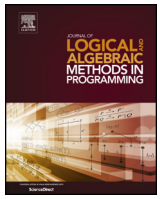
Journal of Logical and Algebraic Methods in Programming 85 (2016) 972–984

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp



Possible values: Exploring a concept for concurrency

Cliff B. Jones [a,*], Ian J. Hayes [b]

[a] School of Computing Science, Newcastle University, UK
[b] School of Information Technology and Electrical Engineering, The University of Queensland, Australia

ARTICLE INFO

Article history:
Received 15 June 2015
Received in revised form 10 September 2015
Accepted 8 January 2016
Available online 13 January 2016

Keywords:
Concurrent programming
Rely/guarantee conditions
Possible values

ABSTRACT

An important issue in concurrency is interference. This issue manifests itself in both shared-variable and communication-based concurrency — this paper focuses on the former case where interference is caused by the environment of a process changing the values of shared variables. Rely/guarantee approaches have been shown to be useful in specifying and reasoning compositionally about concurrent programs. This paper explores the use of a "possible values" notation for reasoning about variables whose values can be changed multiple times by interference. Apart from the value of this concept in providing clear specifications, it offers a principled way of avoiding the need for some auxiliary (or ghost) variables whose unwise use can destroy compositionality.

1. Introduction

High on the list of issues that make the design of concurrent programs difficult to get right is 'interference'. Reproducing a situation that exhibited a 'bug' can be frustrating; attempting to reason informally about all possible interleavings of interference can be exasperating; and designing formal approaches to the verification of concurrent programs is challenging.

Recording post conditions for sequential programs applies the only real tool that we have: abstraction is achieved by winnowing out what is inessential in the relationship between the initial and final states of a computation. Post conditions record the required relationship without fixing an algorithm to bring about the transformation; furthermore, they record required properties only of those variables which the environment will use. The rely/guarantee approach (see Section 1.1) uses abstraction in the same way to provide specifications of concurrent software components that are more abstract than their implementations: for any component, rely conditions are relations that record interference that the component must tolerate and guarantee conditions document the interference that the environment of the component must accept.

This paper explores a concept that fits well with rely/guarantee reasoning but probably has wider applicability. In relational post conditions, it is necessary to be able to refer to the initial value x and final value x' of a variable x (e.g. $x \leq x' \leq x + 9$). If however it is necessary to record something as simple as the fact that a local variable x captures one of the values of a shared variable y, it is inadequate to write $x' = y \vee x' = y'$ in the case where y might be changed many times by the environment. Enter 'possible values': the suggested notation is that $\widehat{y}$ denotes the set of values which variable y contains during the execution of the operation in whose specification $\widehat{y}$ is written. So, (assuming the access to read the value of y is atomic):

$$post\text{-}Op : x' \in \widehat{y}$$

is satisfied by a simple assignment of y to x.

* Corresponding author.
E-mail address: cliff.jones@ncl.ac.uk (C.B. Jones).

http://dx.doi.org/10.1016/j.jlamp.2016.01.002

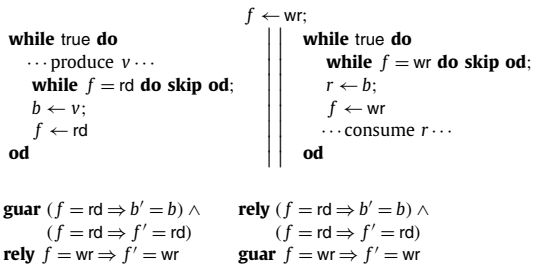


Fig. 1. A one-place buffer.

1.1. Rely/Guarantee thinking

Before going into more detail on the possible values notation (see Section 2), a brief overview of background work is offered. The specifications given in Section 3 are written in the notation of VDM [12,15]. It is unlikely that they will present difficulties even to readers unfamiliar with that specific notation because similar ideas for sequential programs are present in Z [6], B [1], Event-B [2], and TLA [23]. The basic idea is of state-based specifications with operations (or events) transforming the state and being specified by something like pre and post conditions. Pre conditions are predicates over states that indicate what can be assumed about states in which an operation can be initiated. Post conditions are relations over initial and final states that specify the required relations between the initial and final values of state components. Good sequential specifications eschew any details of implementation algorithms: they do not specify anything about intermediate states; in fact an implementation might use a state with more components. At first sight, it might appear surprising that there is not a precise functional requirement on the final state but using non-determinism in specifications turns out to be an extremely useful way of postponing design decisions.

The use of abstract objects in specifications is a crucial tool for larger applications. Moreover, datatype invariants can make specifications clearer: restricting types by predicates simplifies pre/post conditions and also offers a way for the specifier to record the intention of a specification. Another useful aspect of VDM is the ability to define more tightly the 'frame' of an operation by recording whether access to state components is for (only) reading or for both reading and writing.[1]

The basic rely/guarantee [13,14] idea[2] is simple: interference is documented and proof rules are given which support reasoning about interference in concurrent threads. Just as in sequential specifications, the role of a state is central to recording rely/guarantee specifications. For concurrency, it is accepted that the environment of a process can change values in the state during execution of an operation.[3] Such changes are however assumed to be constrained by a rely condition. In order to reason about the combined effect of operations, the interference that a process can inflict on its environment is also recorded; this is done in a guarantee condition. Both rely and guarantee conditions are, for obvious reasons, relations over states. In the original form – and after many experiments – both conditions are reflexive and transitive covering the possibility of zero or many steps. Such relations often indicate monotonic evolution of variables.

It is useful to compare the roles of rely and guarantee conditions with the better known pre/post conditions. Pre conditions are essentially an invitation to the designer of a specified component to ignore some starting states; in the same way, the developer can ignore the possibility that interference will make state changes that do not satisfy the rely condition. In neither case should a developer include code to test these assumptions; there is an implicit requirement to prove that the component is only used in an appropriate context. In contrast, post conditions and guarantee conditions are obligations on the running code that the developer has to create; these conditions record properties on which the deployer can depend.

