Elucidating concurrent algorithms via layers of abstraction and reification

Cliff B. Jones, Ken G. Pierce

Under consideration for publication in Formal Aspects of Computing

Elucidating concurrent algorithms via layers of abstraction and reification

Cliff B. Jones and Ken G. Pierce
School of Computing Science, Newcastle University, UK

Version C Dated: 27 February 2010

Abstract. Arguing that intricate concurrent programs satisfy their specifications can be difficult; recording understandable explanations is important for subsequent readers. Abstraction is a key tool even for sequential programs. The purpose here is to explore some abstractions that help readers (and writers) understand the design of concurrent programs. As an illustration, the paper presents a formal development of a non-trivial parallel program: Simpson's implementation of asynchronous communication mechanisms (ACMs). Although the correctness of this "4-slot algorithm" has been shown elsewhere, earlier proofs fail to offer much insight into the design. From an understandable (yet formal) design history of this one algorithm, the techniques employed in the explanation are teased out for wider application. Among these techniques is using a "fiction of atomicity" as an aid to understanding the initial steps of development. The rely-guarantee approach is, here, combined with notions of read/write frames and "phased" specifications; furthermore, the atomicity assumptions implied by the rely/guarantee conditions are achieved by clever choice of data representations.

1. Introduction

This paper is intended to contribute to methods of developing parallel programs; in particular it extends the repertoire of ways of "splitting (software) atoms safely". As an illustration, it explains an intricate parallel program.

The general case for developing programs from abstractions is taken as read (cf. [Jon90, Abr96]). The VDM literature uses the terms "operation decomposition" and "data reification" for design steps of sequential programs and provides proof obligations to justify such steps. Even if –as here– what is being recorded is a rational reconstruction of a design, the resulting documentation offers clarity and captures a design history to inform subsequent modification.

Research on rely/guarantee conditions (see Section 1.2 below) extends the formal tools so as to cope with shared-variable concurrent programs. As has been repeatedly made clear in the literature, "compositionality" is essential to derive real pay off from a "posit and prove" approach.

More recently, research has looked at using a "fiction of atomicity" [Jon03] as an additional abstraction in the specification of parallel programs; the corresponding development notion is sometimes referred to as "splitting (software) atoms safely"; an example of this approach is the transformation rules for $\pi o\beta\lambda$ as in [Jon96]. This paper uses rely/guarantee conditions in reasoning about "splitting atoms". In particular, the example illustrates the combination of rely/guarantee reasoning with data reification as outlined in [Jon07].

Correspondence and offprint requests to: cliff.jones@ncl.ac.uk

The example application chosen concerns “Asynchronous Communication Methods” — specifically, the four-slot implementation of ACMs published by Hugo Simpson [Sim97][1] — see Section 2. The algorithm is ingenious and its correctness by no means obvious. Rather than being just another proof, our hope is that this development offers insight into why Simpson’s algorithm satisfies the requirements.

A comment is perhaps in order here about the use of support tools in establishing formal properties of programs. The current authors are working on such tools but consider it crucial that one’s ideas are clear before using support tools. Furthermore, one should be wary of tools that constrain modes of expression so that one ends up “coding” intuition into a restricted language.

The main message of the current paper is however the (generic) approach outlined: Section 5 restates the methods used so that it is clear what the reader can take from the specific example to other specification and design challenges. Clearly not all applications will use exactly the set of ideas here: many applications will be less demanding than Simpson’s intricate algorithm — doubtless, other approaches will also be invented or deployed. But the authors hope that readers can derive benefit from the collection of ideas used here.

Although this paper offers comparisons (see Section 6), it is quite specifically not competitive. The first author co-supervised Neil Henderson’s PhD and encouraged the view that each of the approaches used in [Hen04] threw different light on the intricate algorithm that has also been chosen for the current paper. Furthermore, this paper differs significantly from the earlier (invited) conference paper on the same topic [JP08] and Section 6.5 reviews the reasons for the changes. The remainder of this section briefly sketches state-of-the-art methods; any reader who is totally unfamiliar with any of these approaches should consult the cited publications.

