l] = B' \tag{38}$$

$$\boldsymbol{wa} \wedge l = L \wedge \boldsymbol{ws} = WS \star b[1 - l] = B \rightsquigarrow \neg \boldsymbol{wa} \wedge l = 1 - L \wedge \boldsymbol{ws} = WS.B \star b[l] = B \tag{39}$$

The reader's actions are

$$c \rightsquigarrow \neg c \tag{40}$$

$$\boldsymbol{rs} = RS \wedge \lfloor \boldsymbol{rs}.B \rfloor \preccurlyeq \boldsymbol{ws} \rightsquigarrow \boldsymbol{rs} = RS.B \tag{41}$$

As usual these actions keep $\lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = b[l]$ invariant.

Verification of the writer is shown in fig. 27. After $\langle\!\langle c := true; \boldsymbol{wa} := true \rangle\!\rangle$, which produces action (37), we have to weaken the postcondition to forget the value of c, because the reader can set it to false. But then, because the reader never changes anything in the shared store, both $\langle\!\langle wt := 1 - l \rangle\!\rangle$ and the non-atomic write $b[wt] := w$ are unproblematic.

Verification of the reader is more subtle and shown in fig. 28. First, $\langle\!\langle c := \text{false} \rangle\!\rangle$ performs either action (40) or the identity action, depending on the value of c beforehand. Instantaneously afterwards we have $\neg c \wedge l = L \star b[L] = B$. If $\boldsymbol{wa}$ holds the writer can write to $b[1 - L]$ with (38), but that is no problem. If it changes the value of l with action (39) it will set $\neg \boldsymbol{wa}$, and then to write to $b[L]$ it must first set $c \wedge \boldsymbol{wa}$ with (37). This gives us a three-way disjunction: either nothing has happened to c, l and $b[L]$, or l has changed and a new element is last in $\boldsymbol{ws}$, or c has changed and we are no longer sure what is in the slots of b. Note that in the second arm of the disjunction the value B' is a valid extension of $\boldsymbol{rs}$ and is later than B, the value which was last when the reader executed $\langle\!\langle c := \text{false} \rangle\!\rangle$. If the reader successfully exits the loop $\mathsf{do} \ldots \mathsf{until}\ \neg rt$ then it will have retrieved either the value which was last when it started the loop, or the next in the written sequence: freshness is assured provided the reader doesn't starve.

After $\langle\!\langle rt := l \rangle\!\rangle$ we have, instantaneously, $rt = l = L''$, which is unstable. But if $\neg c$ continues to hold then, whatever else happens, $b[L'']$ will not be overwritten, is a valid extension to $\boldsymbol{rs}$, and is at least as fresh as the value B which was freshest at the beginning of the loop. We express this with an implication and must apply a non-separating conjunction to ensure that whatever the value of c we still have the same heap resource.

The non-atomic read rule (26) is inadequate for this proof, because $b[L''] = B$ is not stable. We need

$$\left\{w = W \wedge \boldsymbol{ws} = WS \star \boxed{\neg \boldsymbol{wa} \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star b[l] = B \star b[1-l] = \_}\right\}$$
$$\langle\!\langle c := \text{true}; \boldsymbol{wa} := \textbf{true} \rangle\!\rangle$$
$$\left\{w = W \wedge \boldsymbol{ws} = WS \star \boxed{c \wedge \boldsymbol{wa} \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star b[l] = B \star b[1-l] = \_}\right\} \Rightarrow \text{(stability)}$$
$$\left\{w = W \wedge \boldsymbol{ws} = WS \star \boxed{\boldsymbol{wa} \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star b[l] = B \star b[1-l] = \_}\right\}$$
$$\langle\!\langle wt := 1 - l \rangle\!\rangle$$
$$\left\{\begin{array}{l} wt = 1 - L \wedge w = W \wedge \boldsymbol{ws} = WS \star \\ \quad \boxed{\boldsymbol{wa} \wedge l = L \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star b[l] = B \star b[1-l] = \_} \end{array}\right\}$$
$$\text{—} \left\{\begin{array}{l} b[1-l] = B' \text{ stable under } R; \\ \boldsymbol{wa} \wedge b[1-l] = B \rightsquigarrow \boldsymbol{wa} \wedge b[1-l] = B' \subseteq G \end{array}\right.$$
$$b[wt] := w$$
$$\left\{\begin{array}{l} wt = 1 - L \wedge w = W \wedge \boldsymbol{ws} = WS \star \\ \quad \boxed{\boldsymbol{wa} \wedge l = L \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star b[l] = B \star b[1-l] = W} \end{array}\right\}$$
$$\langle\!\langle l := wt; \boldsymbol{wa} := \textbf{false}; \boldsymbol{ws} := \boldsymbol{ws}.\boldsymbol{w} \rangle\!\rangle$$
$$\left\{w = W \wedge \boldsymbol{ws} = WS.W \star \boxed{\neg \boldsymbol{wa} \wedge l = L \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = W \star b[1-l] = B \star b[l] = W}\right\}$$

Fig. 27. Verification of the two-slot colliding writer

an enhanced rule to deal with an implied stability. To derive one,[5] we note once again that a non-atomic read will be executed as a sequence of atomic reads, because the store automatically serialises reads at *some* level of granularity. We can suppose that y and $b[rt]$ are made up of k separate pieces, and the non-atomic read $y := b[rt]$ is executed as the sequence $\langle\!\langle y_1 := b[rt]_1 \rangle\!\rangle; \langle\!\langle y_2 := b[rt]_2 \rangle\!\rangle; \ldots \langle\!\langle y_k := b[rt]_k \rangle\!\rangle$. Then for any particular element of this sequence we may deduce, using the Floyd-Hoare disjunction rule

$$\left\{rt = L \star \boxed{c \vee b[L] = B}\right\} \ \langle\!\langle y_i := b[rt]_i \rangle\!\rangle \ \left\{\exists X_i \left(rt = L \wedge y_i = X_i \star \boxed{c \vee (X_i = B_i \star b[L] = B)} \right)\right\} \tag{42}$$

– we have read a segment of $b[L]$ and if $\neg c$ holds that is also a segment of B. If c is stable, so we cannot have $\neg c$ then c then $\neg c$, then if $\neg c$ 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 B.

The following rule is thus derived to deal with a non-atomic read from a shared heap cell to a local variable, which is what is needed for the proof in fig. 28 (it would be no more difficult to deal with a read from a shared variable):

$$\frac{\exists v(E \mapsto v \star P(v)),\ (D \Rightarrow E \mapsto V) \star P(V),\ \neg D \text{ stable under } R}{\left\{x = \_ \star \boxed{\exists v(E \mapsto v \star P(v))}\right\}\ x := [E]\ \left\{\exists v' \left(x = v' \star \boxed{\exists v(E \mapsto v \star P(v) \star (D \Rightarrow v = v'))} \right)\right\}}\ \text{non-atomic read} \tag{43}$$

- the shared resource must include the heap cell throughout the read: $\exists v(E \mapsto v \star P(v))$ stable under R;
- when D holds, the value in the heap cell is fixed: $(D \Rightarrow E \mapsto V) \star P(V)$ stable under R;
- D does not change value from false to true during the read: $\neg D$ stable under R.

In fig. 28 D is $\neg c$; E is $b + L''$; V is B; P is $\lfloor \boldsymbol{rs}.B \rfloor \preccurlyeq \boldsymbol{ws}$. At the end of $y := b[rt]$ we can be sure that if $\neg c$ holds, then the value in y is also the value in $b[l]$.

The rest is straightforward. Because we must not contaminate $\boldsymbol{rs}$ with values that are never returned from *read*( ), we place the update of $\boldsymbol{rs}$ after successful exit from the loop. Note also that we have freshness: after $\langle\!\langle rt := l \rangle\!\rangle$ the reader is guaranteed the value then last in $\boldsymbol{ws}$, if it manages to complete $y := b[rt]$ before c is set by the writer.