The simplest form of relation that could be used in rely or guarantee conditions is to state that the value of a variable remains unchanged (e.g. $b' = b$). Such unconditional constraints are normally better handled by marking an operation (or part thereof) as having only read access. There is however an important way to combine 'monotonic' changes to flags with assertions about variables remaining unchanged. Consider a simple one-place buffer in which a producer process places a value in a buffer variable b from which a consumer process extracts values. Testing and setting flag f in Fig. 1 ensures that the producer and consumer alternate their access to b. During its read phase, the consumer needs to rely on the fact that the value of b cannot change but this is too strong as a rely condition for the whole of the consumer process — the producer process could never insert anything into the buffer if it were required to achieve a guarantee condition of $b' = b$. But the consumer process can instead rely on $f = \mathsf{rd} \Rightarrow b' = b$, which in turn is easy for the producer to guarantee. The 'monotonic' behaviour of the flags means that the producer has also to guarantee that $f = \mathsf{rd} \Rightarrow f' = \mathsf{rd}$ and the consumer must guarantee $f = \mathsf{wr} \Rightarrow f' = \mathsf{wr}$. This example shows one way in which rely/guarantee conditions can be used

[1] Most of the literature on rely/guarantee conditions is limited to normal (or 'scoped') variables; [21] shows how 'heap' variables can be viewed as representations of more abstract states.

[2] The literature on rely/guarantee approaches continues to expand; see [9,11] for further references. For a reader who is completely unfamiliar with rely/guarantee concepts, a useful brief presentation can be found in [16].

[3] Notice that there is an essential difference here from 'actions' [5] or 'events' [2] which view execution of a guarded action as atomic.

to reason about race-free programs. It also illustrates a technique that is used in Section 3 to locate what is going on in the environment without adding auxiliary variables. The example tackled in Section 3 is however much more challenging than this simple one-place buffer.

1.2. Law for mutual strengthening of guarantee and rely

As part of the example in Section 3, a new facet of rely/guarantee refinement is needed: it allows mutual strengthening of both rely and guarantee conditions for a portion of one process. The approach is a contribution to rely/guarantee refinement and it makes it possible to avoid introducing additional auxiliary variables (see Section 4.3) in order to handle the example in Section 3.

In the standard approach to rely/guarantee refinement, when two parallel processes are introduced each has an associated rely/guarantee pair and there is an obligation to show that the guarantee of each implies the rely of the other. Normally the one rely/guarantee pair suffices to handle the refinement of a process but for the example in Section 3 that is not sufficient.

In the standard theory, rely/guarantee pairs are often mutually dependent: for the two-process case, a process P maintaining its guarantee may be dependent on its environment (process Q) maintaining the rely of P (by Q maintaining its guarantee) and vice versa. For example, P may guarantee to maintain $x \geq 0$ provided it can rely on its environment maintaining $x \geq 0$. The guarantee, g, of a process has to hold for every atomic program step it makes and hence g has to be weak enough to be maintained by every step. However, for a subpart S of P, all the atomic steps of S may imply a stronger guarantee gs. As P forms the environment of process Q, while P is executing subpart S, Q may assume a stronger rely condition of gs and as a consequence of this its own guarantee may be strengthened from r to rs, which in turn allows process P to assume a stronger rely condition rs, but only while it is executing subpart S. Note that while only a subpart S of P is of concern, the whole of Q has to be considered for the strengthening of its rely and guarantee.

In order to establish the strengthened rely/guarantee pair for the duration of S, the state when P enters S may need to satisfy an initial condition j. For the example in Section 3 a special case of the above reasoning applies in which the guarantee of P is strengthened to state that P does not modify any shared variables. In this case one needs to show that process Q maintains the stronger rely rs from any initial state satisfying j provided Q suffers no inference from P.

1.3. Connection to data abstraction/reification

It is important to appreciate how rely relations abstract from the detail of the actual environmental interference of an operation. Obviously, the most detailed information about an environment is the actual state changes it makes. But designing to such concrete detail would create a component that is not robust to change. Just as post conditions deliberately omit implementation details of a specified operation, it is useful to strive for an abstract documentation of interference. It is clear that relations cannot record certain sorts of information but, if they are adequate for a given task, their use will yield a more compositional development than the detail of the environment.

The extended example in Section 3 shows the importance of linking rely/guarantee ideas with data abstraction and reification. Specification using abstract mathematical objects and the process of stepwise introduction of more concrete (i.e. closer to hardware) objects is well established for sequential programs and for significant applications is often more telling than the abstraction that comes from post conditions — see, for example, [15]. In addition to layering design decisions, careful use of abstract objects in the development of concurrent programs offers other advantages. In particular, developments can appear to allow data races at an abstract level that are removed by careful choice of a concrete representation — this is discussed in [17]. One reason that this is interesting is Peter O'Hearn's suggested dichotomy in [24] that separation logic is appropriate for reasoning about race avoidance whilst rely/guarantee methods fit 'racy' programs. The distinction between abstract and concrete data races is perfectly illustrated in Section 3 but the example is not easy to summarise. A simpler example is searching an array to find the lowest index of an element that satisfies a predicate P by means of two parallel processes that search the elements with, respectively, even and odd indices (for a full development of this example, see [9]). If a single variable t were used to record the least index of an element that satisfies P, it would be necessary to have locks in the two processes to avoid a data race on t. A neat way to avoid the 'write/write' race is to represent t by the minimum of two variables, et and ot that record the least value of, respectively, even and odd indices where the array element satisfies P. The 'write/write' race, which is useful in an abstract description of the design, is reduced to a 'read/write' race because the actual code for each process updates only one of the variables although it reads the other variable in its loop test (and on the completion of both processes t can be retrieved as $min(et, ot)$).

The citations above relate to the original form of rely/guarantee reasoning in which the (potentially) four conditions are combined. More recent work has shown how separate rely and/or guarantee constraints can be wrapped around any command including conventional refinement calculus style specifications. The presentation in [9,11] of rely/guarantee thinking makes algebraic properties clearer.

1.4. Plan of this paper

This paper provides evidence of the usefulness of the possible values concept. Section 2 presents a notation for the concept while Section 3 is an extended example using the concept and notation. Section 2.2 outlines how a semantic model

can be provided and looks at the form of laws that would fit the newer presentation of rely/guarantee reasoning [9,11]. The current authors recognise that this paper represents the start of an exploration — some avenues to be investigated are mentioned in Section 4.

2. Possible values

It is argued above that the confessed expressive weakness of rely/guarantee specifications serves the purpose of preserving some form of compositionality in the design of concurrent programs. However, if notations can be found that increase expressive power, they should be evaluated both for expressiveness and tractability. The simple case mentioned above of using one or more possible values terms in a post condition is considered first and issues about extension are deferred to Section 4.

2.1. Possible values of variables

If an operation only has read access to a shared variable y and x is a local variable of the process, then:

$$post\text{-}Op : x' \in \widehat{y} \tag{1}$$

requires that the final value of the variable x should contain one of the values that the environment places in the variable y — this includes the (initial) value of y at the time Op began execution. So $\widehat{y}$ denotes a set of values whose elements have the type of y.