1.1. Data reification

For many systems, data abstraction is key to achieving a concise and perspicuous specification. An algorithm might be easy to specify or describe in terms of tractable mathematical objects; its implementation might have to represent the abstraction in a complicated way — perhaps to achieve performance. Separation of these issues results in clearer documentation of design histories.

The preferred data reification development rule in VDM [Jon90] works where the chosen representation (of the abstraction) can be understood using a “retrieve function” that is a many-to-one mapping from the representation back to the abstraction. This is possible where the abstraction is free from “implementation bias”. The preferred VDM reification rule basically checks that (starting with a representation state) composing the retrieve function with the post condition of an abstract operation gives the same result as composing the post condition of the operation on the representation with the retrieve function. There are restrictions to pre conditions –but here they are minimal– and an obligation to prove “adequacy” of a representation. All of this is explained in [Jon90, Chapter 8].

There are however situations where the abstraction has to record information to express potential non-determinacy and this information is superfluous in a step of development that reduces the non-determinacy. In a sense this is intentional “bias” in the abstraction. In such situations it is necessary to use the development rule introduced by Tobias Nipkow in [Nip86, Nip87] that expresses a general relation between the abstraction and its reification. For an exhaustive discussion of “data refinement” see [dRE99]; for a historical account of the development of the VDM rules see [Jon89].

1.2. Rely/guarantee thinking

Just as pre conditions simplify a designer’s task by limiting the starting states in which the specified object is to be deployed, rely conditions indicate assumptions that a developer is allowed to make about the expected interference to a (shared-variable) concurrent program. Similarly, guarantee conditions can be compared to post conditions in that both are constraints on the behaviour of the created program.

VDM’s operation decomposition rules for sequential programs have always used post conditions that relate the final state to the initial state (this is in contrast to the many approaches that try to get by with predicates of the final state). Both rely and guarantee conditions are also relations between two states.

The general idea of documenting and reasoning about interference has many embodiments; some of the

[1] The authors are grateful to the reviewer who pointed out earlier relevant work — see Section 6.

references are [Jon81, Jon83a, Jon83b, Jon96] but a number of other theses have extended the basic idea.[2] As the title of this section suggests, the approach is seen as a general way of thinking and reasoning about the design of concurrent systems rather than being limited to a specific set of rules. (In fact, the general approach can also be applied to communication-based concurrency.) Rules for introducing parallel program constructs are similar to those for sequential programming; examples are presented in [CJ07] and [Jon10] discusses why there are more choices to be made for concurrent –rather than the sequential– rules. Interestingly, no rule for the introduction of concurrent constructs is needed in the development below because concurrency is present in the initial specification.

Once again, de Roever provides an encyclopaedic treatment in [dR01]; a particularly valuable contribution is the clear identification of the fact that rely/guarantee thinking achieves "compositionality". A further aspect of this is studied in [Jon10] which makes clear that "auxiliary variables" can be used in ways that can damage compositionality.

1.3. Atomicity refinement

A more recent development is the link made in [Jon07], between the achievement of a rely/guarantee specification and the designer's ability to find an appropriate data representation. This observation throws light on several older developments and is crucial to the design step in Section 4 below. Essentially, an abstraction is used that could be said to be using the "fiction of atomicity". The splitting of operations that have to be atomic on the abstraction is made possible by judicious choice of representation. So, for example, a variable whose monotonic reduction would imply locking can be represented by an expression involving the minimum of two values each of which can only be updated by one of two parallel processes. This topic is discussed in more detail in [Jon10].

2. ACMs and their specification

The abstraction and development ideas are first illustrated (in this section and Sections 3, 4) on a specific example and then refreshed in Section 5.

Asynchronous Communication Methods (ACMs) address an extremely interesting application scenario. Consider two processes that are independently timed in the sense that they are not synchronised in any way (thus "asynchronous"); furthermore, suppose that one process produces values that are to be "communicated" to the other (one writes and the other reads); the key requirement is that communication must be achieved with *no delay* to either process. Thus it is *not*, for example, possible to use a conventional shared variable –access to which is controlled by some device such as semaphores– since this can delay a process waiting for a lock to be released. To sharpen the issues, it might be useful to think of *Value*s below as being large — something that certainly can't be assigned in one machine cycle ("atomically"). ACMs are used in important high speed communication situations such as passing values from sensors to flight control software.