This is a startling algorithm, since it permits but escapes from incoherence. The writer never waits, and the reader never blocks; but it is possible that the reader could starve indefinitely if it is unlucky in its

[5] Our derivation of rule (43) and the rule itself were suggested to us by Vafeiadis and Parkinson (private communication).

$$\left\{\boldsymbol{rs} = RS \star \boxed{\exists B, L(l = L \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star b[L] = B \star b[1-L] = \_)}\right\}$$

do

$$\left\{\boldsymbol{rs} = RS \star \boxed{\exists B, L(l = L \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star b[L] = B \star b[1-L] = \_)}\right\}$$

$\langle\!\langle c := \text{false} \rangle\!\rangle$

$$\left\{\boldsymbol{rs} = RS \star \boxed{\exists B, L(l = L \wedge \neg c \wedge \lfloor \boldsymbol{rs}.B \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star b[L] = B \star b[1-L] = \_)}\right\} \Rightarrow \text{(stability)}$$

$$\left\{\begin{array}{l}\boldsymbol{rs} = RS \star \\ \boxed{\exists B, L \left(\begin{array}{l}(l = L \wedge \neg c \wedge \lfloor \boldsymbol{rs}.B \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star b[L] = B \star b[1-L] = \_) \vee \\ \exists B'(l = 1 - L \wedge \neg c \wedge \neg \boldsymbol{wa} \wedge \lfloor \boldsymbol{rs}.B.B' \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B' \star \\ \quad b[L] = B \star b[1-L] = B') \vee \\ \exists B'(c \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B' \star b[L] = B' \star b[1-L] = \_)\end{array}\right)}\end{array}\right\}$$

$\langle\!\langle rt := l \rangle\!\rangle$;

$$\left\{\exists L'' \left(\begin{array}{l}\boldsymbol{rs} = RS \wedge rt = L'' \star \\ \boxed{\exists B, L \left(\begin{array}{l}\left(\begin{array}{l}l = L = L'' \wedge \neg c \wedge \lfloor \boldsymbol{rs}.B \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star \\ b[L] = B \star b[1-L] = \_\end{array}\right) \vee \\ \exists B' \left(\begin{array}{l}l = 1 - L = L'' \wedge \neg c \wedge \neg \boldsymbol{wa} \wedge \\ \lfloor \boldsymbol{rs}.B.B' \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B' \star \\ b[L] = B \star b[1-L] = B'\end{array}\right) \vee \\ \exists B' \left(\begin{array}{l}c \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B' \star \\ b[L'] = B' \star b[1-L] = \_\end{array}\right)\end{array}\right)}\end{array}\right)\right\} \Rightarrow \text{(stability)}$$

$$\left\{\exists L'' \left(\begin{array}{l}\boldsymbol{rs} = RS \wedge rt = L'' \star \\ \boxed{\begin{array}{l}(\neg c \Rightarrow \exists B(\lfloor \boldsymbol{rs}.B \rfloor \preccurlyeq \boldsymbol{ws} \star b[L''] = B \star b[1-L''] = \_)) \wedge \\ \exists B'(\lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B' \wedge b[l] = B' \star b[1-l] = \_)\end{array}}\end{array}\right)\right\}$$

— $\{\exists v(b[L] = v), \neg c \Rightarrow b[L] = B, c \text{ stable under } R$

$y := b[rt]$

$$\left\{\exists B'', L'' \left(\begin{array}{l}\boldsymbol{rs} = RS \wedge rt = L'' \wedge y = B'' \star \\ \boxed{\begin{array}{l}(\neg c \Rightarrow \exists B(B = B'' \wedge \lfloor \boldsymbol{rs}.B \rfloor \preccurlyeq \boldsymbol{ws} \star b[L''] = B \star b[1-L] = \_)) \wedge \\ \exists B'(\lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B' \wedge b[l] = B' \star b[1-l] = \_)\end{array}}\end{array}\right)\right\}$$

$\langle\!\langle rt := c \rangle\!\rangle$

$$\left\{\begin{array}{l}\left(\boldsymbol{rs} = RS \wedge rt \star \boxed{c \wedge \exists B'(\lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B' \wedge b[l] = B' \star b[1-l] = \_)}\right) \vee \\ \exists B'' \left(\begin{array}{l}\boldsymbol{rs} = RS \wedge \neg rt \wedge y = B'' \star \\ \boxed{\begin{array}{l}(\neg c \wedge \lfloor \boldsymbol{rs}.B'' \rfloor \preccurlyeq \boldsymbol{ws} \star b[l] = B'' \star b[1-l] = \_) \wedge \\ \exists B'(\lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B' \wedge b[l] = B' \star b[1-l] = \_)\end{array}}\end{array}\right)\end{array}\right\} \Rightarrow \text{(stability)}$$

$$\left\{\begin{array}{l}\left(\boldsymbol{rs} = RS \wedge rt \star \boxed{c \wedge \exists B'(\lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B' \wedge b[l] = B' \star b[1-l] = \_)}\right) \vee \\ \exists B \left(\begin{array}{l}\boldsymbol{rs} = RS \wedge \neg rt \wedge y = B \star \\ \boxed{\lfloor \boldsymbol{rs}.B \rfloor \preccurlyeq \boldsymbol{ws} \wedge \exists B'(\lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B' \wedge b[l] = B' \star b[1-l] = \_)}\end{array}\right)\end{array}\right\}$$

until $\neg rt$

$$\left\{\exists B \left(\begin{array}{l}\boldsymbol{rs} = RS \wedge y = B \star \\ \boxed{\lfloor \boldsymbol{rs}.B \rfloor \preccurlyeq \boldsymbol{ws} \wedge \exists B'(\lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B' \wedge b[l] = B' \star b[1-l] = \_)}\end{array}\right)\right\}$$

$\langle\!\langle \boldsymbol{rs} := \boldsymbol{rs}.\boldsymbol{y} \rangle\!\rangle$

$$\left\{\boldsymbol{rs} = RS.y \star \boxed{\exists B'(\lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B' \wedge b[l] = B' \star b[1-l] = \_)}\right\}$$

Fig. 28. Verification of the two-slot colliding reader

$$
\begin{array}{l}
\mathsf{local}\, l = 0, r = 0, b[3] = (null, null, null)\ \mathsf{in} \\
\left(\begin{array}{l|l}
\ldots & \ldots \\
write(w) = & read(\,) = \\
\quad \mathsf{local}\, wt, wt'\ \mathsf{in} & \quad \mathsf{local}\, rt\ \mathsf{in} \\
\quad\quad \langle\!\langle wt := l \rangle\!\rangle; & \quad\quad \langle\!\langle rt := l \rangle\!\rangle; \\
\quad\quad \langle\!\langle wt := \mathrm{differ}(r, wt) \rangle\!\rangle; & \quad\quad \langle\!\langle r := rt \rangle\!\rangle; \\
\quad\quad b[wt] := w; & \quad\quad y := b[rt]; \\
\quad\quad \langle\!\langle l := wt \rangle\!\rangle; & \quad\quad \mathsf{return}\, y
\end{array}\right)
\end{array}
$$

Fig. 29. An incoherent three-slot buffer

$$
\left(\begin{array}{r|l|l}
\{wt = 2\} & r = 0 \wedge l = 1 & \\
 & & \langle\!\langle rt := l \rangle\!\rangle \\
 & & \{rt = 1\} \\
\langle\!\langle l := wt \rangle\!\rangle & & \\
 & r = 0 \wedge l = 2 & \\
\langle\!\langle wt := l \rangle\!\rangle & & \\
\{wt = 2\} & & \\
\langle\!\langle wt := \mathrm{differ}(r, wt) \rangle\!\rangle & & \\
\{wt = 1\} & & \\
 & & \langle\!\langle r := rt \rangle\!\rangle \\
 & r = 1 \wedge l = 2 & \\
b[wt] := w & & y := b[rt]
\end{array}\right)
$$

Fig. 30. Incoherence in fig. 29

timing. It is obstruction-free [HLM03] at least (the writer always terminates, and the reader terminates if the writer is not active), but not wait-free so not a complete solution to the asynchronous buffer problem.

6. Three slots

The two-slot buffers of fig. 19 are incoherent because there is no slot from which the reader can safely read. If we have three slots, perhaps the reader could be reading from one and the writer could alternate between the other two. Fig. 29 shows an attempt: we presume atomic access to the shared two-bit variables r and l (that is a larger assumption than the single-bit atomicity of earlier examples) and $\mathrm{differ}(E, F)$ delivers a value in $\{0, 1, 2\}$ which is distinct from either of its arguments. It would seem at first sight that the writer never writes to the $b[r]$ or $b[l]$ slots, and that the reader would therefore be perfectly safe with its atomic read. As Simpson [Sim90] has shown, however, this algorithm is incoherent, and as with fig. 19(b), the problem is that there can be a delay between $\langle\!\langle rt := l \rangle\!\rangle$ – the reader decides which slot to use – and $\langle\!\langle r := rt \rangle\!\rangle$ – the information is made available to the writer – during which time the writer can move the goalposts (fig. 30).

This problem can be fixed, with some considerable complexity, but it is not worthwhile to do so. Indeed, we do not know a coherent version of this three-slot buffer that does any better than the two-slot algorithm of fig. 22.

Tim Harris (op. cit) proposes an alternative approach, in which collision (incoherence) is detected and avoided. It is a three-slot buffer algorithm, shown in our notation in fig. 31. It uses the same signalling mechanism as the colliding two-slot algorithm of fig. 25, but in addition the writer detects a potential collision situation and copies the value that the reader would be seeking into a shared variable b'. The reader, if it seems that there has been a collision, can read from b'. At the cost of sometimes forcing a double write and sometimes a double read, we achieve full asynchronous wait-free behaviour.