Notice that the post condition above is 'stable' in the sense that the environment might change the value of y after Op accesses the variable and the post condition is still true. In contrast, it would be unwise to write a post condition that contained $x' \notin \widehat{y}$ because this would not be stable and it would appear to require that every possible change that the environment makes to the value of y is observed. (In some cases, it would be possible to establish such a result under a suitable rely condition; but some form of (local) datatype invariant should also be considered in such cases.)

So, for the straightforward case, the post condition (1) can be established by the atomic assignment $x \leftarrow y$. As is reported in Section 3.2, an instance of this simple case was the inspiration for the possible values notation. There are however several vectors of extension. If the process in which the $\widehat{y}$ term is written also has write access to the variable y, it is necessary to take a position on whether both environment assignments to y and those of the component itself are reflected in $\widehat{y}$; the view of the current authors is that $\widehat{y}$ contains all values of y that could be observed by the process.

2.2. Semantics and laws

It is not difficult to see how a formal meaning can be given to the simple form of the possible values notation in a semantics such as that in [9]: basically, that portion of the sequence of states that corresponds to the execution of an operation is distinguished so as to identify the first and last states in order to give a semantics to post conditions. It is only necessary to consider all of the states in that portion and to extract the set of values of the relevant variable.

Another interesting semantic issue concerns locking. In fact, the possible values notation forces consideration of a number of facets of 'atomicity'. Locking may be used to ensure mutually exclusive access to a set of variables. A process may lock a resource protecting a set of variables. While it owns the lock, it may make multiple changes to the variables protected by the lock, however, any other processes accessing the protected variables cannot observe any of the intermediate states of the protected variables. Hence a process in the scope of a resource with a set of protected variables can only observe the initial and final states of a protected block within another process. Throughout the body of a protected block a process can rely on the protected variables being stable. Furthermore, any guarantee involving just the protected variables has to hold only between the initial and final states of the protected block.

Just as the semantics for the straightforward use of possible values terms in a post condition poses no difficulties in terms of the underlying traces, a rather simple law suffices to reason about the notation. Here, it is convenient to switch to the refinement calculus style of [9,11] in which the specification statement $x : [q]$ establishes the postcondition q and modifies only x, and the command c in a rely context of r is written $\mathbf{rely}\ r \cdot c$. Assuming a read of y is atomic, the following law holds.

$$\mathbf{rely}\ (x' = x) \cdot x : [x' \in \widehat{y}] \quad \sqsubseteq \quad x \leftarrow y \tag{2}$$

The rely condition $x' = x$ is required to ensure that the environment doesn't change x after the assignment is made. For example, x may be a local variable or, as below in Section 3.3, annotated **owns wr** x.

2.3. Possible values of expressions

For the set of possible values of an expression, $\widehat{e}$, one needs to consider the corresponding set of states of the execution and form the set of values of e, each evaluated in one of those states. Importantly, all values of program variables used in e are sampled in a single state for each evaluation. For example, for the specification

$$x : [x' \in \widehat{y + y}] \tag{3}$$

both occurrences of y are sampled in the same state and hence the resultant value is always even (assuming the variables are integer valued). Note that there is a subtle difference between (3), which samples y once, and

$$x : [\exists v, w \cdot v \in \widehat{y} \wedge w \in \widehat{y} \wedge x' = v + w]$$

which samples y twice so that the values of v and w may differ.

Replacing y in the law (2) with an expression e introduces the complication that each variable reference in the evaluation of e in the assignment could be accessed in a different state. Note that if e has multiple references to a single variable y, each reference could be accessed in a different state. However, if e has only a single reference to a variable y and all other variables in e are stable, any evaluation of e is equivalent to evaluating it in the state in which y is accessed and the law is valid. Let S be a set of variables such that the free variables of e are contained in $S \cup \{y\}$ and e has only a single reference to y and accesses to y are atomic, then

$$\textbf{rely}\ (x' = x \wedge (\bigwedge z \in S \cdot z' = z)) \cdot x : [x' \in \widehat{e}] \ \sqsubseteq\ x \leftarrow e\ . \tag{4}$$

If e is of the form $d(f)$ for a mapping d and expression f, stability is required on the program variables in f but stability is not required for the whole of d, just $d(f)$, because the other elements of d have no effect on the expressions value.

If the expression e contains multiple references to a variable x, saving x in a local variable t and then evaluating e in terms of t ensures that the value used for x is from a single state. The following refinement law ensures x is sampled once. It is assumed that r and t are local variables (and hence the environment cannot change them) and that r and t do not occur free in e.

$$t, r : [r' \in \widehat{e[x/v]}] \ \sqsubseteq\ \langle t \leftarrow x \rangle\ ;\ r : [r' \in \widehat{e[t/v]}] \tag{5}$$

For this to be valid one needs to rely on the environment maintaining $\widehat{e[t/v]} \subseteq \widehat{e[x/v]}$, for the duration of the command. This holds provided the rely condition

$$x' \neq t \wedge t' = t \Rightarrow e[t/v]' = e[t/v]$$

is maintained by the environment, where $e[t/v]'$ stands for $e[t/v]$ evaluated in the after state, i.e. e' is e with every program variable y in e replaced by y'.

Law (5) can be justified as follows. The atomic statement $\langle t \leftarrow x \rangle$ establishes $e[t/v] = e[x/v]$. An environment step that has a final state in which $x' = t$ establishes $e[t/v]' = e[x/v]'$ otherwise the environment establishes $e[t/v]' = e[t/v] = e[x/v]$.

As an example consider the case in which the expression e is $d(v)$. Applying (5) gives

$$t, r : [r' \in \widehat{d(x)}] \ \sqsubseteq\ \langle t \leftarrow x \rangle\ ;\ r : [r' \in \widehat{d(t)}] \tag{6}$$

provided its environment ensures the condition: $x' \neq t \Rightarrow d'(t) = d(t)$. Immediately after the atomic assignment to t,

$$d(t) = d(x) \in \widehat{d(x)}\ . \tag{7}$$

If the environment makes a step that does not change x, (7) is maintained because $d'(t') = d'(x')$ but if the environment changes x so that it no longer equals t one can no longer rely on $d'(t')$ being the same as $d'(x')$. However, if one can rely on $d(t)$ being stable and because $d(t) = d(x)$ and $d'(t') = d(t)$, one can still deduce $d'(t') = d(x)$.