Sections 3 and 4 present a formal development of a well-known –and extremely ingenious– implementation of ACMs but it is clearly necessary to offer a formal starting point for such a development. The aim here is to provide a way of specifying ACM behaviour with which a user can feel comfortable.

It would fit the "splitting atoms" programme nicely if it were possible to present a specification using a simple (atomic) variable. Such a simple model would show that successive reads can see the same written value. This is because there is no synchronization between the process writing and that reading and two reads can occur without an intervening write.

Unfortunately, the highly asynchronous nature of ACMs brings further complications that mean a single simple variable is not an appropriate abstraction because it does not show all potential behaviours of an ACM. It is necessary to consider, even in the specification, the question of what behaviours are allowed when reading and writing overlap in time. The obvious case of new behaviour comes when a read action starts but a complete write executes before the read finishes. In such a case, the read is allowed to return either the value at the start of the read or that at the end. This can be extended to the case where multiple writes "overtake" a read. The specification below splits read actions into *start-Read* and *end-Read* sub-actions in

[2] See an on-line attempt to keep track of the literature at:
http://homepages.cs.ncl.ac.uk/cliff.jones/ftp-stuff/rg-hist.pdf

order to show this behaviour. This is done in a way that makes an essential limitation on the behaviour: two successive reads must not be able to return first a "newer" value followed in time by an "older" value.

A number of non-obvious consequences follow from the asynchronous essence of ACMs. The simplest is that it is certainly valid for the reader to see the same value multiple times if it cycles faster than the writer. Specifying ACMs in an understandable way is itself non-trivial. (See Section 6 for alternative specifications.) Sections 2.1 and 2.2 attempt to offer intuition before the actual specification is given in Section 2.3.

2.1. Intuition from pseudo-code

Consider the following pseudo-code:

$$\mathit{INIT};$$
$$\left(\begin{array}{l}\textbf{while true do } \mathit{Write}(v\colon \mathit{Value}) < \mathit{data\text{-}w} \leftarrow \mathit{data\text{-}w} \curvearrowright [v] > \textbf{od} \\ || \\ \textbf{while true do } \mathit{Read}()r\colon \mathit{Value} < r \leftarrow \mathit{data\text{-}w}(\textbf{len}\, \mathit{data\text{-}w}) > \textbf{od}\end{array}\right)$$

This would allow observable behaviours like (v going into *Write*; r coming out of *Read*):
in: $[x, y, z]$
out: $[x, x, z, z]$
These are (two) important aspects of asynchronous communication: the same value can be read more than once and written values might never be read.[3]

"Atomic brackets" ($< \cdots >$) are used in this pseudo-code because of multiple references to shared variables. (Note that no assumption about assignment statements being executed atomically is made here — cf. [CJ07].) Ultimately, we seek an algorithm whose only atomicity assumption is that single bit indicators can be set atomically (think of this as a signal on a single wire). There is a technique here that is familiar from the database world: that is the split of making change (to *data-w*) then committing it (by update to *fresh-w*).

So far, so good, but the pseudo-code above does not give us a way of discussing the issue of the *Read* and *Write* overlapping which is another important facet of asynchronous behaviour. What for example are the permitted behaviours of *Read*s and *Write*s overlapping? More behaviours can be discussed if we split both *Read* and *Write* into two phases. With states:[4]

$$\begin{array}{lll}\Sigma^a :: & \mathit{data\text{-}w} & : \mathit{Value}^* \\ & \mathit{fresh\text{-}w} & : \mathbb{N}_1 \\ & \mathit{hold\text{-}r} & : \mathbb{N}_1\end{array}$$

$$\textbf{inv } (\mathit{mk}\text{-}\Sigma^a(\mathit{d\text{-}w}, \mathit{fr\text{-}w}, \mathit{ho\text{-}r})) \quad \triangleq \quad 1 \leq \mathit{ho\text{-}r} \leq \mathit{fr\text{-}w} \leq \textbf{len}\, \mathit{d\text{-}w}$$

$$\textbf{init } \mathit{mk}\text{-}\Sigma^a([\mathtt{x}], 1, 1)$$

It is obviously necessary to initialise the state. Most authors who give formal presentations do this by assuming that a value, say, `x` has been written *and* read once. This can be shown as in $\mathit{init}\text{-}\Sigma^a$.

