COMPUTING SCIENCE

Possible values: exploring a concept for concurrency

Cliff B. Jones and Ian J. Hayes

TECHNICAL REPORT SERIES

No. CS-TR-1483 September 2015

TECHNICAL REPORT SERIES

No. CS-TR-1483 September, 2015

Possible values: exploring a concept for concurrency

Cliff B. Jones; Ian J. Hayes

Abstract

An important issue in concurrency is interference. This issue manifests itself in both shared-variable and communication based concurrency - this paper focusses 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 "possible values" 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. The possible values concept also helps sharpen some issues around atomicity.

Printed and published by Newcastle University,
Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.

Bibliographical details

JONES, C. B; HAYES, I. J.

An Empirical Study Comparing the PEPA Eclipse Plug-in and GPA Tools
[By] C. B Jones and I. J Hayes

Newcastle upon Tyne: Newcastle University: Computing Science, 2015.

(Newcastle University, Computing Science, Technical Report Series, No. CS-TR-1483)

Added entries

NEWCASTLE UNIVERSITY
Computing Science. Technical Report Series. CS-TR-1483

Abstract

An important issue in concurrency is interference. This issue manifests itself in both shared-variable and communication based concurrency - this paper focusses 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 "possible values" 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. The possible values concept also helps sharpen some issues around atomicity.

About the authors

Cliff Jones is Professor of Computing Science at Newcastle University. He is best known for his research into "formal methods" for the design and verification of computer systems; under this heading, current topics of research include concurrency, support systems and logics. He is also currently applying research on formal methods to wider issues of dependability.

Prof. Ian J. Hayes has spent several periods at the School of Computing Science and these have resulted in joint publications (including several with Prof. Cliff Jones). He is now pursuing joint research which will result in further papers. Prof. Hayes also gives seminars at the School. He has given keynote presentations at five conferences in the last ten years including: the International Conference on Theoretical Aspects of Computing 2010 and the International Symposium on Unifying Theories of Programming 2006. He also won the best paper (joint with Dr. R. Colvin) at the 7th International Conference on Integrated Formal Methods in 2009.

Suggested keywords

CONCURRENT PROGRAMMING
RELY-GUARANTEE CONDITIONS
POSSIBLE VALUES

Possible values: exploring a concept for concurrency

As for José Nuno Oliveira's *Festschrift*

Cliff B. Jones and Ian J. Hayes

September 29, 2015

Abstract

An important issue in concurrency is interference. This issue manifests itself in both shared-variable and communication based concurrency — this paper focusses 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 "possible values" 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. The possible values concept also helps sharpen some issues around atomicity.

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 (cf. 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:

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

is satisfied by a simple assignment of y to x (assuming the access to read the value of y is atomic).

1.1 Rely/Guarantee thinking

Before going into more detail on the possible values notation (cf. Section 2), a brief overview of background work is offered. The specifications given in Section 3 are written in the notation of VDM [Jon80, Jon90]. It is unlikely that they will present difficulties even to readers unfamiliar with that notation because similar ideas for sequential programs are present in Z [Hay93], B [Abr96] and Event-B [Abr10]. 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 any 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 [Jon81, Jon83][2] idea 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

[1] Much of the literature on rely/guarantee conditions is limited to normal (or 'scoped') variables; [JY15] discusses 'heap' variables.

[2] The literature on Rely/Guarantee approaches continues to expand; see [JHC15, HJC14] for further references. For a reader who is completely unfamiliar with rely/guarantee concepts, a useful brief presentation can be found in [Jon96].

recording 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 including the case that the environment will change the polarity of a flag in one direction and the specified process in the reverse way.

It is useful to compare the roles of the two new 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.

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 will 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 a more 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. (This topic is returned to in Section 1.2.)

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, [Jon90]. 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 [Jon07]. One reason that this is interesting is Peter O'Hearn's suggested dichotomy in [O'H07] 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, respec-

[3] Notice that there is an essential difference here from 'actions' [Bac89] or 'events' [Abr10] which view execution of a guarded action as atomic.

tively, even and odd indices (for a full development of this example, see [HJC14]). If a single variable t were used to record the least index of an element that satisfies P, there would be a data race between the two processes potentially changing 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 a form of rely/guarantee reasoning in which the (potentially) four conditions are combined. More recent work [JHC15, HJC14] has broken the connection to yield a refinement calculus style in which rely and/or guarantee constraints can be wrapped around any command including conventional relations.