3. Asynchronous Communication Mechanisms

An Asynchronous Communication Mechanism (ACM) logically provides a one-place buffer between a single writer and a single reader (see Fig. 2). This sounds trivial but the snag is in the adjective: ACMs are asynchronous in the sense that neither the reader nor the writer should ever be held up by locks.[4] Unless the value being communicated via the buffer is small enough to be read and written atomically, it should be obvious that one slot is not enough to realise the buffer; a little thought shows that a buffer representation with two slots is also inadequate; the topic of how many slots are required is returned to in Section 3.4. In [28], Hugo Simpson proposed a 'four-slot' algorithm to implement an ACM for which, while the code is short, extremely subtle reasoning is required for its justification.

3.1. ACM requirements

The requirement is to communicate the "most recent" value from a single producer to a single consumer via a shared buffer. More precisely, it must satisfy the following.

[4] This contrasts with the simple one-place buffer in Section 1 where the code would 'busy wait' on the value of a flag to control alternation between the producer and consumer.

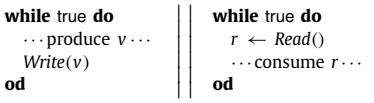


Fig. 2. Code to clarify reader/writer structure.

- It is assumed that there is only a single reader and a single writer but the reader and writer processes operate completely asynchronously
- A write puts a new value in the buffer
- A read gets a completely written value from the buffer
- The value read is at least as fresh as the last completely written value when the read started – this implies that, for two consecutive reads, the value read by the second read will be at least as fresh as that read by the first
- Reads and writes must not block (no locks)
- Reads and writes of values can't be assumed to be atomic (i.e. a single value may be larger than the atomic changes made by the hardware)
- The only thing Simpson assumes to be atomic is the setting of single bits (and they are actually realised by wires)
- The buffer is initialised with a data value (so there is always something to read)
- The buffer is shared by the reading and writing processes alone (i.e. no third process can modify the buffer)

In the terminology of Lamport [22] this can be summarised as implementing a single-reader wait-free atomic register in terms of atomic Boolean control registers.

3.2. Approaches to specifying ACM

There is an interesting range of approaches as to how the requirements that are listed above can be expressed in a formal specification. Without surveying all of them, it fits the theme of this paper to review two strands of publications[5]: one motivated by (Concurrent) Separation Logic [24,25] and the other by rely/guarantee methods. Surveying the latter also pinpoints the origin of the possible value notation.

Richard Bornat is an expert on separation logic so it is interesting to look at how he has formalised the specification and development of Simpson's 'four slot' algorithm. In [3], separation logic is certainly used but it is interesting to see that the paper also uses rely/guarantee concepts. In contrast, [4] makes no real use of separation logic and the specification uses the concept of linearisability [10]. The reason that this history is enlightening is that the essence of Simpson's algorithm is the exchange of 'ownership' of the four slots between the reader and writer processes. This is done precisely to ensure (data) race freedom so one would anticipate that separation logic would be in its element. There is, in fact, one paper that uses separation logic for precisely this form of argument; unfortunately [31] does not include an argument that the reader always gets the 'freshest' value and a recent private correspondence with one of the authors indicates that they have not extended their work to cover this essential property.

It is only fair to make an equally critical assessment of two papers [19,20] that use rely/guarantee ideas. In the development recorded in [19],[6] it is necessary to assert that the value of one variable (lw) is assigned to another variable (cr); this assertion was recorded as:

$$cr' = lw \vee cr' = lw' \ .$$

This plausible attempt says that the final value of cr is either the initial or final value of lw. Unfortunately, during the operation being specified, the value of lw could potentially be changed more than once. This observation was precisely the stimulus that led to the invention of the notation for possible values. In addition to various improvements and clarifications in the development, the journal version [20] resolves the problem by using

$$cr' \in \widehat{lw} \ .$$

Rushby [27] noted a similar issue in model checking Simpson's algorithm: a version checking for just the before or after values fails in the case of multiple writes overlapping a single read. To handle this in the model checking context, Rushby restricts the sequence of data values written so that they are strictly increasing in value, and then checks that the sequence of values read is nondecreasing, which he concludes is necessary but may not be sufficient. He concedes that this is a limitation of the expressiveness of the model checking specification language (which does not have the (unbounded) expressive power of the possible values notation).

[5] Other approaches include [2,8].

[6] The variable names in the Jones/Pierce papers are *hold-r*/*fresh-w*; for the reader's convenience, these have been changed in the extracts in the current paper to match the names used here (cr/lw).

There is however a deeper objection to both of the Jones/Pierce specifications of ACMs. In both cases, the most abstract specification uses a variable (*data-w*) that contains the entire history of values written by the write process. This is in spite of the fact that a read operation cannot access values in the sequence earlier than the last value added before the read began. This sort of redundancy is deprecated in [15, Sect. 9.3] as using a 'biased' representation: the state contains values that have no influence on subsequent operations. Where there is no bias in the representation underlying a specification, a homomorphism (retrieve function) relates a representation back to the abstraction; in the case of a biased representation, a relation between the abstraction and the representation is used to argue that the operations on the latter fit those on the former. In situations where it is necessary to express non-determinism in a specification that can be removed in the design process, biased specifications are sometimes unavoidable — but, where there is an alternative, unbiased specifications should normally be preferred because they make it easier to see the range of possible implementations. One further surprising fact about the specifications in [19,20] is that, even at the most abstract level, the specifications of both *Read* and *Write* are each split into two sub-operations which are joined by sequential composition. Although the semantics of such a specification are clear, it means that the task of convincing users that their requirements have been adequately captured involves a rather algorithmic discussion.

Having been self-critical of these specifications, there is one important positive point that needs preserving in the approach below: the issue of data-race freedom is handled in [20] at the level of an abstract intermediate representation. This is an important general point: rely/guarantee conditions can be used to record interference on an abstraction where the final code is certainly not 'racy'.

3.3. Specification using possible values

In contrast to the above attempts, a top-level specification using 'possible values' notation appears to be much more natural and perspicuous. The abstract specification uses a state with just a single value buffer b of type *Value*. The use of this intuitively simple state is only made possible by employing the possible value notation in the post condition of *Read*, where $\widehat{b}$ stands for the set of possible values of b during the execution of *Read*. The *Read* operation is described as returning a value (r) so the post condition is simply $r' \in \widehat{b}$. This means that a single read operation can return the value of the write most recently completed at the time the read begins or of any write that executes an assignment to b during the execution of the read operation. Notice that there is no danger of a subsequent read operation obtaining an older value than the current read because the reference point for the possible values of the newer read is the start of its execution.

As in [20], the specification can be made clearer by annotating whether the external state variables accessed by an operation can be only read (**rd**) or both read and written (**wr**).

Thus, the specification of *Read* can be given simply as:

Read() r : *Value*

ext rd b : *Value*

post $r' \in \widehat{b}$

When generating proof obligations, the **ext rd** is equivalent to a guarantee condition $b' = b$.