The observable behaviour (permissible outputs) comes from the pseudo-code in Figure 1. We use a "fiction of atomicity" — but split *Write* and *Read* each into two parts. The division of the write action (into *start-Write* and *commit-Write*) is not strictly necessary as far as admitting extra behaviours but it is a convenient way of discussing the overlap between read and write operations: remembering that the values being passed might be large in the sense that they might not be changed by a machine in one atomic action, it is useful to think about readying the data before it is committed.[5]

The idea here is that *data-w* retains all values written; *start-Write* first stores a new value but only *commit-Write* releases it for access by updating *fresh-w*. Conversely, *start-Read* notes the index of values that must be regarded as "fresh" and *end-Read* makes a non-deterministic choice of an index between the *hold-r* and the value of *fresh-w* at the time of completion of the read. (The suffixes of the variable names

[3] Notice that this shows that a "circular buffer" is *not* a model of the required behaviour.
[4] Remember that types in VDM are restricted by invariants; so, for example, quantifying over Σ^a only considers records that satisfy its invariant.
[5] As an aside: It would be reasonable to assume that a *Read* operation will run in less time than a *Write* — in this case it would be impossible for multiple *Write*s to complete within the time of a *Read* — such an assumption can slightly simplify solutions. This assumption is not made here (nor in most other papers).

```
while true do
    $start\text{-}Write(v: Value)$
     $< data\text{-}w \leftarrow data\text{-}w \curvearrowright [v] >$
    ;
    $commit\text{-}Write()$
    $fresh\text{-}w \leftarrow \mathbf{len}\ data\text{-}w$
od
||
while true do
    $start\text{-}Read()$
    $hold\text{-}r \leftarrow fresh\text{-}w$
    ;
    $end\text{-}Read()r: Value$
    $r \leftarrow data\text{-}w(hold\text{-}r)$
od
```

Fig. 1. Pseudo-code description of ACM

$start\text{-}Write(\mathtt{y})$	..	$mk\text{-}\Sigma^a([\mathtt{x},\mathtt{y}],1,1)$
$commit\text{-}Write()$	..	$mk\text{-}\Sigma^a([\mathtt{x},\mathtt{y}],2,1)$
$start\text{-}Read()$	..	$mk\text{-}\Sigma^a([\mathtt{x},\mathtt{y}],2,2)$
$end\text{-}Read()$	..	$r = \mathtt{y}$

Fig. 2. Sequential case

$start\text{-}Write(\mathtt{y})$	..	$mk\text{-}\Sigma^a([\mathtt{x},\mathtt{y}],1,1)$
$start\text{-}Read()$	..	$mk\text{-}\Sigma^a([\mathtt{x},\mathtt{y}],1,1)$
$end\text{-}Read()$	..	$r = \mathtt{x}$
$commit\text{-}Write()$	..	$mk\text{-}\Sigma^a([\mathtt{x},\mathtt{y}],2,1)$

Fig. 3. Interleaved case

$start\text{-}Read()$	..	$mk\text{-}\Sigma^a([\mathtt{x}],1,1)$
$start\text{-}Write(\mathtt{y})$	..	$mk\text{-}\Sigma^a([\mathtt{x},\mathtt{y}],1,1)$
$commit\text{-}Write()$	..	$mk\text{-}\Sigma^a([\mathtt{x},\mathtt{y}],2,1)$
$start\text{-}Write(\mathtt{z})$	..	$mk\text{-}\Sigma^a([\mathtt{x},\mathtt{y},\mathtt{z}],2,1)$
$commit\text{-}Write()$	..	$mk\text{-}\Sigma^a([\mathtt{x},\mathtt{y},\mathtt{z}],3,1)$
$end\text{-}Read()$	..	$r \in \{\mathtt{x},\mathtt{y},\mathtt{z}\}$
$start\text{-}Read()$	..	$mk\text{-}\Sigma^a([\mathtt{x},\mathtt{y},\mathtt{z}],3,3)$
$end\text{-}Read()$	..	$r = \mathtt{z}$

Fig. 4. Non-deterministic case

indicate whether the reader or writer can change their values; this shows straightaway that there are no variables changed by both reader and writer.)