1.2 Auxiliary variables

The statement is made in [Jon10] 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: an 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 — cf. [JY15]
- 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.

1.3 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 4 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 [HJC14]. The current authors recognise that this paper represents the start of an exploration — some avenues to be investigated are mentioned in Section 5. They hope that the exploratory nature of the contribution will please the dedicatee!

2 Possible values of variables

It is argued above that the confessed expressive weakness of rely/guarantee specifications might actually serve 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 5.

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

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

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 assignment $x \leftarrow y$. As is pointed out in Section 3, an instance of this simple case was the inspiration for the possible values notation and also suffices for the current example. 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.

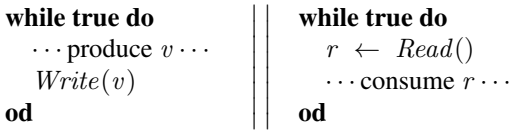

Figure 1: Code to clarify reader/writer structure

3 Asynchronous Communication Mechanisms

An Asynchronous Communication Mechanism (ACM) logically provides a one-place buffer between a single writer and a single reader (see Figure 1). 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. 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 [Sim90], Hugo Simpson proposed a 'four-slot' algorithm to implement an ACM that, 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.

- It is assumed that there is only a single reader and a single writer but the reader and writer processes operate completely asynchronously
- 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)
- 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
- 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 [Lam86] 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 these requirements should be expressed in a formal specification and there are many attempts. (Other approaches include [Hen04, Abr10].) Without surveying all of them, it fits the theme of this paper to review two strands of publications: one motivated by (Concurrent) Separation Logic [Rey02, O'H07] 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 [BA10], separation logic is certainly used but it is interesting to see that the paper also uses rely/guarantee concepts. In contrast, [BA13] makes no real use of separation logic and the specification uses the concept of linearisability [HW90]. 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 [WW12] 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 [JP08, JP11] that use rely/guarantee ideas. In the development recorded in [JP08],[4] 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 [JP11] resolves the problem by using

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

Rushby [Rus02] 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

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

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).

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 ($data\text{-}w$) 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 [Jon90, Sect. 9.3] as using a 'biased' representation: the state contains values which 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 are normally preferred because they make it easier to see the range of possible implementations. One further surprising fact about the specifications in [JP08, JP11] 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 [JP11] 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 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 $Value$. This only works by using 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 $Read$. As in some earlier specifications (including [JP11]), 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 [JP11], 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 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 [JP11]) 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. That the observable effect of both operations takes place in a single atomic step links to Bornat's use of the concept of linearisability [BA13].

The initial state (as in all specifications) 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 been shown to offer a short and clear overall specification of ACM behaviour.

3.4 Intermediate reification

The challenge of presenting a specification that makes sense to potential users is addressed in Section 3.3. The next hurdle is to show how to structure an explanation of the design decisions that show what is going on in Simpson's algorithm: this is tackled here and in the next sub-section.

The first refinement step is based on the approach in [JP11]: a generalisation of Simpson's four slots is represented by a map of an indexed set of 'slots' $X \xrightarrow{m} Value$; the index set X is deliberately left unspecified at this stage. Here, this part of the state is named dw. In addition, the state contains three variables whose values are used to communicate between $Write_i$ and $Read_i$. 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 an interesting observation that none of the variables can be modified by both operations. Each of these variables contains a single value from the index set X. As a VDM record, the state is:

$$\begin{array}{lllll} \Sigma_i :: & dw & : X \xrightarrow{m} Value & \text{– the data value buffers} \\ & lw & : X & \text{– index of the last completely written slot} \\ & cw & : X & \text{– index of the current write slot} \\ & cr & : X & \text{– index of the current read slot} \end{array}$$

There is also a data type invariant that requires that the (potentially partial) map has a value in every slot **dom** $dw = X$. 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.

It is not difficult to follow the lines of the data reification being undertaken in this section: 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 **dom** $dw = X$.