The specification of the *Write* operation is interesting. If the parameter to *Write* is v, one would expect the post condition to be $b' = v$ — and this is certainly required. In addition, it is necessary to rule out the possibility that *Write*(v) puts some spurious value(s) into b that might be accessed by *Read* before the *Write*(v) corrects its wayward behaviour and achieves its post condition. This can be expressed in a guarantee condition $b' \neq b \Rightarrow b' = v$. Extending (again, as in [20]) the **ext** annotation to mark write ownership yields a specification:

Write(v : *Value*)

ext owns wr b : *Value*

guar $b' \neq b \Rightarrow b' = v$

post $b' = v$

Here, the proof obligation expansion of **ext owns wr** is a rely condition $b' = b$, which matches the implicit guarantee of *Read* courtesy of its **ext rd** annotation.

The role of the guarantee of *Write* here is to provide an intuitive specification; the more standard use is to show that processes can co-exist and this usage occurs in the development below. The guarantee of *Write* ensures that only valid values are observable in the buffer (by *Read*). It is an important part of the specification of *Write* but note that there is no corresponding rely condition in *Read*. Firstly, there is the technical issue that v is local to *Write* and hence cannot be referred to in (the rely of) *Read*. Secondly, several *Write* operations might take place during a single *Read* and hence there may be multiple changes to the buffer during a *Read*, even though each *Write* only changes the buffer (at most) once. In fact, the possible multiple changes of the buffer during a *Read* motivates the use of $\widehat{b}$ in its post condition. It is worth

observing that $\widehat{b}$ is applied to an abstract variable b — the development that follows employs a representation of b that is by no means obvious.

The guarantee of *Write* requires that the observable effect of the operation takes place in a single atomic step and the use of the possible values notation in the post condition of *Read* ensures that the observable effect of *Read* also takes place in a single atomic step.

The initial value of b is assumed to contain a valid *Value* so that it is possible for a *Read* operation to precede the first *Write*.

Thus far, the possible values concept – that was devised in order to document an intermediate design – has here been shown to offer a short and clear overall specification of ACM behaviour. Freshness comes from the possible values notation and the effect of it being relative to the start of each *Read* operation. The implementation has to find a way of achieving the atomic change behaviour of b in the abstraction without such an atomicity assumption.

3.4. Understanding Simpson's representation

The challenge of presenting a specification that makes sense to potential users is addressed in Section 3.3. A development using a single data reification step to a version of Simpson's code is presented in Section 3.5 — that development makes interesting further use of the possible values concept and is thus presented in some detail. The current section attempts to provide an intuition of the 'four-slot' data structure. The operations corresponding to *Write* and *Read* of Section 3.3 are here named $Write_i$ and $Read_i$.

The importance of data abstraction and reification are commented in Section 1.3. Rather than jump immediately to Simpson's decision to use exactly four slots to represent the abstract variable b, a useful intermediate refinement step uses a data structure that contains an abstract map of an indexed set of 'slots' $X \xrightarrow{m} Value$. Here, this part of the state is named dw. There is also a data type invariant that requires that the (potentially partial) map has a value in every slot: $\mathbf{dom}\, dw = X$.[7]

As in [20], the index set X is deliberately left unspecified at this stage. $Write_i$ is decomposed into a three parts:

- $Write\text{-}ch_i$ chooses an index ($\in X$) that is safe to use
- $Write\text{-}upd_i(v)$ updates the map dw at the chosen index to v
- $Write\text{-}com_i$ commits the index by exposing it to $Read_i$

$Read_i$ is split into two parts:

- $Read\text{-}sel_i$ selects the most recently written index and stores that index in a local variable
- $r \leftarrow Read\text{-}acc_i$ accesses the indexed slot

$Write_i$ must inform $Read_i$ of the index of the slot which has been (most recently) written. In addition, $Read_i$ must have a way of alerting $Write_i$ to the index of the slot that is claimed for reading. Remember that the reader and writer processes are in no way synchronised and the implementation is designed to allow (multiple) reads to occur during a write or multiple writes to overlap with a single read.

It should be clear that the potential number of slots (the cardinality of the set X) must be at least three because the writer has to select a member of X that is neither the most recently written nor one which the reader might access (these could be the same but are not necessarily so). It is possible to build a 'three slot' implementation *providing* there is an atomic way of communicating index values between *Read* and *Write*.

It is tempting to make $Read_i$ reserve a single element of X to $Write_i$ but this does not actually provide an abstraction of Simpson's code. What that code effectively does is to reserve more than one slot. This is shown here as pr being a set of indexes.

The intermediate state is thus:

$$
\begin{array}{lll}
\Sigma_i :: & dw : X \xrightarrow{m} Value & \text{– space for values} \\
& lw : X & \text{– index of the last committed write} \\
& cw : X & \text{– index claimed by the writer} \\
& cr : X & \text{– index claimed by the reader} \\
& pr : X\text{-}\mathbf{set} & \text{– potential elements of } X \text{ that the reader might use}
\end{array}
$$

It is an interesting observation that none of the variables can be modified by both operations. The final letter of each variable name records which process, reader or writer, can write to that variable (e.g. lw can only be modified by $Write_i$).

It is not difficult to see the lines of the data reification required here: the retrieve function is $b = dw(lw)$. The initial state must, of course, satisfy the invariant; the initial value in the buffer must be $dw(lw)$ and there must be some arbitrary value in every slot to ensure that $\mathbf{dom}\, dw = X$.

[7] Note that, in the concurrent context, the data type invariant must hold for every step, not just initially and at the end of each operation.

$$
\begin{array}{lll}
\Sigma_f :: & dsw : P \times S \xrightarrow{m} Value & \text{– two pairs of two data slots each} \\
& sw : P \xrightarrow{m} S & \text{– } sw(p) \text{ is the last written slot for pair } p \\
& lpw : P & \text{– last written pair} \\
& cpw : P & \text{– current write pair} \\
& cpr : P & \text{– current read pair} \\
& csr : S & \text{– current read slot}
\end{array}
$$

Fig. 3. The final state Σ_f.

$$
\begin{array}{ll}
Read_f()r : Value & Write_f(v : Value) \\
\quad \textbf{var}\, t \in P; & \quad \langle cpw \leftarrow \neg cpr \rangle; \\
\quad \langle t \leftarrow lpw \rangle; & \quad dsw(cpw, \neg sw(cpw)) \leftarrow v; \\
\quad \langle cpr \leftarrow t \rangle; & \quad \langle sw(cpw) \leftarrow \neg sw(cpw) \rangle; \\
\quad \langle csr \leftarrow sw(cpr) \rangle; & \quad \langle lpw \leftarrow cpw \rangle \\
\quad r \leftarrow dsw(cpr, csr) &
\end{array}
$$

Fig. 4. Code for Simpson's algorithm.