The sub-actions can be characterised by the pseudo-code shown in Figure 1. Although *end-Read* might not select the newest item in the sequence, a value only becomes old when a newer item is returned. Since *start-Read* sets *hold-r* to the value of *fresh-w* before the choice is made and *hold-r* is never greater than *fresh-w*, the read process cannot return an "old" value (though the same value may be returned more than once).

The continued use of atomic brackets around changes to *data-w* is an indication of development work still required (see Section 2.5).

2.2. Intuition from selected test cases

Figures 2, 3 and 4 give possible executions of the pseudo-code (giving the operation name and corresponding final sate). Figure 2 is a simple sequential write and read: `y` is added to *data-w*, marked as fresh and subsequently read. In Figure 3, the read begins before the write ends and the read yields `x`.

The more complex case in Figure 4 shows the non-determinism of the read operation. By the time *end-Read* is ready to return a result, three possible values are available and one will be selected non-deterministically. Note however that a subsequent read can return neither `x` nor `y` because *hold-r* is updated to the value of *fresh-w* at the start of the read.

```
$Write(v\colon Value)$
**owns wr** $data\text{-}w, fresh\text{-}w$
  $start\text{-}Write(v\colon Value)$
    **wr** $data\text{-}w$
    **guar** $\{1..fresh\text{-}w\} \lhd data\text{-}w = \{1..fresh\text{-}w\} \lhd \overleftarrow{data\text{-}w}$
    **post** $data\text{-}w = \overleftarrow{data\text{-}w} \frown [v]$
  $commit\text{-}Write()$
    **wr** $fresh\text{-}w$
    **rd** $data\text{-}w$
    **guar** $\overleftarrow{fresh\text{-}w} \leq fresh\text{-}w$
    **post** $fresh\text{-}w = \mathbf{len}\ data\text{-}w$

$Read()r\colon Value$
**owns wr** $hold\text{-}r$
  $start\text{-}Read()$
    **wr** $hold\text{-}r$
    **rd** $fresh\text{-}w$
    **guar** $\overleftarrow{fresh\text{-}w} \leq fresh\text{-}w$
    **post** $hold\text{-}r \in \widehat{fresh\text{-}w}$
  $end\text{-}Read()r\colon Value$
    **rd** $data\text{-}w, hold\text{-}r$
    **rely** $data\text{-}w(hold\text{-}r) = \overleftarrow{data\text{-}w}(hold\text{-}r)$
    **post** $r = data\text{-}w(hold\text{-}r)$
```

Fig. 5. Specification of sub-operations on Σ^a with rely/guarantee

2.3. A specification

The pseudo-code in Figure 1 is brief and offers the intuition of what can happen[6] but for the development that follows, this needs to be presented as formal (VDM) specifications of the four operations. Furthermore, the specification has to cover interference. This is exactly the role of rely/guarantee conditions (cf. Section 1.2). Figure 5 uses VDM's **rd/wr** markings; in addition, it employs **owns wr** to indicate that no parallel process is allowed to write into these variables. This simplifies the rely conditions. Note that there are no variables changed by both *Read* and *Write*.[7] The efficacy of these markings is addressed in the thesis of the second author [Pie09]. "Phasing" shows *start-Write* and *commit-Write* can't interfere with each other (nor can *start-Read* and *end-Read*). This reduces the complexity of the rely/guarantee conditions, it avoids the need for (auxiliary vars, and) implications and it simplifies the subsequent proofs.

A new notation (since the formulation in [JP08]) is the use of $\widehat{x}$ for any "possible values" that can occur during the execution of the operation. Notice that it is possible for $\{\overleftarrow{fresh\text{-}w}, fresh\text{-}w\} \subset \widehat{fresh\text{-}w}$; in fact, there could be an arbitrary number of changes to the variable *fresh-w* if the *Write* process cycles faster than *Read*. The concept of "any possible values" is intuitive and it seems reasonable to grace it with a formal expression. We could actually avoid the need to write $\widehat{fresh\text{-}w}$ at this point because the range of indices $\{hold\text{-}r..fresh\text{-}w\}$ offers a form of auxiliary variable — but such fortuitous auxiliary variables are not always to hand and [Jon10] presents reservations about adding auxiliary variables. Finally, the use of $\widehat{fresh\text{-}w}$ in Section 3.2 cannot be avoided without a specially contrived auxiliary variable.