The retrieve function and the initial specification of $Write$ suggest immediately:

$$\begin{array}{l} post\text{-}Write_i\colon dw'(lw') = v \\ guar\text{-}Write_i\colon dw'(lw') \neq dw(lw) \;\Rightarrow\; dw'(lw') = v \end{array}$$

But, because the two operations can be executed concurrently, it is necessary to ensure that $Write_i$ does not interfere with a slot that $Read_i$ might be accessing. Facilitating this non-interference is exactly the role of cr and an additional conjunct is added to the guarantee condition:

$$guar\text{-}Write_i\colon (\cdots) \wedge dw'(cr) = dw(cr)$$

However, if $Read_i$ could change cr arbitrarily, this would be unachievable, so it is necessary to record that cr can only change to the index of the last completely written slot:

$$rely\text{-}Write_i\colon cr' \in \{cr, lw\}$$

Turning to $Read_i$, it is clear that its guarantee condition must match $rely\text{-}Write_i$:

$$guar\text{-}Read_i\colon cr' \in \{cr, lw\}$$

$Write_i(v\colon Value)$
owns wr dw, lw, cw
ext rd cr
rely $cr' \in \{cr, lw\}$
guar $(dw'(lw') \neq dw(lw) \;\Rightarrow\; dw'(lw') = v) \wedge dw'(cr) = dw(cr)$
post $dw'(lw') = v$

$Read_i()\ r\colon Value$
owns wr cr
ext rd dw, lw
rely $dw'(cr) = dw(cr)$
guar $cr' \in \{cr, lw\}$
post $r' \in \widehat{dw(lw)} \wedge cr' \in \widehat{lw} \wedge r' = dw'(cr')$

Figure 2: The intermediate specifications

As indicated above, the $Read_i$ operation uses cr to indicate to $Write_i$ which slot it is accessing. For the post condition, the retrieve function gives:

$$post\text{-}Read_i\colon r' \in \widehat{dw(lw)}$$

but it is also necessary to remove the freedom left in the guarantee condition and insist that cr corresponds to either the most recent completely written value at the start of the read or a later completely written value — this adds:

$$post\text{-}Read_i\colon (\cdots) \wedge cr' \in \widehat{lw} \wedge r' = dw'(cr')$$

This result can only be achieved if the design of $Read_i$ can assume the value being read is stable:

$$rely\text{-}Read_i\colon dw'(cr) = dw(cr)$$

The specifications are summarised in Figure 2. It is a consequence of the fact that $Write_i$ must not change $dw(cr)$ together with the fact that cr can change but only to the value of lw that there must be at least three slots (i.e. **card** $X \geq 3$). This observation is perhaps more obvious when pseudo code for this intermediate refinement is considered (cf. Figure 3) because it uses the variable cw as foreseen by the earlier description. It must, however, be emphasised that this 'code' is only given as an aid to the reader's intuition — it is not part of the formal development.

In Figure 3, the marking of $\langle cr \;\leftarrow\; lw \rangle$ as atomic (via enclosing it in angle brackets) is also interesting: because the environment ($Write_i$) could change the value of lw between its access by $Read_i$ and the completion of the write to cr, the guarantee condition of $Read_i$ would not be met with a non-atomic assignment. Of course, such atomic statements are not permitted in the final implementation so the removal of this atomicity requirement is a challenge for the subsequent development.

Furthermore, although the whole point of using multiple slots in order to achieve the ACM requirements is to avoid any assumption about it being possible to read and

$Write_i(v\colon Value)$
  $cw :\in X - \{cr, lw\}$;
  $dw(cw) \leftarrow v$;
  $lw \leftarrow cw$

$Read_i()r\colon Value$
  $\langle cr \leftarrow lw\rangle$;
  $r \leftarrow dw(cr)$

Figure 3: Suggestive pseudo code for the intermediate development step

write elements of $Value$ atomically, precisely this assumption is so far being made about elements of X. Reducing this assumption to the communication of single bits is one of the key achievements of Hugo Simpson's 'four slot' implementation, as is outlined in the Section 3.5.