When the reader has finished it sets c to reduce unnecessary copying in the writer. To a programmer this is an obvious thing to do, but it causes a slight complication in the proof. Without that assignment the reader would only set $\neg c$ and the writer only c. It would then be obvious that reader and writer alternate in use of b': the writer writes when $\neg c$, which is stable under the actions of the reader; and the reader reads when c, which is stable under the actions of the writer. With the extra assignment we have to add an additional

$$
\begin{array}{l}
\mathsf{local}\, c = \mathrm{true}, l = 0, b[2] = (null, null), b', \boldsymbol{ws} = null., \boldsymbol{rs} = ., \boldsymbol{wa} = \mathrm{false}, \boldsymbol{ra} = \mathrm{false}\, \mathsf{in} \\
\left(\begin{array}{l|l}
\ldots & \ldots \\
write(w) = & read(\,) = \\
\quad \mathsf{local}\, wt\, \mathsf{in} & \quad \mathsf{local}\, rt\, \mathsf{in} \\
\quad\quad \langle\!\langle wt := c; \boldsymbol{wa} := \boldsymbol{c} \rangle\!\rangle; & \quad\quad \langle\!\langle c := \mathrm{false}; \boldsymbol{ra} := \mathbf{true} \rangle\!\rangle; \\
\quad\quad \mathsf{if}\, \neg wt\, \mathsf{then} & \quad\quad \langle\!\langle rt := l \rangle\!\rangle; \\
\quad\quad\quad \mathsf{local}\, w'\, \mathsf{in} & \quad\quad y := b[rt]; \\
\quad\quad\quad\quad w' := b[l]; b' := w' & \quad\quad \langle\!\langle rt := c \rangle\!\rangle; \\
\quad\quad\quad \langle\!\langle c := \mathrm{true}; \boldsymbol{wa} := \mathbf{true} \rangle\!\rangle & \quad\quad \mathsf{if}\, rt\, \mathsf{then}\, y := b'\, \mathsf{else\, skip\, fi}; \\
\quad\quad \mathsf{else\, skip\, fi}; & \quad\quad \langle\!\langle c := \mathrm{true}; \boldsymbol{ra} := \mathbf{false}; \boldsymbol{rs} := \boldsymbol{rs}.\boldsymbol{y} \rangle\!\rangle; \\
\quad\quad \langle\!\langle wt := 1 - l \rangle\!\rangle; & \quad\quad \mathsf{return}\, y \\
\quad\quad b[wt] := w; & \\
\quad\quad \langle\!\langle l := wt; \boldsymbol{wa} := \mathbf{false}; \boldsymbol{ws} := \boldsymbol{ws}.\boldsymbol{w} \rangle\!\rangle; &
\end{array}\right)
\end{array}
$$

Fig. 31. Harris's three-slot algorithm

$$
\begin{array}{l}
\left\{ w = W \wedge \boldsymbol{ws} = WS \star \boxed{\exists B(\neg \boldsymbol{wa} \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star b[l] = B \star b[1-l] = \_)} \right\} \\
\quad \langle\!\langle wt := c; \boldsymbol{wa} := \boldsymbol{c} \rangle\!\rangle \\
\left\{ \begin{array}{l}
\left(\begin{array}{l} wt \wedge w = W \wedge \boldsymbol{ws} = WS \star \\ \quad \boxed{\exists B(c \wedge \boldsymbol{wa} \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star b[l] = B \star b[1-l] = \_)} \end{array} \right) \vee \\
\left(\begin{array}{l} \neg wt \wedge w = W \wedge \boldsymbol{ws} = WS \star \\ \quad \boxed{\exists B(\neg c \wedge \neg \boldsymbol{wa} \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star b[l] = B \star b[1-l] = \_)} \end{array} \right)
\end{array} \right\} \Rightarrow \text{(stability)} \\
\left\{ \begin{array}{l}
\left(wt \wedge w = W \wedge \boldsymbol{ws} = WS \star \boxed{\exists B(\boldsymbol{wa} \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star b[l] = B \star b[1-l] = \_)} \right) \vee \\
\left(\begin{array}{l} \neg wt \wedge w = W \wedge \boldsymbol{ws} = WS \star \\ \quad \boxed{\exists B((\neg c \vee \neg \boldsymbol{ra}) \wedge \neg \boldsymbol{wa} \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star b[l] = B \star b[1-l] = \_)} \end{array} \right)
\end{array} \right\}
\end{array}
$$

Fig. 32. First part of the colliding three-slot writer

auxiliary $\boldsymbol{ra}$ to make explicit the sequence of the reader's actions: first it reads from b' while $\neg c$ holds, and then it sets c. From the writer's point of view it is $\neg c \vee \neg \boldsymbol{ra}$ which is stable.

The writer's actions (omitting the variable contexts and using array-indexing notation) are

$$c \wedge \neg \boldsymbol{wa} \rightsquigarrow c \wedge \boldsymbol{wa} \tag{44}$$

$$(\neg c \vee \neg \boldsymbol{ra}) \wedge b' = B \rightsquigarrow b' = B' \tag{45}$$

$$(\neg c \vee \neg \boldsymbol{ra}) \wedge \neg \boldsymbol{wa} \wedge b' = B \star b[l] = B \rightsquigarrow c \wedge \boldsymbol{wa} \wedge b' = B \star b[l] = B \tag{46}$$

$$\boldsymbol{wa} \wedge l = L \wedge b[1-l] = B \rightsquigarrow \boldsymbol{wa} \wedge l = L \wedge b[1-l] = B' \tag{47}$$

$$\boldsymbol{wa} \wedge l = L \wedge \boldsymbol{ws} = WS \star b[1-l] = B \rightsquigarrow \neg \boldsymbol{wa} \wedge l = 1 - L \wedge \boldsymbol{ws} = WS.B \star b[l] = B \tag{48}$$

– it only writes to b' when $\neg c \vee \neg \boldsymbol{ra}$, and if it changes $\neg c$ to c then $b' = b[l]$. Actions (47) and (48) are just as in the colliding two-slot (38) and (39); (44) and (46) together change c and $\boldsymbol{wa}$ in the same way as the colliding two-slot (37).

The reader's actions are

$$\neg \boldsymbol{ra} \rightsquigarrow \neg c \wedge \boldsymbol{ra} \tag{49}$$

$$\boldsymbol{ra} \wedge \boldsymbol{rs} = RS \wedge \lfloor \boldsymbol{rs}.B \rfloor \preccurlyeq \boldsymbol{ws} \rightsquigarrow c \wedge \neg \boldsymbol{ra} \wedge \boldsymbol{rs} = RS.B \tag{50}$$

As in the colliding two-slot, $\lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = b[l]$ is invariant under the actions of both writer and reader.

For $write(w)$ the first step, shown in fig. 32, begins with $\neg \boldsymbol{wa}$. The command $\langle\!\langle wt := c; \boldsymbol{wa} := \boldsymbol{c} \rangle\!\rangle$ may produce action (44), if c holds at the instant of execution, or the identity action, if $\neg c$ holds. To simplify treatment of the conditional which follows we treat stability immediately: in the first disjunct of the result c

$$\left\{ \begin{array}{l} \left(wt \wedge w = W \wedge \boldsymbol{ws} = WS \star \boxed{\exists B(\boldsymbol{wa} \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star b[l] = B \star b[1-l] = \_)} \right) \vee \\ \left(\neg wt \wedge w = W \wedge \boldsymbol{ws} = WS \star \right. \\ \quad \left. \boxed{\exists B((\neg c \vee \neg \boldsymbol{ra}) \wedge \neg \boldsymbol{wa} \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star b[l] = B \star b[1-l] = \_)} \right) \end{array} \right\}$$

if $\neg wt$ then

local w' in

$$\left\{ \begin{array}{l} \neg wt \wedge w = W \wedge \boldsymbol{ws} = WS \star \\ \quad \boxed{\exists B((\neg c \vee \neg \boldsymbol{ra}) \wedge \neg \boldsymbol{wa} \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star b[l] = B \star b[1-l] = \_)} \end{array} \right\}$$

— $\{b[l] = B$ stable under R

$w' := b[l]$

$$\left\{ \exists B \left(\begin{array}{l} \neg wt \wedge w = W \wedge \boldsymbol{ws} = WS \wedge w' = B \star \\ \quad \boxed{(\neg c \vee \neg \boldsymbol{ra}) \wedge \neg \boldsymbol{wa} \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star b[l] = B \star b[1-l] = \_} \end{array} \right) \right\}$$

— $\left\{ \begin{array}{l} (\neg c \vee \neg \boldsymbol{ra}) \wedge b' = B \rightsquigarrow b' = B' \subseteq G \\ b' = B \text{ stable under } R \end{array} \right.$

$b' := w'$

$$\left\{ \exists B \left(\begin{array}{l} \neg wt \wedge w = W \wedge \boldsymbol{ws} = WS \wedge w' = B \star \\ \quad \boxed{(\neg c \vee \neg \boldsymbol{ra}) \wedge \neg \boldsymbol{wa} \wedge b' = B \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star b[l] = B \star b[1-l] = \_} \end{array} \right) \right\}$$