It is interesting to note that the issue of (data) race freedom on the slots is worked out at this level of abstraction with rely/guarantee conditions. This can be contrasted with Peter O'Hearn's view in [24] that separation logic is the tool of choice for reasoning about race freedom and rely/guarantee reasoning is for 'racy' programs. The decisive point appears to be that, here, race freedom is established on a data structure that is more abstract than the final representation.

Although the observation is made above that three slots would be adequate to avoid clashing,[8] the genius of the representation proposed by Hugo Simpson is that – if four slots are used – communication can be reduced to using single bits; furthermore, in a physical implementation, these bits can be realised as wires connecting the $Read_f$ and $Write_f$ processes running on separate processors. Simpson describes the algorithm in terms of choosing 'pairs' and 'slots'. As in [20], this intuition is followed by using two sets P and S each of which has two possible values. However, here, toggling between the two values is achieved by a "¬" operator. Although both sets P and S can be implemented as Booleans, the temptation to use Booleans is resisted at this stage because separating the types P and S provides useful information as to whether each index variable refers to a pair or a slot (and has the potential to flag incorrect use as a type error).

The final representation (Σ_f) is given in Fig. 3. This is related to Σ_i by a retrieve function where:

- dw is directly modelled by dsw with the set X reified to a (P, S) pair
- cw is represented by $(cpw, \neg sw(cpw))$
- cr is represented by (cpr, csr)
- lw is represented by $(lpw, sw(lpw))$
- pr is represented by $\{(cpr, sl) \mid sl \in S\}$

3.5. One step argument

This section presents a single-step data refinement from the top level specification using possible values to Simpson's algorithm. Although the approach to refining the code from the specification is new, the end code comes from Simpson's insights and motivates the approach. The final state representation is as in Fig. 3 and the relationship between the abstract buffer b and this representation state is

$$b = dsw(lpw, sw(lpw)) \ .$$

The code for Simpson's algorithm is given in Fig. 4. First note that the $Write_f$ operation has the following guarantee[9]:

$$\forall i, j \cdot (i, j) \neq (cpw, \neg sw(cpw)) \Rightarrow dsw'(i, j) = dsw(i, j) \tag{8}$$

and that when the write operation is not active, the writing process does not modify any of the variables in the representation.

The specification of the $Read_f$ operation after mapping through the representation relation and extending the frame with an appropriate subset of the representation variables is

$$t, cpr, csr, r : [r' \in \widehat{dsw(lpw, sw(lpw))}] \tag{9}$$

[8] In fact, [4] also considers a three slot implementation.

[9] Although ACMs are much more complicated than the one-place buffer, the idea mentioned in Section 1.1 of locating where a key value is unchanged without adding auxiliary variables is evident here.

The first refinement step uses law (5) from Section 2.2 with lpw corresponding to x and $dsw(v, sw(v))$ corresponding to e. In fact the law needs to be extended to accommodate extra variables in the frame but that is straightforward.

$$(9) \sqsubseteq \langle t \leftarrow lpw \rangle; \tag{10}$$

$$cpr, csr, r : [r' \in \widehat{dsw(t, sw(t))}] \tag{11}$$

For this one needs to rely on

$$lpw' \neq t \Rightarrow dsw'(t, sw'(t)) = dsw(t, sw(t)) \ . \tag{12}$$

Note that $sw(t)$ is only changed by $Write_f$ if $t = cwp$ but the code also guarantees that $t = cpw \Rightarrow t = lpw$ and hence $sw(t)$ can be changed only if $lpw' = t$ and hence (12) holds. If $sw(t)$ does not change, (12) is guaranteed by $Write_f$ by (8) because $(t, sw(t)) \neq (cpw, \neg sw(cpw))$ because either $t \neq cpw$ or if $t = cpw$ then $sw(t) = sw(cpw) \neq \neg sw(cpw)$. This use of the slots vector sw in Simpson's algorithm is one of the smart parts of how it works.

The second refinement step uses law (5) once more to refine (11).

$$(11) \sqsubseteq \langle cpr \leftarrow t \rangle; \tag{13}$$

$$csr, r : [r' \in \widehat{dsw(cpr, sw(cpr))}] \tag{14}$$

provided one can rely on $t' \neq cpr \Rightarrow dsw'(cpr, sw'(cpr)) = dsw(cpr, sw(cpr))$ which holds trivially as $t = cpr$ is invariant over (14).

The third refinement step again uses law (5) to refine (14).

$$(14) \sqsubseteq \langle csr \leftarrow sw(cpr) \rangle; \tag{15}$$

$$r : [r' \in \widehat{dsw(cpr, csr)}] \tag{16}$$

provided one can rely on

$$sw'(cpr) \neq csr \Rightarrow dsw'(cpr, csr) = dsw(cpr, csr) \tag{17}$$

being maintained by $Write_f$ for the duration of (16). Here (17) can be strengthened to

$$dsw'(cpr, csr) = dsw(cpr, csr) \tag{18}$$

and this can be shown to be maintained by $Write_f$ using the approach outlined in Section 1.2. For the duration of (16), $Read_f$ strengthens its guarantee to state that it does not modify any shared variables (r is local to $Read_f$). Because (15) establishes

$$(cpr, csr) \neq (cpw, \neg sw(cpw)) \tag{19}$$

it is sufficient to show that $Write_f$ maintains (18) from initial states satisfying (19) provided there is no interference on its shared variables. If $Write_f$ is executing its write phase from any state satisfying (19), it guarantees (18) because the slot being written is not (cpr, csr). Once $Write_f$ finishes its write phase (or if it is not initially in its write phase) it does not modify dsw at all (and hence maintains (18)) until after it executes $cpw \leftarrow \neg cpr$ which re-establishes (19) for the next write phase.

The final refinement step uses law (4) to refine (16).

$$(16) \sqsubseteq r \leftarrow dsw(cpr, csr)$$

This is valid provided $dsw(cpr, csr)$ is stable, which follows from the argument given above for the previous refinement step.

A pleasing aspect of the above refinement is that, having started from a specification (9) using the possible values concept (which allows for non-determinism in the value read), the refinement steps have maintained the use of possible values (and hence the non-determinism) until the last step, when it is clear which slot is being read (and that the slot is stable).