Before moving to that material however, there are some illuminating observations about the development thus far. The guarantee condition of $Write$ used in the abstract specification was there to fix the overall function of what it means to execute a write to the buffer; there is no matching rely condition in the abstract $Read$ operation. The additional rely and guarantee conditions in the intermediate refinement ($Read_i$, $Write_i$) play the more normal role of fixing the permitted interference between the two threads. In fact, they have established the protocol of the exchange of ownership of the slots between the threads (cf. the discussion at the beginning of Section 3).

There is however a point that needs to be made clear here. Just as one can weaken pre conditions (which are permissions for the developer to make assumptions about the context of use) and strengthen post conditions (which are obligations on the code to be developed), it is permissible to weaken rely conditions and strengthen guarantee conditions.[5] In the text above, the first stab at any of the conditions is derived by mapping them –under the retrieve function– from the abstract to the intermediate level. What might surprise the reader is that new terms are added to these derived rely conditions. This is permissible because the context (see Figure 1) defines that only the reader and writer processes can update the variables, so the only obligation is to show that the strengthened guarantee conditions imply the strengthened rely conditions.

The post condition of the top-level specification $Read$ requires $r' \in \widehat{b}$ which via the retrieve function, $b = dw(lw)$, requires $r' \in \widehat{dw(lw)}$ for the intermediate reification $Read_i$. Note that $\widehat{dw(lw)}$ corresponds to the possible values of $dw(lw)$ in one of the states and hence for each possible value of $dw(lw)$ the values of dw and lw are taken in the same state. Returning to the post condition of $Read_i$ in Figure 2, note that $r' \in \widehat{dw(lw)}$ is implied by

$$dw'(cr') \in \widehat{dw(lw)} \wedge r' = dw'(cr')$$

[5] In contrast to the original 5-tuple form of rely/guarantee specification, this is much clearer in the new 'deconstructed' version where [JHC15, HJC14] has precise laws for these changes.

where the variable cr records the slot currently being read. The rely condition of $Read_i$ requires that $dw(cr)$ is stable and hence the above requirement is equivalent to

$$\widehat{dw(cr')} \subseteq \widehat{dw(lw)} \wedge r' = dw'(cr')$$

which is in turn implied by $cr' \in \widehat{lw} \wedge r' = dw'(cr')$. Note how the task of ensuring that $dw'(cr') \in \widehat{dw(lw)}$ has been reduced to ensuring that the index $cr' \in \widehat{lw}$.

The guarantee of $Write$, $b' \neq b \;\Rightarrow\; b' = v$, requires that the update of the buffer happens atomically. But because it is assumed the values of type $Value$ cannot be written atomically, the implementation makes use of multiple buffers. The write (non-atomically) updates a buffer that differs from both the current read slot (cr) and the last completely written slot (lw), which a new read may access. It achieves apparent atomicity by updating the index lw atomically.

It is interesting to note that the issue of (data) race freedom on the slots is worked out with rely/guarantee conditions. This can be contrasted with Peter O'Hearn's view in [O'H07] 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.

3.5 Focus on four slots

This paper contains only an outline of the remaining development. It broadly follows [JP11]. Although the observation is made in Section 3.4 that three slots would be adequate to avoid clashing,[6] 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 real implementation, these bits can be 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 [JP11], 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 "$\neg$" 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).

Thus, the representation of X from the intermediate representation is a pair $P \times S$. The concrete state is:

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

[6] In fact, [BA13] also considers a three slot implementation.

$Write_i(v\colon Value)$
  $cpw \leftarrow \neg\, cpr$;
  $dsw(cpw, \neg\, sw(cpw)) \leftarrow v$;
  $sw(cpw) \leftarrow \neg\, sw(cpw)$;
  $lpw \leftarrow cpw$

$Read_i()r\colon Value$
  $cpr \leftarrow lpw$;
  $csr \leftarrow sw(cpr)$;
  $r \leftarrow dsw(cpr, csr)$ — rely ensures expression is stable

Figure 4: Suggestive pseudo code for the final development step

Pseudo-code is given in Figure 4 to indicate the algorithm on Σ_f — just as with Figure 3, this will not form part of the final correctness argument: it is shown only to aid the reader's intuition.

4 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 [HJC14]: 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. This simple case covers the usage of possible values in the example of Section 3 and all of the others which have been driven by practical applications. A decision has to be made about the required meaning in the case where both the process (in whose post condition the possible values term is used) and its environment can change the relevant variable. This topic will be addressed when a practical application shows the need (and hopefully gives a steer as to the most appropriate choice).