$\langle\!\langle c := \text{true}; \boldsymbol{wa} := \textbf{true} \rangle\!\rangle$

$$\left\{ \begin{array}{l} \neg wt \wedge w = W \wedge \boldsymbol{ws} = WS \star \\ \quad \boxed{\exists B(c \wedge \boldsymbol{wa} \wedge b' = B \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star b[l] = B \star b[1-l] = \_)} \end{array} \right\} \Rightarrow \text{ (stability)}$$

$$\left\{ \begin{array}{l} \neg wt \wedge w = W \wedge \boldsymbol{ws} = WS \star \\ \quad \boxed{\exists B(\boldsymbol{wa} \wedge b' = B \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star b[l] = B \star b[1-l] = \_)} \end{array} \right\}$$

else

$$\left\{ wt \wedge w = W \wedge \boldsymbol{ws} = WS \star \boxed{\exists B(\boldsymbol{wa} \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star b[l] = B \star b[1-l] = \_)} \right\}$$

skip

$$\left\{ wt \wedge w = W \wedge \boldsymbol{ws} = WS \star \boxed{\exists B(\boldsymbol{wa} \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star b[l] = B \star b[1-l] = \_)} \right\}$$

fi

$$\left\{ w = W \wedge \boldsymbol{ws} = WS \star \boxed{\exists B(\boldsymbol{wa} \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star b[l] = B \star b[1-l] = \_)} \right\}$$

Fig. 33. Second part of the three-slot colliding writer

is not stable, so we forget it; in the second we have instantaneously $\neg c$; the reader may change $\neg c$ to c, but only if it simultaneously sets $\neg \boldsymbol{ra}$, and it may change c to $\neg c$, but only if it sets $\boldsymbol{ra}$; so $\neg c \vee \neg \boldsymbol{ra}$ is stable.

In the second step, shown in fig. 33, the writer checks the possibility of collision, and if so copies $b[l]$ into b'. The final steps, in fig. 34, proceed almost exactly as in the colliding two-slot proof of fig. 27.

In the reader, $c \wedge \boldsymbol{ra} \Rightarrow \lfloor \boldsymbol{rs}.b' \rfloor \preccurlyeq \boldsymbol{ws}$ is invariant, stable under the action of the writer. The first four steps, in fig. 35, proceed exactly as in the colliding two-slot proof of fig. 28, except for the additional manipulation of the shared variable $\boldsymbol{ra}$. In the final disjunction, in the first arm where we have $c \wedge \boldsymbol{ra}$, we can simplify the invariant implication, and we can forget it in the second arm where it will be irrelevant. In the rest of the proof, shown in fig. 36, we deal with the disjunction, read b' if necessary, and establish $\lfloor \boldsymbol{rs} \rfloor.y \preccurlyeq \boldsymbol{ws}$.

In this algorithm the writer sometimes makes two non-atomic assignments, and so does the reader. That makes it less efficient than Simpson's four-slot algorithm, but it is wait-free.

$$\Big\{ wt \wedge w = W \wedge \boldsymbol{ws} = WS \star \boxed{\exists B(\boldsymbol{wa} \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star b[l] = B \star b[1-l] = \_)} \Big\}$$
$$\langle\!\langle wt := 1 - l \rangle\!\rangle;$$
$$b[wt] := w;$$
$$\langle\!\langle l := wt; \boldsymbol{wa} := \mathbf{false}; \boldsymbol{ws} := \boldsymbol{ws}.\boldsymbol{w} \rangle\!\rangle$$
$$\Big\{ w = W \wedge \boldsymbol{ws} = WS.W \star \boxed{\exists B(\neg \boldsymbol{wa} \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = W \star b[1-l] = B \star b[l] = W)} \Big\}$$

Fig. 34. Final part of the three-slot colliding writer

$$\left\{ \begin{array}{l} \boldsymbol{rs} = RS \star \\ \quad \boxed{\exists B(\neg \boldsymbol{ra} \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \wedge (c \wedge \boldsymbol{ra} \Rightarrow \lfloor \boldsymbol{rs}.b' \rfloor \preccurlyeq \boldsymbol{ws}) \star b[l] = B \star b[1-l] = \_)} \end{array} \right\}$$
$$\langle\!\langle c := \text{false}; \boldsymbol{ra} := \mathbf{true} \rangle\!\rangle;$$
$$\langle\!\langle rt := l \rangle\!\rangle;$$
$$y := b[rt]$$
$$\langle\!\langle rt := c \rangle\!\rangle$$
$$\left\{ \begin{array}{l} \left(\begin{array}{l} \boldsymbol{rs} = RS \wedge rt \star \\ \quad \boxed{\exists B' \, (\, c \wedge \boldsymbol{ra} \wedge \lfloor \boldsymbol{rs}.b' \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B' \star b[l] = B' \star b[1-l] = \_)} \end{array} \right) \vee \\ \exists B \left(\begin{array}{l} \boldsymbol{rs} = RS \wedge \neg rt \wedge y = B \star \\ \quad \boxed{\exists B' \, (\, \boldsymbol{ra} \wedge \lfloor \boldsymbol{rs}.B \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B' \star b[l] = B' \star b[1-l] = \_)} \end{array} \right) \end{array} \right\}$$

Fig. 35. First steps of the three-slot colliding reader

7. Four slots

In our attempt to build an explanatory bridge to Simpson's algorithm, Harris's three-slot buffer is a diversion. Simpson's four slots are in two groups of two, and they use the two-slot techniques of fig. 19 orthogonally.

The mechanism of fig. 19(b) lets the reader indicate a slot the writer should avoid, using shared variables l to show where the writer has left the latest value and r to show where the reader is reading; the mechanism of fig. 19(a) lets the writer use the slot the reader is not using, using shared variable l alone. Neither of these mechanisms, on its own, produces coherence. But if the write to $1 - r$ / read from l technique is used to indicate *pairs* of slots and if, within each pair, the write to $1 - l$ / read from l technique shows where the writer can write and the reader can read, we do get coherence. In effect, when the reader moves the pair-goalposts then the writer shoots next in another pair; when the writer moves the within-pair-goalposts then either the reader is in the other pair, and unaffected, or the writer shoots next in the other pair. Simpson uses a two-element array of single-bit variables to indicate which data slot in each pair can be read, and which can be written to.

Our transcription of Simpson's algorithm is shown in fig. 37. The variables which control use of pairs are lp and rp; the array which controls the use of slots within a pair is li. We have had to use some ingenuity in updating $\boldsymbol{ws}$: to preserve the invariant $\boldsymbol{ws}_\Omega = b[lp][li[lp]]$ it is necessary to update $\boldsymbol{ws}$ either when $li[lp]$ is altered with $li[wtp] := wti$ or when lp is altered with $lp := wtp$. We have also had to add two auxiliaries: $\boldsymbol{wa}$ sequences the writer's activity, $\boldsymbol{wp}$ shows the pair that the writer is using so that we can reason in global assertions about its activity, and $\boldsymbol{rtb}$ facilitates the argument about freshness. The enclosed conditionals in the two final instructions of the writer are auxiliary because each makes only auxiliary assignments.

The writer's actions (omitting the variable context, using array-index notation and writing C-style $!E$ in

$$\left\{\begin{array}{l}\left(\begin{array}{l}\boldsymbol{rs} = RS \wedge rt \star \\ \boxed{\exists B'\,(\,c \wedge \boldsymbol{ra} \wedge \lfloor \boldsymbol{rs}.b' \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B' \star b[l] = B' \star b[1-l] = \_\,)}\end{array}\right) \vee \\ \exists B \left(\begin{array}{l}\boldsymbol{rs} = RS \wedge \neg rt \wedge y = B \star \\ \boxed{\exists B'\,(\,\boldsymbol{ra} \wedge \lfloor \boldsymbol{rs}.B \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B' \star b[l] = B' \star b[1-l] = \_\,)}\end{array}\right)\end{array}\right\}$$

if rt then

$$\left\{\begin{array}{l}\boldsymbol{rs} = RS \wedge rt \star \\ \boxed{\exists B', B''\,(\,c \wedge \boldsymbol{ra} \wedge b' = B'' \wedge \lfloor \boldsymbol{rs}.b' \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B' \star b[l] = B' \star b[1-l] = \_\,)}\end{array}\right\}$$