An astute reader might be worried that massive assumptions are being made here about what can be changed atomically. Such assumptions have to be eliminated in subsequent development. What is achieved

[6] For those who feel queasy about the use of sequentially composed sub-operations in a specification, Section 6 discusses alternatives. Furthermore, the approach of the current section can be proved to give the same behaviours as attempts at more "implicit" specifications.

[7] The suffixes of names such as *fresh-w*, *hold-r* provide a useful reminder of which process can write to the variable.

here is to show that the "fiction of atomicity" idea can provide an intuitive understanding of extremely delicate code.

The details of the rely and guarantee operations are, here, made much simpler to write because of the way that the sub-operations are ordered (by semicolon). Were one to try to record specifications of the entire *Read* and *Write* operations, they would be festooned with implications. The structure of the program (e.g. that *Write* cannot interfere with *Write*) simplifies the specifications of the sub-operations.

2.4. Proofs

Even on a specification, there are proof obligations: notably involving $inv\text{-}\Sigma^a$.
Initial state satisfies invariant:
$inv\text{-}\Sigma^a(\sigma_0^a)$ is immediate

Preservation of $inv\text{-}\Sigma^a$ by each operation:
The argument needs a form of "dynamic invariant":

$$dinv\text{-}\Sigma^a : \Sigma^a \times \Sigma^a \to \mathbb{B}$$

$$dinv\text{-}\Sigma^a(mk\text{-}\Sigma^a(d\text{-}w, fr\text{-}w, ho\text{-}r), mk\text{-}\Sigma^a(d\text{-}w', fr\text{-}w', ho\text{-}r')) \quad \triangleq \quad fr\text{-}w \le fr\text{-}w'$$

We need to prove that each operation preserves the invariant; we simultaneously cover the dynamic invariant.

$$\forall \overleftarrow{\sigma^a} \in \Sigma^a \cdot post\text{-}start\text{-}Write^a(\overleftarrow{\sigma^a}, in, \sigma^a) \;\Rightarrow\; dinv\text{-}\Sigma^a(\overleftarrow{\sigma^a}, \sigma^a) \wedge \sigma^a \in \Sigma^a$$

is immediate.

$$\forall \overleftarrow{\sigma^a} \in \Sigma^a \cdot post\text{-}commit\text{-}Write^a(\overleftarrow{\sigma^a}, \sigma^a) \;\Rightarrow\; dinv\text{-}\Sigma^a(\overleftarrow{\sigma^a}, \sigma^a) \wedge \sigma^a \in \Sigma^a$$

needs information from the invariant to establish the dynamic invariant.

$$\forall \overleftarrow{\sigma^a} \in \Sigma^a \cdot post\text{-}start\text{-}Read^a(\overleftarrow{\sigma^a}, \sigma^a) \;\Rightarrow\; dinv\text{-}\Sigma^a(\overleftarrow{\sigma^a}, \sigma^a) \wedge \sigma^a \in \Sigma^a$$

is immediate.

$$\forall \overleftarrow{\sigma^a} \in \Sigma^a \cdot post\text{-}end\text{-}Read^a(\overleftarrow{\sigma^a}, \sigma^a) \;\Rightarrow\; dinv\text{-}\Sigma^a(\overleftarrow{\sigma^a}, \sigma^a) \wedge \sigma^a \in \Sigma^a$$

is trivial since $\sigma^a = \overleftarrow{\sigma^a}$.

2.5. Taking stock

This section has established a clear specification in Figure 5; the non-determinism allowed is important in that it reflects details of timing. The rely/guarantee conditions are an essential aspect of the specification because they ensure mutual exclusion. That having been said, all of the *development* work remains to be done. In particular, it is clear that the atomicity considerations have to be relaxed — it is our claim that the key to splitting the atomicity constraints is data reification. In fact, finding a more economical representation is the next task.