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 mutual 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 [JHC15, HJC14] in which the specification statement $x{:}\big[q\big]$ establishes the postcondition q and modifies only x, and the command c in a rely context of r is written **rely** $r \cdot c$. Assuming a read of y is atomic, the following law holds.

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

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 above, annotated **owns wr** x.

Replacing y in the above law 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

$$\textbf{rely}(x' = x \wedge (\bigwedge z \in S \cdot z' = z)) \cdot x{:}\big[x' \in \widehat{e}\big] \quad \sqsubseteq \quad x \leftarrow e\,. \tag{2}$$

If the environment can be relied upon to ensure that the value of an expression e increases monotonically, i.e. $e \leq e'$, then evaluating e in any intermediate state gives a value between the initial and final values of e.

$$\textbf{rely}\; x' = x \wedge e \leq e' \cdot x{:}\big[e \leq x' \leq e'\big] \;\sqsubseteq\; \textbf{rely}\; x' = x \wedge e \leq e' \cdot x{:}\big[x' \in \widehat{e}\big]$$

This generalises to any reflexive, transitive relation r as follows.

$$\begin{array}{ll} & \textbf{rely}\; x' = x \wedge r(e, e') \cdot x{:}\big[r(e, x') \wedge r(x', e')\big] \\ \sqsubseteq & \textbf{rely}\; x' = x \wedge r(e, e') \cdot x{:}\big[x' \in \widehat{e}\big] \end{array}$$

The earlier law (2) can be used to specialise the resultant refinement further provided e satisfies the constraints in the earlier law.

If every step of the environment ensures that the value of e either does not change or, if it does, it changes to some value v just once, the possible values of e are either its initial value or v, which leads to the following law.

$$\textbf{con}\; v \cdot \textbf{rely}\; x' = x \wedge (e' \neq e \;\Rightarrow\; e' = v) \cdot x{:}\big[x' \in \widehat{e}\big] \quad \sqsubseteq \quad x \leftarrow e$$

The logical constant **con** v is effectively quantified over the whole construct (as opposed to existentially quantifying it within the rely relation which would reduce the second conjunct of the rely condition to true).

Note that there is a subtle difference between the specifications

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

which samples y once and hence ensures the final value of x is even, and

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

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

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 [HBDJ13] 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.

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

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

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

5 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 a layered description. 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 is set out in [HBDJ13]. This paper is 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. The authors recognise that this is the beginning of an exploration rather than a finished proposal but they hope that José Nuno Oliveira will accept the tentative material as our birthday offering.

This closing section hints at 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.

5.1 Further applications

In the spirit of birthday fun, it can perhaps be mentioned that the possible values notation appears to have some potential for recording arguments about brain-teaser puzzles. At a recent meeting of IFIP WG 2.3, 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 (cf. [ŜVZN$^+$11, VN11, Rid10]) about 'weak memory' (a.k.a. 'relaxed memory') also give the feeling that they are gratuitous puzzles. At a recent Schloss Dagstuhl meeting (15191), one of the authors again 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. The 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.

Acknowledgements

Outlines of this material were presented at the 2015 meeting of IFIP WG 2.3 and at the Schloss Dagstuhl meeting 15191 — on both occasions comments were made that have helped clarify the current presentation. In particular, useful discussions with Viktor Vafeiadis helped one author understand the issues around weak memory. Further useful comments on a draft from Diego Machado Dias are gratefully acknowledged as are those of the anonymous journal referees.

For the funding for their research, the authors of this paper gratefully acknowledge the EPSRC responsive mode grant on "Taming Concurrency", the EPSRC Platform Grant "TrAmS-2" and the ARC grant DP130102901.

References

[Abr96] J.-R. Abrial. *The B-Book: Assigning programs to meanings*. Cambridge University Press, 1996.

[Abr10] J.-R. Abrial. *Modeling in Event-B*. Cambridge University Press, 2010.

[BA10] Richard Bornat and Hasan Amjad. Inter-process buffers in separation logic with rely-guarantee. *Formal Aspects of Computing*, 22(6):735–772, 2010.