— $\{c \wedge \boldsymbol{ra} \wedge b' = B''$ stable under R

$y := b'$

$$\left\{\begin{array}{l}\boldsymbol{rs} = RS \wedge rt \wedge y = B'' \star \\ \boxed{\exists B', B''\,(\,c \wedge \boldsymbol{ra} \wedge b' = B'' \wedge \lfloor \boldsymbol{rs}.b' \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B' \star b[l] = B' \star b[1-l] = \_\,)}\end{array}\right\}$$

else

$$\left\{\exists B \left(\begin{array}{l}\boldsymbol{rs} = RS \wedge \neg rt \wedge y = B \star \\ \boxed{\exists B'\,(\,\boldsymbol{ra} \wedge \lfloor \boldsymbol{rs}.B \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B' \star b[l] = B' \star b[1-l] = \_\,)}\end{array}\right)\right\}$$

skip

$$\left\{\exists B \left(\begin{array}{l}\boldsymbol{rs} = RS \wedge \neg rt \wedge y = B \star \\ \boxed{\exists B'\,(\,\boldsymbol{ra} \wedge \lfloor \boldsymbol{rs}.B \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B' \star b[l] = B' \star b[1-l] = \_\,)}\end{array}\right)\right\}$$

fi

$$\left\{\exists B \left(\boldsymbol{rs} = RS \wedge y = B \star \boxed{\exists B'(\boldsymbol{ra} \wedge \lfloor \boldsymbol{rs}.B \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B' \star b[l] = B' \star b[1-l] = \_)}\right)\right\}$$

$\langle\!\langle c := \text{true}; \boldsymbol{ra} := \textbf{false}; \boldsymbol{rs} := \boldsymbol{rs}.\boldsymbol{y} \rangle\!\rangle$

$$\left\{\boldsymbol{rs} = RS.y \star \boxed{\exists B'(c \wedge \neg \boldsymbol{ra} \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B' \star b[l] = B' \star b[1-l] = \_)}\right\}$$

Fig. 36. Final steps of the three-slot colliding reader

$$\begin{array}{l}\textsf{local}\, b[2][2] = ((null, null), (null, null)), li[2] = (0,0), lp = 0, r = 0, \boldsymbol{ws} = null., \boldsymbol{rs} = ., \neg \boldsymbol{wa}, \boldsymbol{wp}\ \textsf{in} \\ \left(\begin{array}{l|l} \ldots & \ldots \\ write(w) = & read(\,) = \\ \quad \textsf{local}\, wtp, wti; & \quad \textsf{local}\, y, rtp, rti, \boldsymbol{rtb}; \\ \quad \langle\!\langle wtp := 1 - rp; \boldsymbol{wp} := \boldsymbol{wtp}; \boldsymbol{wa} := \boldsymbol{1} \rangle\!\rangle; & \quad \langle\!\langle rtp := lp; \boldsymbol{rtb} := \boldsymbol{ws_\Omega} \rangle\!\rangle; \\ \quad \langle\!\langle wti := 1 - li[wtp] \rangle\!\rangle; & \quad \langle\!\langle rp := rtp \rangle\!\rangle; \\ \quad b[wtp][wti] := w; & \quad \langle\!\langle rti := li[rtp] \rangle\!\rangle; \\ \quad \langle\!\langle \left(\begin{array}{l} li[wtp] := wti; \\ \textbf{if } \boldsymbol{wtp} = \boldsymbol{lp} \textbf{ then } \boldsymbol{wa} := \boldsymbol{0}; \boldsymbol{ws} := \boldsymbol{ws}.\boldsymbol{w} \\ \textbf{else } \boldsymbol{wa} := \boldsymbol{2}\ \textbf{fi} \end{array}\right) \rangle\!\rangle; & \quad y := b[rtp][rti]; \\ \quad \langle\!\langle \left(\begin{array}{l} lp := wtp; \\ \textbf{if } \boldsymbol{wa} = \boldsymbol{2} \textbf{ then } \boldsymbol{wa} := \boldsymbol{0}; \boldsymbol{ws} := \boldsymbol{ws}.\boldsymbol{w} \\ \textbf{else skip fi} \end{array}\right) \rangle\!\rangle & \quad \langle\!\langle \boldsymbol{rs} := \boldsymbol{rs}.\boldsymbol{y} \rangle\!\rangle \\ & \quad \textsf{return}\, y \end{array}\right) \end{array}$$

Fig. 37. Simpson's four-slot buffer, adapted from [Sim90]

place of $1 - E$ to compress the assertions) are

$$\boldsymbol{wa} = 0 \wedge rp = P \rightsquigarrow \boldsymbol{wa} = 1 \wedge rp = P \wedge \boldsymbol{wp} = !P \tag{51}$$

$$\boldsymbol{wa} = 1 \star b[\boldsymbol{wp}][!li[\boldsymbol{wp}]] = V \rightsquigarrow \boldsymbol{wa} = 1 \wedge b[\boldsymbol{wp}][!li[\boldsymbol{wp}]] = V' \tag{52}$$

$$\left(\begin{array}{l}\boldsymbol{wa} = 1 \wedge lp = L = \boldsymbol{wp} \wedge \boldsymbol{ws} = WS \star \\ li[lp] = I \star b[lp][!I] = B\end{array}\right) \rightsquigarrow \left(\begin{array}{l}\boldsymbol{wa} = 0 \wedge lp = L = \boldsymbol{wp} \wedge \boldsymbol{ws} = WS.B \star \\ li[lp] = !I \star b[lp][!I] = B\end{array}\right) \tag{53}$$

$$\boldsymbol{wa} = 1 \wedge lp = L = !\boldsymbol{wp} = rp \star li[\boldsymbol{wp}] = I \rightsquigarrow \boldsymbol{wa} = 2 \wedge lp = L = !\boldsymbol{wp} = rp \star li[\boldsymbol{wp}] = !I \tag{54}$$

$$\left(\begin{array}{l}\boldsymbol{wa} = 2 \wedge lp = L = !\boldsymbol{wp} = rp \wedge \boldsymbol{ws} = WS \star \\ li[\boldsymbol{wp}] = I \star b[\boldsymbol{wp}][I] = B\end{array}\right) \rightsquigarrow \left(\begin{array}{l}\boldsymbol{wa} = 0 \wedge lp = !L = \boldsymbol{wp} = !rp \wedge \boldsymbol{ws} = WS.B \star \\ li[lp] = I \star b[lp][I] = B\end{array}\right) \tag{55}$$

The reader's actions are

$$rp = !L = !lp \rightsquigarrow rp = lp = L \tag{56}$$

$$\boldsymbol{rs} = RS \wedge \boldsymbol{ws} = WS \wedge \lfloor RS.B \rfloor \preccurlyeq WS \star b[rp][I] = B \rightsquigarrow \boldsymbol{rs} = RS.B \wedge \boldsymbol{ws} = WS \star b[rp][I] = B \tag{57}$$

Invariant is $\lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = b[lp][li[lp]]$. Note that when the writer changes lp, which only happens in action (55), it converts $lp = rp$ into $lp = !rp$ – never the other way round – and that when the reader changes rp in action (56) it changes $lp = !rp$ into $lp = rp$. These are important and surprising properties of the algorithm.

We reason about the writer in fig. 38. First, $\langle\!\langle wtp := 1 - rp; \ldots \rangle\!\rangle$ produces action (51) and we have, instantaneously, $wtp = !rp$. The value of rp is not in general stable, but $lp = rp$ implies that rp is stable: action (56) then will not disturb it and action (57) does not alter lp or rp. Thus we stabilise by adding the implication $lp = !P \Rightarrow rp = !P$, and it is this which eventually ensures that the writer's final step only changes lp if it makes $lp \neq rp$.

Second, $\langle\!\langle wti := 1 - li[wtp] \rangle\!\rangle$ produces the identity action. The contents of li are stable under the reader's actions, but we show two conjoined views of the array to show both the value I which points to the latest value in the buffer and the value I' which was extracted into wti.

Third, the non-atomic write $b[wtp][wti] := w$ is unproblematic, but to show its effect it is necessary to split the shared assertion according to whether the writer is in the lp pair ($lp = P$) or not. That separation is then used when dealing with the next two commands, which do not update $\boldsymbol{ws}$ until $b[lp][li[lp]] = W$.

Fourth, $\langle\!\langle li[wtp] := wti; \ldots \rangle\!\rangle$ produces either action (53), inverting $li[lp]$ and appending to $\boldsymbol{ws}$ when $lp = rp$, or (54), inverting $li[lp]$ only when $lp \neq rp$.

Fifth and finally, the effect of $\langle\!\langle lp := wtp; \ldots \rangle\!\rangle$ is either the identity action, when $lp = P$, or it is (55), inverting lp and appending to $\boldsymbol{ws}$ when $lp = rp = !P$.