Using the approach of locally strengthening a guarantee – and hence indirectly strengthening a rely – (see Section 1.2) obviates the need to introduce and reason about auxiliary variables. However, a development using auxiliary Boolean variables *reading* and *writing* is also possible, where *reading* is true if and only if the *Read* process is actually reading from *dsw* and *writing* is true if and only if the *Write* process is actually writing *dsw*. With these auxiliary variables the important invariant is,

$$reading \wedge writing \Rightarrow (cpr, csr) \neq (cpw, \neg sw(cpw))$$

which ensures that the *Read* and *Write* processes are not simultaneously using the same slot. This represents the weakest invariant to ensure correct operation of the algorithm.

Because the specification of $Write_f$ does not make use of possible values notation its refinement is not presented in detail here. An important property of $Write_f$ is that during its writing phase the slot being written differs from any slot that could be read concurrently, which has been covered in the refinement of $Read_f$. The other aspect of $Write_f$ worth noting is that its guarantee in the abstract specification requires the buffer b to be updated to v atomically. Recalling that b is represented by $dsw(lpw, sw(lpw))$, that guarantee is achieved by (non-atomically) assigning to slot $(cpw, \neg sw(cpw))$, which can never correspond to b. The switch of $(lpw, sw(lpw))$ to $(cpw, \neg sw(cpw))$ is then achieved either by the assignment $sw(cpw) \leftarrow \neg sw(cpw)$ if lpw already equals cpw, or by the following assignment $lpw \leftarrow cpw$ if they differed initially.

Finally note that all of the atomic assignments in Fig. 4 are now in a form in which they can be implemented by the corresponding non-atomic assignment, assuming each read and write of a shared variable other than dsw is atomic. This assumption is in line with the requirements because the shared flags can be implemented as single bits (or, indeed, realised as wires).

4. Conclusions and further work

The concept of possible values arose in an attempt to provide a clear design rationale of code which is delicate in the sense that slight changes destroy its correctness. A seemingly simple and intuitive notational idea contributed to the description of a layered development. The proposal was clearly motivated by a need in a practical application. The next bonus came in the link to the non-deterministic state ideas: this connection is set out in [7]. The current paper contains the first publication of the specification given in Section 3.3 and the simplicity of the overall specification comes as strong encouragement for the concept and notation of possible values. This is further reinforced by the development of Simpson's algorithm in Section 3.5 which retains the use of the possible values notation and utilises laws taking advantage of the possible values notation.

This closing section points to further avenues that appear to have potential but certainly require more work. As with the steps to date, the motivation for the decisions should come from practical examples.

4.1. Further applications

It can perhaps be mentioned that the possible values notation appears to have some potential for recording arguments about brain-teaser puzzles. At the March 2015 meeting of IFIP WG 2.3 in Istanbul, Michael Jackson posed a hide-and-seek puzzle which is apparently described in several contexts. Here, a mole is what must be located. There are five holes in a line; the mole moves each night to an adjacent hole; the seeker can only check one hole per night and must devise a strategy that eventually locates the mole whose non-deterministic nocturnal movements are only constrained at either end of the line of holes. This paper doesn't spoil the reader's fun by providing an answer; it only mentions that one of the authors recorded the argument for termination using the possible values notation.

Sadly, most of the examples (see [26,29,30]) using 'weak memory' (a.k.a. 'relaxed memory') also give the feeling that they are gratuitous puzzles. At a recent Schloss Dagstuhl meeting (15 191), one of the authors tried to use the possible values notation to record the non-determinism that results from not knowing when the various caches are flushed. It must be conceded that, on the pure puzzle examples, possible values are doing little more than providing an alternative notation for disjunctions. A challenge is to find a genuinely useful piece of code that, despite non-determinism, satisfies a coherent overall specification under, say, total store order (TSO) or partial store order (PSO) memory models. Only on such an application should the judgement about the usefulness of possible values be based.

There are also alternative views of the possible values notation itself. For example, $\widehat{b}$ could yield a sequence of values rather than a set. There is however an argument for preserving a (direct) way of denoting the set of possible values.

4.2. Possible evaluations of expressions

As well as possible values of an expression $\widehat{e}$ that is the set of values of e evaluated in each state of the execution, one can define $\widehat{\widehat{e}}$ as the set of all possible evaluations of e over the execution interval: each instance of a variable x in e takes on one of the values of x in the interval so that different occurrences of x within e may take on different values, and the values of separate variables x and y may be taken from different states. The set of evaluations includes those in which the values of all the variables are taken in a single state and hence $\widehat{e} \subseteq \widehat{\widehat{e}}$. In [7] the possible values concept was linked to different forms of nondeterministic expression evaluation corresponding to $\widehat{e}$ and $\widehat{\widehat{e}}$.

The following simple rule requires no restriction on e other than it does not contain references to x because x is in the frame of the specification.

$$\textbf{rely}\ x' = x \cdot x : [x' \in \widehat{\widehat{e}}] \quad \sqsubseteq \quad x \leftarrow e\ .$$

If e satisfies the single reference property over the execution interval (as defined earlier) then $\widehat{\widehat{e}} = \widehat{e}$ and hence

$$\begin{aligned} & \textbf{rely}\ x' = x \wedge (\textstyle\bigwedge z \in S \cdot z' = z) \cdot x : [x' \in \widehat{e}] \\ = \ & \textbf{rely}\ x' = x \wedge (\textstyle\bigwedge z \in S \cdot z' = z) \cdot x : [x' \in \widehat{\widehat{e}}] \\ \sqsubseteq \ & x \leftarrow e\ . \end{aligned}$$

4.3. Auxiliary variables

The statement is made in [18] that using auxiliary (a.k.a. ghost) variables in the specification of a software component *can* destroy compositionality by encoding too much information about the environment. Studying possible values has helped put the position more clearly:

- having the code of the environment gives maximum information — but minimal compositionality
- the same distinction is actually there with sequential programs where post conditions provide an abstract description of functionality without committing to an algorithm (they can also leave unconstrained the values left in temporary variables etc.)
- for concurrency, things are much more sensitive: one ideal is that the visible variables (read and write) of parallel processes are 'separate' — this might be true on a concrete representation even when an abstract description appears to admit interference — see [21]
- rely/guarantee conditions are an attempt to state only what matters
- the expressive 'weakness' of rely/guarantee conditions (is conceded and) can be a positive attribute
- auxiliary variables can be used to encode extra information about the environment — in the extreme, with use of statement counters, they can encode as much as the program being executed by the environment

The advice is to minimise the use of auxiliary variables — even when writing assertions, abstraction from the environment can be lost if gratuitous information is recorded in auxiliary variables. The 'possible values' notation appears to offer an intuitive specification tool and a principled way of avoiding the need for some auxiliary variables.