[BA13] Richard Bornat and Hasan Amjad. Explanation of two non-blocking shared-variable communication algorithms. *Formal Aspects of Computing*, 25(6):893–931, 2013.

[Bac89] R.J.R. Back. A method for refining atomicity in parallel algorithms. In Eddy Odijk, Martin Rem, and Jean-Claude Syre, editors, *PARLE '89 Parallel Architectures and Languages Europe*, volume 366 of *LNCS*, pages 199–216. 1989.

[Hay93] Ian Hayes, editor. *Specification Case Studies*. Prentice Hall International, second edition, 1993.

[HBDJ13] Ian J. Hayes, Alan Burns, Brijesh Dongol, and Cliff B. Jones. Comparing degrees of non-determinism in expression evaluation. *The Computer Journal*, 56(6):741–755, 2013.

[Hen04] Neil Henderson. *Formal Modelling and Analysis of an Asynchronous Communication Mechanism*. PhD thesis, University of Newcastle upon Tyne, 2004.

[HJC14] Ian J. Hayes, Cliff B. Jones, and Robert J. Colvin. Laws and semantics for rely-guarantee refinement. Technical Report CS-TR-1425, Newcastle University, July 2014.

[HW90] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[JHC15] Cliff B. Jones, Ian J. Hayes, and Robert J. Colvin. Balancing expressiveness in formal approaches to concurrency. *Formal Aspects of Computing*, 27:475–497, 2015.

[Jon80] C. B. Jones. *Software Development: A Rigorous Approach*. Prentice Hall International, 1980.

[Jon81] 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.

[Jon83] C. B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332. North-Holland, 1983.

[Jon90] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, second edition, 1990.

[Jon96] C. B. Jones. Accommodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122, March 1996.

[Jon07] C. B. Jones. Splitting atoms safely. *Theoretical Computer Science*, 375(1–3):109–119, 2007.

[Jon10] C. B. Jones. The role of auxiliary variables in the formal development of concurrent programs. In Cliff B. Jones, A. W. Roscoe, and Kenneth Wood, editors, *Reflections on the work of C.A.R. Hoare*, chapter 8, pages 167–188. Springer, 2010.

[JP08] Cliff B. Jones and Ken G. Pierce. Splitting atoms with rely/guarantee conditions coupled with data reification. In *ABZ2008*, number 5238 in LNCS, pages 360–377. Springer, 2008.

[JP11] Cliff B. Jones and Ken G. Pierce. Elucidating concurrent algorithms via layers of abstraction and reification. *Formal Aspects of Computing*, 23(3):289–306, 2011.

[JY15] Cliff B. Jones and Nisansala Yatapanage. Reasoning about separation using abstraction and reification. In Radu Calinescu and Bernhard Rumpe, editors, *Software Engineering and Formal Methods*, volume 9276 of *LNCS*, pages 3–19. Springer, 2015.

[Lam86] Leslie Lamport. On interprocess communication, part ii: Algorithms. *Distributed Computing*, 1(2):86–101, 1986.

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

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of 17th LICS*, pages 55–74. IEEE, 2002.

[Rid10] Tom Ridge. A rely-guarantee proof system for x86-TSO. In *Verified Software: Theories, Tools, Experiments*, pages 55–70. Springer, 2010.

[Rus02] John Rushby. Model checking simpsons four-slot fully asynchronous communication mechanism. Technical report, Computer Science Laboratory, SRI International, Menlo Park CA 94025 USA, July 2002.

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

[ŜVZN+11] Jaroslav Ŝevčik, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. Relaxed-memory concurrency and verified compilation. In *ACM SIGPLAN Notices*, volume 46, pages 43–54. ACM, 2011.

[VN11] Viktor Vafeiadis and Francesco Zappa Nardelli. Verifying fence elimination optimisations. In Eran Yahav, editor, *Static Analysis*, volume 6887 of *LNCS*, pages 146–162. Springer, 2011.

[WW12] Shuling Wang and Xu Wang. Proving Simpson's four-slot algorithm using ownership transfer. In Markus Aderhold, Serge Autexier, and Heiko Mantel, editors, *VERIFY-2010*, volume 3 of *EPiC Series*, pages 126–140. EasyChair, 2012.