The reader is shown in fig. 39. First, $\langle\!\langle rtp := lp \rangle\!\rangle$ generates the identity action. We have instantaneously $rtp = lp$, but the value of lp is not stable in general. It is stable, however, if $lp \neq rp$, so $rp = !P \Rightarrow lp = P$ is stable. Also $\boldsymbol{wa} = 1 \wedge \boldsymbol{wp} = P \Rightarrow lp = P$ is stable: it is true initially, because $lp = P$; when $\boldsymbol{wp}$ is set by (51) we have either $rp = P$ in which case $\boldsymbol{wp} \neq P$, or we have $rp = !P$, in which case $\boldsymbol{wp} = P$, but because $rp = !P \Rightarrow lp = P$, also $lp = P$. Since (53) only ever writes to $b[\boldsymbol{wp}][!li[\boldsymbol{wp}]]$ the value in $b[\boldsymbol{wp}][li[\boldsymbol{wp}]]$ is stably a member of the written sequence; since $\boldsymbol{wa} = 1 \wedge \boldsymbol{wp} = P \Rightarrow lp = P$ the writer cannot execute (54) when $\boldsymbol{wp} = P$; therefore when $li[\boldsymbol{wp}]$ changes it is because of (53) or (55) and in either case the value in $b[\boldsymbol{wp}][li[\boldsymbol{wp}]]$ is $\boldsymbol{ws}_\Omega$. So we have, stably, that the value in $b[P][li[P]]$ is always a valid extension of $\boldsymbol{rs}$ at least as fresh as the value it instantaneously held when the reader began. Unlike earlier examples the reader does not choose a value in its first step, and the referent of $b[P][li[P]]$ could change arbitrarily often before the reader selects a slot in its third step: but this argument shows that it does not matter if it does change, because freshness is guaranteed. The conjunction describing the slot and index arrays simply deals with the fact that $lp = P$ is not stable.

Second, $\langle\!\langle rp := rtp \rangle\!\rangle$ generates either the identity action when $rp = P$ or (56) when $rp = !P = !lp$. Nothing else changes: in particular we are still sure that $b[P][li[P]]$ is a valid, and fresh, extension to $\boldsymbol{rs}$.

Third, $\langle\!\langle rti := li[rtp] \rangle\!\rangle$ selects a slot to read. Instantaneously this is in the $li[P]$ position, but if $\boldsymbol{wp} = P$ this can be altered by (53). The slot itself, however, cannot be altered: (52) could only write to the $li[!P]$ slot, and once $li[P]$ has been altered, (51) can only set $\boldsymbol{wp} = !P$.

Fourth, now that we have a stable slot to read from, $y := b[rtp][rti]$ non-atomically reads from that slot, and finally $\langle\!\langle \boldsymbol{rs} := \boldsymbol{rs}.\boldsymbol{y} \rangle\!\rangle$ updates $\boldsymbol{rs}$ and produces action (57).

8. What has been achieved?

We set out to construct a sequence of verifications of single-slot buffer algorithms which would lead towards a proof of Simpson's algorithm. We did construct that sequence, and along the way we found a proof (the first, so far as we are aware) of Harris's algorithm. We were able to introduce and use the techniques of RGSep in our proofs, and the final proof encompassed safety (values transmitted are coherent) as well as functional correctness (the sequence read is a stuttering subsequence of the sequence written and the values retrieved by the reader are freshly written at the instant retrieval begins). We have helped to demonstrate the utility of RGSep, a new approach to the verification of shared-memory concurrent algorithms. We have worked throughout on the code itself rather than an abstraction or a state-machine model.

$$\left\{\begin{array}{l} w = W \wedge \boldsymbol{ws} = WS \star \\ \boxed{\exists B, I \left(\begin{array}{l} \boldsymbol{wa} = 0 \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star \\ \quad li[lp] = I \star li[!lp] = \_ \star b[lp][I] = B \star b[lp][!I] = \_ \star b[!lp] \mapsto \_, \_ \end{array}\right)} \end{array}\right\}$$

$\langle\!\langle wtp := 1 - rp; \boldsymbol{wp} := \boldsymbol{wtp}; \boldsymbol{wa} := \boldsymbol{1} \rangle\!\rangle$

$$\left\{\exists P \left(\begin{array}{l} w = W \wedge wtp = P \wedge \boldsymbol{ws} = WS \star \\ \boxed{\exists B, I \left(\begin{array}{l} \boldsymbol{wa} = 1 \wedge \boldsymbol{wp} = P \wedge rp = !P \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star \\ \quad li[lp] = I \star li[!lp] = \_ \star b[lp][I] = B \star b[lp][!I] = \_ \star b[!lp] \mapsto \_, \_ \end{array}\right)} \end{array}\right)\right\} \Rightarrow \text{(stability)}$$

$$\left\{\exists P \left(\begin{array}{l} w = W \wedge wtp = P \wedge \boldsymbol{ws} = WS \star \\ \boxed{\exists B, I \left(\begin{array}{l} \boldsymbol{wa} = 1 \wedge \boldsymbol{wp} = P \wedge (lp = !P \Rightarrow rp = !P) \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star \\ \quad li[lp] = I \star li[!lp] = \_ \star b[lp][I] = B \star b[lp][!I] = \_ \star b[!lp] \mapsto \_, \_ \end{array}\right)} \end{array}\right)\right\}$$

$\langle\!\langle wti := 1 - li[wtp] \rangle\!\rangle$

$$\left\{\exists P, I' \left(\begin{array}{l} w = W \wedge wtp = P \wedge wti = I' \wedge \boldsymbol{ws} = WS \star \\ \boxed{\exists B, I \left(\begin{array}{l} \boldsymbol{wa} = 1 \wedge \boldsymbol{wp} = P \wedge (lp = !P \Rightarrow rp = !P) \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star \\ \quad ((li[P] = !I' \star li[!P] = \_) \wedge (li[lp] = I \star li[!lp] = \_)) \star \\ \quad b[lp][I] = B \star b[lp][!I] = \_ \star b[!lp] \mapsto \_, \_ \end{array}\right)} \end{array}\right)\right\}$$

$$\text{—} \left\{\begin{array}{l} b[\boldsymbol{wp}][!li[\boldsymbol{wp}]] = V \text{ stable under } R \\ \boldsymbol{wa} = 1 \wedge b[\boldsymbol{wp}][!li[\boldsymbol{wp}]] = V \rightsquigarrow \boldsymbol{wa} = 1 \wedge b[\boldsymbol{wp}][!li[\boldsymbol{wp}]] = V' \subseteq G \end{array}\right.$$

$b[wtp][wti] := w$

$$\left\{\exists P, I' \left(\begin{array}{l} w = W \wedge wtp = P \wedge wti = I' \wedge \boldsymbol{ws} = WS \star \\ \boxed{\exists B, I \left(\begin{array}{l} \boldsymbol{wa} = 1 \wedge \boldsymbol{wp} = P \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star \\ \quad ((li[P] = !I' \star li[!P] = \_) \wedge (li[lp] = I \star li[!lp] = I')) \star \\ \quad \left(\begin{array}{l} (lp = P \wedge b[lp][I] = B \star b[lp][!I] = W \star b[!lp] \mapsto \_, \_) \vee \\ \left(\begin{array}{l} lp = rp = !P \star b[lp][I] = B \star b[lp][!I] = \_ \star \\ \qquad b[!lp][I'] = \_ \star b[!lp][!I'] = W \end{array}\right) \end{array}\right) \end{array}\right)} \end{array}\right)\right\}$$

$\langle\!\langle li[wtp] := wti; \textsf{if } \boldsymbol{wtp} = \boldsymbol{lp} \textsf{ then } \boldsymbol{wa} := \boldsymbol{0}; \boldsymbol{ws} := \boldsymbol{ws.w} \textsf{ else } \boldsymbol{wa} := \boldsymbol{2} \textsf{ fi} \rangle\!\rangle$;

$$\left\{\exists P, I' \left(\begin{array}{l} w = W \wedge wtp = P \wedge wti = I' \star \\ \boxed{\exists B, I \left(\begin{array}{l} \boldsymbol{wp} = P \star li[P] = I' \star li[!P] = \_ \star li[lp] = I \star li[!lp] = I' \star \\ \quad \left(\begin{array}{l} \left(\begin{array}{l} \boldsymbol{wa} = 0 \wedge lp = P \wedge \boldsymbol{ws} = WS.W \wedge \\ \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = W \wedge \\ b[lp][!I] = B \star b[lp][I] = W \star b[!lp] \mapsto \_, \_ \end{array}\right) \vee \\ \left(\begin{array}{l} \boldsymbol{wa} = 2 \wedge lp = rp = !P \wedge \boldsymbol{ws} = WS \wedge \\ \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \wedge \\ b[lp][I] = B \star b[lp][!I] = \_ \star b[!lp][I'] = W \star b[!lp][!I'] = \_ \end{array}\right) \end{array}\right) \end{array}\right)} \end{array}\right)\right\}$$

$\langle\!\langle lp := wtp; \textsf{if } \boldsymbol{wa} = \boldsymbol{2} \textsf{ then } \boldsymbol{wa} := \boldsymbol{0}; \boldsymbol{ws} := \boldsymbol{ws.w} \textsf{ else skip fi} \rangle\!\rangle$

$$\left\{\exists P, I' \left(\begin{array}{l} w = W \wedge wtp = P \wedge wti = I' \wedge \boldsymbol{ws} = WS.W \star \\ \boxed{\exists B, I \left(\begin{array}{l} \boldsymbol{wa} = 0 \wedge \boldsymbol{wp} = lp = P \wedge \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = W \star \\ \quad li[lp] = I \star li[!lp] = \_ \star b[lp][I] = W \star b[lp][!I] = \_ \star b[!lp] \mapsto \_, \_ \end{array}\right)} \end{array}\right)\right\}$$

Fig. 38. Verification of the 4-slot writer

But in three respects we have failed to achieve all that we had hoped. First, and most importantly to us, our final proof is by no means as simple as we had hoped. Though in a sense it might stand as an explanation of how the algorithm works, it does not convince as an explanation of why it works: it is too much a formalisation of a programmer's understanding. Although we have managed to capture some of the regularities of the algorithm's operation in stable implications, we have had too often to appeal to explicitly sequential arguments: after the reader has done this, the writer can only do that, followed by the other.