One indication of the compositional nature of rely conditions is that, if a component with a rely condition r is refined to a sequential composition, each subcomponent inherits the rely condition r. Conversely, a sequential composition guarantees a relation g if each component of the sequential composition guarantees g.

Acknowledgements

An earlier version of this paper was prepared for a conference that celebrated José Nuno Oliveira's sixtieth birthday. The authors of the current paper thank the organisers of that memorable event in Guimarães and take the opportunity to renew their warmest good wishes to José.

Outlines of material on possible values were presented at the 2015 meeting of IFIP WG 2.3 and at the Schloss Dagstuhl meeting 15 191 — on both occasions comments were made that have helped clarify the ideas and their explanation. In Dagstuhl, useful discussions with Viktor Vafeiadis helped one author understand the issues around weak memory. Useful comments on a draft from Diego Machado Dias are gratefully acknowledged as are those of the anonymous journal referees.

The research reported here is funded by the EPSRC responsive mode grant on "Taming Concurrency" EP/K011707/1, the EPSRC Platform Grant "TrAmS-2" EP/J008133/1 and the ARC grant DP130102901; the authors express their thanks for this support.

References

[1] J.-R. Abrial, The B-Book: Assigning Programs to Meanings, Cambridge University Press, 1996.
[2] J.-R. Abrial, Modeling in Event-B, Cambridge University Press, 2010.
[3] Richard Bornat, Hasan Amjad, Inter-process buffers in separation logic with rely-guarantee, Form. Asp. Comput. 22 (6) (2010) 735–772.
[4] Richard Bornat, Hasan Amjad, Explanation of two non-blocking shared-variable communication algorithms, Form. Asp. Comput. 25 (6) (2013) 893–931.
[5] R.J.R. Back, A method for refining atomicity in parallel algorithms, in: Eddy Odijk, Martin Rem, Jean-Claude Syre (Eds.), PARLE '89 Parallel Architectures and Languages Europe, in: LNCS, vol. 366, 1989, pp. 199–216.
[6] Ian Hayes (Ed.), Specification Case Studies, second edition, Prentice Hall International, 1993.
[7] Ian J. Hayes, Alan Burns, Brijesh Dongol, Cliff B. Jones, Comparing degrees of non-determinism in expression evaluation, Comput. J. 56 (6) (2013) 741–755.
[8] Neil Henderson, Formal modelling and analysis of an asynchronous communication mechanism, PhD thesis, University of Newcastle upon Tyne, 2004.
[9] Ian J. Hayes, Cliff B. Jones, Robert J. Colvin, Laws and semantics for rely-guarantee refinement, Technical report CS-TR-1425, Newcastle University, July 2014.
[10] Maurice Herlihy, Jeannette M. Wing, Linearizability: a correctness condition for concurrent objects, ACM Trans. Program. Lang. Syst. 12 (3) (1990) 463–492.
[11] Cliff B. Jones, Ian J. Hayes, Robert J. Colvin, Balancing expressiveness in formal approaches to concurrency, Form. Asp. Comput. 27 (2015) 475–497.
[12] C.B. Jones, Software Development: A Rigorous Approach, Prentice Hall International, 1980.
[13] C.B. Jones, Development methods for computer programs including a notion of interference, PhD thesis, Oxford University, June 1981, Printed as: Programming Research Group, Technical Monograph 25.

[14] C.B. Jones, Specification and design of (parallel) programs, in: Proceedings of IFIP'83, North-Holland, 1983, pp. 321–332.
[15] C.B. Jones, Systematic Software Development Using VDM, second edition, Prentice Hall International, 1990.
[16] C.B. Jones, Accommodating interference in the formal design of concurrent object-based programs, Form. Methods Syst. Des. 8 (2) (March 1996) 105–122.
[17] C.B. Jones, Splitting atoms safely, Theor. Comput. Sci. 375 (1–3) (2007) 109–119.
[18] C.B. Jones, The role of auxiliary variables in the formal development of concurrent programs, in: Cliff B. Jones, A.W. Roscoe, Kenneth Wood (Eds.), Reflections on the Work of C.A.R. Hoare, Springer, 2010, pp. 167–188, Chapter 8.
[19] Cliff B. Jones, Ken G. Pierce, Splitting atoms with rely/guarantee conditions coupled with data reification, in: ABZ2008, in: LNCS, vol. 5238, Springer, 2008, pp. 360–377.
[20] Cliff B. Jones, Ken G. Pierce, Elucidating concurrent algorithms via layers of abstraction and reification, Form. Asp. Comput. 23 (3) (2011) 289–306.
[21] Cliff B. Jones, Nisansala Yatapanage, Reasoning about separation using abstraction and reification, in: Radu Calinescu, Bernhard Rumpe (Eds.), Software Engineering and Formal Methods, in: LNCS, vol. 9276, Springer, 2015, pp. 3–19.
[22] Leslie Lamport, On interprocess communication, part II: algorithms, Distrib. Comput. 1 (2) (1986) 86–101.
[23] Leslie Lamport, Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers, Addison Wesley, 2003.
[24] P.W. O'Hearn, Resources, concurrency and local reasoning, Theor. Comput. Sci. 375 (1–3) (May 2007) 271–307.
[25] John C. Reynolds, Separation logic: a logic for shared mutable data structures, in: Proceedings of 17th LICS, IEEE, 2002, pp. 55–74.
[26] Tom Ridge, A rely-guarantee proof system for x86-TSO, in: Verified Software: Theories, Tools, Experiments, Springer, 2010, pp. 55–70.
[27] John Rushby, Model checking Simpson's four-slot fully asynchronous communication mechanism, Technical report, Computer Science Laboratory, SRI International, Menlo Park CA 94025, USA, July 2002.
[28] H.R. Simpson, Four-slot fully asynchronous communication mechanism, IEE Proc., Comput. Digit. Tech. 137 (1) (1990) 17–30.
[29] Jaroslav Ševčik, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, Peter Sewell, Relaxed-Memory Concurrency and Verified Compilation, ACM SIGPLAN Not., vol. 46, ACM, 2011, pp. 43–54.
[30] Viktor Vafeiadis, Francesco Zappa Nardelli, Verifying fence elimination optimisations, in: Eran Yahav (Ed.), Static Analysis, in: LNCS, vol. 6887, Springer, 2011, pp. 146–162.
[31] Shuling Wang, Xu Wang, Proving Simpson's four-slot algorithm using ownership transfer, in: Markus Aderhold, Serge Autexier, Heiko Mantel (Eds.), VERIFY-2010, in: EPiC Series, vol. 3, EasyChair, 2012, pp. 126–140.