The simplest specification of Simpson's algorithm is the magically-atomic algorithm of fig. 1. A refinement

$$\left\{ \begin{array}{l} \boldsymbol{rs} = RS \star \\ \boxed{\exists B, I(\lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star li[lp] = I \star li[!lp] = \_ \star b[lp][I] = B \star b[lp][!I] = \_ \star b[!lp] \mapsto \_, \_)} \end{array} \right\}$$

$\langle\!\langle rtp := lp; \boldsymbol{rtb} := \boldsymbol{ws}_\Omega \rangle\!\rangle$

$$\left\{ \exists B, P \left(\begin{array}{l} \boldsymbol{rs} = RS \wedge rtp = P \wedge \boldsymbol{rtb} = B \star \\ \boxed{\exists I \left(\begin{array}{l} lp = P \wedge \lfloor \boldsymbol{rs}.B \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star \\ li[P] = I \star li[!P] = \_ \star b[P][I] = B \star b[P][!I] = \_ \star b[!P] \mapsto \_, \_ \end{array} \right)} \end{array} \right) \right\} \Rightarrow \text{(stability)}$$

$$\left\{ \exists B, P \left(\begin{array}{l} \boldsymbol{rs} = RS \wedge rtp = P \wedge \boldsymbol{rtb} = B \star \\ \boxed{\exists B', B'' \left(\begin{array}{l} (rp = !P \Rightarrow lp = P) \wedge (\boldsymbol{wa} = 1 \wedge \boldsymbol{wp} = P \Rightarrow lp = P) \wedge \\ \lfloor \boldsymbol{rs}.B.B' \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B'' \star \\ \left(\begin{array}{l} \exists I(li[P] = I \star li[!P] = \_ \star b[P][I] = B' \star b[P][!I] = \_ \star b[!P] \mapsto \_, \_) \wedge \\ \exists I'(li[lp] = I' \star li[!lp] = \_ \star b[lp][I'] = B'' \star b[lp][!I'] = \_ \star b[!lp] \mapsto \_, \_) \end{array} \right) \end{array} \right)} \end{array} \right) \right\}$$

$\langle\!\langle rp := rtp \rangle\!\rangle$

$$\left\{ \exists B, P \left(\begin{array}{l} \boldsymbol{rs} = RS \wedge rtp = P \wedge \boldsymbol{rtb} = B \star \\ \boxed{\exists B', B'' \left(\begin{array}{l} (rp = P \wedge (\boldsymbol{wa} = 1 \wedge \boldsymbol{wp} = P \Rightarrow lp = P) \wedge \lfloor \boldsymbol{rs}.B.B' \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B'' \star \\ \left(\begin{array}{l} \exists I(li[P] = I \star li[!P] = \_ \star b[P][I] = B' \star b[P][!I] = \_ \star b[!P] \mapsto \_, \_) \wedge \\ \exists I'(li[lp] = I' \star li[!lp] = \_ \star b[lp][I'] = B'' \star b[lp][!I'] = \_ \star b[!lp] \mapsto \_, \_) \end{array} \right) \end{array} \right)} \end{array} \right) \right\}$$

$\langle\!\langle rti := li[rtp] \rangle\!\rangle$

$$\left\{ \begin{array}{l} \exists B, P, I \\ \left(\begin{array}{l} \boldsymbol{rs} = RS \wedge rtp = P \wedge \boldsymbol{rtb} = B \wedge rti = I \star \\ \boxed{\exists B', B'' \left(\begin{array}{l} (rp = P \wedge (\boldsymbol{wa} = 1 \wedge \boldsymbol{wp} = P \Rightarrow lp = P) \wedge \\ \lfloor \boldsymbol{rs}.B.B' \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B'' \star \\ \left(\begin{array}{l} \left(\begin{array}{l} li[P] = I \star li[!P] = \_ \star \\ b[P][I] = B' \star b[P][!I] = \_ \star b[!P] \mapsto \_, \_ \end{array} \right) \wedge \\ \exists I' \left(\begin{array}{l} li[lp] = I' \star li[!lp] = \_ \star \\ b[lp][I'] = B'' \star b[lp][!I'] = \_ \star b[!lp] \mapsto \_, \_ \end{array} \right) \end{array} \right) \end{array} \right)} \end{array} \right) \end{array} \right\} \Rightarrow \text{(stability)}$$

$$\left\{ \exists B, P, I \left(\begin{array}{l} \boldsymbol{rs} = RS \wedge rtp = P \wedge \boldsymbol{rtb} = B \wedge rti = I \star \\ \boxed{\exists B', B'' \left(\begin{array}{l} (rp = P \wedge (\boldsymbol{wa} = 1 \wedge \boldsymbol{wp} = P \Rightarrow lp = P) \wedge \\ \lfloor \boldsymbol{rs}.B.B' \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B'' \star \\ \left(\begin{array}{l} (li \mapsto \_, \_, \_\_ \star b[P][I] = B' \star b[P][!I] = \_ \star b[!P] \mapsto \_, \_) \wedge \\ \exists I' \left(\begin{array}{l} li[lp] = I' \star li[!lp] = \_ \star \\ b[lp][I'] = B'' \star b[lp][!I'] = \_ \star b[!lp] \mapsto \_, \_ \end{array} \right) \end{array} \right) \end{array} \right)} \end{array} \right) \right\}$$

— $\{b[P][I] = B'$ stable under R

$y := b[rtp][rti]$

$$\left\{ \exists B' \left(\begin{array}{l} y = B' \wedge \boldsymbol{rs} = RS \star \\ \boxed{\exists B'', I' \left(\begin{array}{l} \lfloor \boldsymbol{rs}.B' \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B'' \star \\ li[lp] = I' \star li[!lp] = \_ \star b[lp][I'] = B'' \star b[lp][!I'] = \_ \star b[!lp] \mapsto \_, \_ \end{array} \right)} \end{array} \right) \right\}$$

$\langle\!\langle \boldsymbol{rs} := \boldsymbol{rs}.\boldsymbol{y} \rangle\!\rangle$

$$\left\{ \exists B' \left(\begin{array}{l} y = B' \wedge \boldsymbol{rs} = RS.B' \star \\ \boxed{\exists B, I \left(\begin{array}{l} \lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws} \wedge \boldsymbol{ws}_\Omega = B \star \\ li[lp] = I \star li[!lp] = \_ \star b[lp][I] = B \star b[lp][!I'] = \_ \star b[!lp] \mapsto \_, \_ \end{array} \right)} \end{array} \right) \right\}$$

Fig. 39. Verification of the 4-slot reader

one to the other would make a more convincing argument and a more revealing explanation than ours. At the time we began to prepare this paper that was beyond us, and we could not envisage a refinement route to Harris's algorithm.

Second, we had hoped to plot a route towards an automatic proof, one where we give a straightforward specification and leave the internal reasoning entirely to the machine. Our use of RGSep enabled us to deal with examples that did not depend on semaphores and the like, and we hoped that it might remove the need to invent auxiliary variables to deal with sequencing, a significant barrier against automation. That was not possible in the two more intricate examples. We have made some progress elsewhere [AB09] in dealing with sequencing in an automatic proof, but the intricacies required to deal with freshness in Simpson's algorithm will require integration of work on automatic liveness checking [SB06].

Though we cannot yet make our proofs automatic, and though we have not mechanised the proofs reported here even though it would be possible to do so, hand proofs are by no means useless. Our aim is exploration, explanation and understanding. The search for formal proofs by hand is, in our view, a necessary precursor to formal mechanical and/or automatic proof. Our formal specification $\lfloor \boldsymbol{rs} \rfloor \preccurlyeq \boldsymbol{ws}$ is simple and readable, as opposed to state-machines of assertions of earlier work. Our model is the code itself, rather than a state monad. We need to check nine entailments and two stabilisations: the hard part of our proof is finding the right auxiliary state that then allows for easy entailment checks, and this militates against automation rather than mechanisation. We do not believe that the (relative) simplicity of our proof is a result of non-mechanisation. Existing mechanisations of Hoare-style program proof allow direct verification of code. All our entailment and stability checks are purely syntactic, and once the (considerable) logical machinery of RGSep is mechanised in a theorem prover, the proof itself follows directly.

Third, we have run up against the barrier of weak memory. Computing practice always runs ahead of computer science. Hardware always has more capability than we have been able to describe formally in any particular programming language, and programmers have always exploited that capability. Our proofs have relied on an assumption that single-bit writes and reads 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 and cache. The processor can do several hundred operations in the time it takes to read a single data item from the main store, but the cache is much faster. Data read from memory is held in the cache and can be re-accessed in a few processor cycles; data written can be stored in the 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 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 flushed.

If there is more than one processor – if Simpson's reader and writer are running on separate processors, for example – then things become more complicated. The reader's $\langle\!\langle r := rtp \rangle\!\rangle$, which is crucial for signalling the fact that the reader is working in the *rtp* pair, will not necessarily have an effect immediately. That means that we cannot conclude in the shared-memory postcondition that $r = P$ and say that the writer will not write again in the P pair. Or worse, we cannot say that $b[r][I] = B$ is stable, because the writer's last update of that location might not yet have been flushed from its buffer.

Processors are nowadays designed with so-called *weak memory models* which cover its behaviour when updating or reading from memory and describe how it interacts with the cache and buffer. There is supposed to be a formal description of the memory model to which the activity of the processor conforms, but researchers inform us (Peter Sewell, private communication) that those descriptions often do not correspond to processor behaviour. The consequence for the program prover is that semicolon has ceased to be a sequencing operator – writes and reads can occur out of sequence – but we do not and at present cannot have a formalism which deals with the problem.

Researchers who devise concurrent algorithms cope with this situation by trial and error. Special 'fence' or 'barrier' instructions can be used to force the processor to wait until a write, or until all writes terminate. Simpson's algorithm, for example, would work if we inserted a memory barrier instruction after each write to the signalling variables l and r, if by so doing we forced the processor to wait until all its pending writes had been completed, and if the read caches of both processors remained consistent with each other. This may or may not be the memory model of one popular processor: for the time being experiment, and lots and lots of soak testing, seems to be the only way to be sure.

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 have to be remade and we have to remake our proofs correspondingly. Algorithms like Simpson's will perhaps turn out to be unsuited to the modern hardware world, because they need too many memory fence instructions, stopping the processor too often. Or perhaps not, and it does not matter to the prover anyway: our aim is to extend our techniques so that we can deal with the algorithms that are actually used. We still cannot do so as well as we need to, and that must be remedied.

Acknowledgements

We have been helped enormously in our work by tutorials in the use of RGSep from Viktor Vafeiadis and Matthew Parkinson, and from discussions with Peter O'Hearn and others in the East London Massive that still meets occasionally to thrash out separation-logic dilemmas. We acknowledge the helpful criticism of the referees, which has helped us make considerable improvements to the original version of this paper.

References

[AB09] Hasan Amjad and Richard Bornat. Towards Automatic Stability Analysis for Rely-Guarantee Proofs. In Neil D. Jones and Markus Müller-Olm, editors, *VMCAI 10*, number 5403 in Lecture Notes in Computer Science, pages 14–28. Springer, January 2009. 36

[Bar03] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. 3

[BCOP05] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew J. Parkinson. Permission accounting in separation logic. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 259–270, New York, NY, USA, January 2005. ACM Press. 2, 6

[BCY05] Richard Bornat, Cristiano Calcagno, and Hongseok Yang. Variables as resource in Separation Logic. In *Proceedings of MFPS XXI*. Elsevier ENTCS, May 2005. 2, 5

[Bor00] Richard Bornat. Proving Pointer Programs in Hoare Logic. In R. C. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction, 5th International Conference*, LNCS, pages 102–126. Springer, 2000. 3

[Bri02] P. Brinch Hansen, editor. *The Origin of Concurrent Programming.* Springer-Verlag, 2002. 37

[Bro04] Stephen D. Brookes. A Semantics for Concurrent Separation Logic. In *CONCUR'04: 15th International Conference on Concurrency Theory*, volume 3170 of *Lecture Notes in Computer Science*, pages 16–34, London, August 2004. Springer. Extended version is [Bro07]. 6

[Bro07] Stephen D. Brookes. A Semantics for Concurrent Separation Logic. *Theoretical Computer Science*, 375(1-3):227–270, 2007. 5, 6, 37

[Bur72] R.M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972. 3

[CHP71] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Commun. ACM*, 14(10):667–668, 1971. 6

[Dij65] Edsger W. Dijkstra. Cooperating Sequential Processes. Technical Report EWD-123, Technical University, Eindhoven, 1965. Reprinted in [Gen68] and [Bri02]. 2, 5, 6, 11

[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. 15

[Gen68] F. Genuys, editor. *Programming Languages.* Academic Press, 1968. 37

[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. 3

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

[HLM03] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 522, Washington, DC, USA, 2003. IEEE Computer Society. 28

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. 3

[Hoa72] C. A. R. Hoare. Towards a theory of parallel programming. In Hoare and Perrott, editors, *Operating System Techniques*, pages 61–71. Academic Press, 1972. Reprinted in [Bri02]. 5

[Hoa73] C. A. R. Hoare. Hints on programming language design. Technical Report CS-TR-73-403, Stanford University, Computer Science Dept., 1973. Keynote address to ACM SIGPLAN conference. Reprinted in [Jon89], pp 193–216. 3

[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. 3

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, 2008. 3

[Jon83] Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983. 2, 6

[Jon89] Cliff B. Jones, editor. *Essays in Computing Science*. Prentice/Hall International, 1989. 37

[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. 3

[Mic04] Maged M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004. 2

[MN05] Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. *Inf. Comput.*, 199(1-2):200–227, 2005. 3

[Mor82] J.M. Morris. A general axiom of assignment. Assignment and linked data structure. A proof of the Schorr-Waite algorithm. In M. Broy and G. Schmidt, editors, *Theoretical Foundations of Programming Methodology (Proceedings of the 1981 Marktoberdorf Summer School)*, pages 25–51. Reidel, 1982. 3

[OG76a] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 19:319–340, 1976. 6

[OG76b] S. Owicki and D. Gries. Verifying Properties of Parallel Programs: an axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976. 6

[O'H04] Peter O'Hearn. Resources, Concurrency and Local Reasoning. In *CONCUR'04: 15th International Conference on Concurrency Theory*, volume 3170 of *Lecture Notes in Computer Science*, pages 49–67, London, August 2004. Springer. Extended version is [O'H07]. 5, 6, 10, 14

[O'H07] Peter O'Hearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1-3):271–307, 2007. 5, 6, 10, 13, 38

[OP99] Peter O'Hearn and D. Pym. The Logic of Bunched Implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999. 3

[ORY01] Peter O'Hearn, John C. Reynolds, and Hongseok Yang. Local Reasoning about Programs that Alter Data Structures. In L. Fribourg, editor, *CSL 2001*, pages 1–19. Springer-Verlag, 2001. LNCS 2142. 2, 3

[Owi75] Susan Speer Owicki. *Axiomatic proof techniques for parallel programs.* PhD thesis, Cornell, 1975. Technical report TR75-251. 6

[PBC06] Matthew J. Parkinson, Richard Bornat, and Cristiano Calcagno. Variables as resource in Hoare logics. In *Proceedings of LICS*, pages 137–146. IEEE, 2006. 2, 5

[PBO07] Matthew J. Parkinson, Richard Bornat, and Peter O'Hearn. Modular verification of a non-blocking stack. *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 297–302, 2007. 3

[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. 3

[Pym99] D. Pym. On Bunched Predicate Logic. In *14h Symposium on Logic in Computer Science*, pages 183–192, Trento, Italy, July 1999. IEEE Computer Society Press. 3

[Rey00] John C. Reynolds. Intuitionistic Reasoning about Shared Mutable Data Structure. In Jim Davies, Bill Roscoe, and Jim Woodcock, editors, *Millennial Perspectives in Computer Science*, pages 303–321. Palgrave, 2000. 3

[Rey02] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society. 2, 3

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

[SB06] Viktor Schuppan and Armin Biere. Liveness checking as safety checking for infinite state spaces. *Electr. Notes Theor. Comput. Sci.*, 149(1):79–96, 2006. 36

[Sim90] H.R. Simpson. Four-Slot Fully Asynchronous Communication Mechanism. *IEE Proceedings*, 137(1):17–30, January 1990. 2, 21, 24, 28, 32

[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. 21, 24

[Sim97b] H.R. Simpson. Role Model Analysis of an Asynchronous Communication Mechanism. *IEE Proceedings - Computers and Digital Techniques*, 144(4):232–240, July 1997. 2

[Vaf07] Viktor Vafeiadis. *Modular fine-grained concurrency verification.* PhD thesis, University of Cambridge, 2007. 7, 17

[VP06] Viktor Vafeiadis and Matthew J. Parkinson. A Marriage of Rely/Guarantee and Separation Logic. http://www.cl.cam.ac.uk/ mjp41/RGSep.pdf, November 2006. 7

[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. 2

[YO02] Hongseok Yang and Peter W. O'Hearn. A Semantic Basis for Local Reasoning. In *FoSSaCS '02: Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures*, number 2303 in Lecture Notes in Computer Science, pages 402–416, London, UK, 2002. Springer-Verlag. 4