3. Reusing cells without clashing

It should be obvious that the behaviour of the algorithm does not depend on retaining all of the values in *data-w*. This step of development introduces a potential reduction in the number of *Value*s retained by storing them in a mapping indexed by an arbitrary set X. The essence of this step is to show "ownership" of the

```
$Write(v: Value)$
  owns wr $data\text{-}w, fresh\text{-}w, hold\text{-}w$
  $start\text{-}Write(v: Value)$
    $hold\text{-}w :\in (X - \{hold\text{-}r, fresh\text{-}w\})$;
    $data\text{-}w(hold\text{-}w) \leftarrow v$
  $commit\text{-}Write()$
    $fresh\text{-}w \leftarrow hold\text{-}w$

$Read()r: Value$
  owns wr $hold\text{-}r$
  $start\text{-}Read()$
    $hold\text{-}r \leftarrow fresh\text{-}w$
  $end\text{-}Read()r: Value$
    $r \leftarrow data\text{-}w(hold\text{-}r)$
```

Fig. 6. Intermediate pseudo-code

indices (in X); in particular, that an element of *data-w* whose index could be used by *Read* is not overwritten. Essentially, a careful data reification step is bringing in some of the design decisions without going all the way to Simpson's code. Rely/guarantee conditions are again used to investigate the requirements.

The state used throughout this section is:

$$\begin{array}{lll} \Sigma^i :: & data\text{-}w & : X \xrightarrow{m} Value \\ & fresh\text{-}w & : X \\ & hold\text{-}r & : X \\ & hold\text{-}w & : X \end{array}$$

$$\textbf{inv}\ (mk\text{-}\Sigma^i(d\text{-}w, fr\text{-}w, ho\text{-}r, ho\text{-}w)) \quad \triangleq \quad \{fr\text{-}w, ho\text{-}r, ho\text{-}w\} \subseteq \textbf{dom}\ d\text{-}w$$

$$\textbf{init}\ mk\text{-}\Sigma^i(\{\alpha \mapsto \mathtt{x}\}, \alpha, \alpha, \alpha)$$

Note that *hold-w* is strictly local (not even accessed by *Read*). We did consider using a special notation for **local** *hold-w* but it actually saves little.

3.1. Intuition from pseudo-code

It is again possible to use pseudo-code to convey the intuition of what has to be given below as a formal specification — this is presented in Figure 6. Notice that the *Write* process now needs to access *hold-r*: essentially, this means that the *Read* process can communicate the fact that a certain cell must not be destroyed or corrupted.

3.2. Specifications of the sub-operations on Σ^i

The specifications of the four sub-operations are shown in Figure 7. There is masses of non-determinism here — in fact, one valid implementation is to have $X = \mathbb{N}_1$ and retain the whole sequence as in Section 2 — but it is shown below that **card** X must be at least 3 (cf. *start-Write*) — we return to this point in Section 3.4.

The *Write* process in Section 2.3 avoided destroying (or even worse, corrupting) any *Value* required by *Read* by concatenating new values to the end of *data-w* (and only exposing them in *commit-Write*) — we now need to be more explicit about setting *hold-w* (R/G to the rescue).

The post condition of *start-Write* clearly shows that we need at least three slots in order to avoid "race conditions" on individual *Value*s.

The property in *rely-start-Write* (mirrored in *guar-Start-Read*) ensures that there are only two possible values during any single execution of *start-Write*.

Notice that –in contrast to the situation with $\widehat{fresh\text{-}w}$ in Section 2.3– here, we can only talk about the possible values of *fresh-w* in *start-Read* by use of the new notation (or by adding auxiliary variables).

```
$Write(v\colon Value)$
**owns wr** $data\text{-}w, fresh\text{-}w, hold\text{-}w$
   $start\text{-}Write(v\colon Value)$
      **wr** $data\text{-}w, hold\text{-}w$
      **rd** $hold\text{-}r, fresh\text{-}w$
      **rely** $hold\text{-}r \neq \overleftarrow{hold\text{-}r} \;\Rightarrow\; hold\text{-}r = fresh\text{-}w$
      **guar** $\wideparen{hold\text{-}r} \lhd data\text{-}w = \wideparen{hold\text{-}r} \lhd \overleftarrow{data\text{-}w}$
      **post** $hold\text{-}w \notin \{\overleftarrow{hold\text{-}r}, fresh\text{-}w\} \wedge data\text{-}w = \overleftarrow{data\text{-}w} \dagger \{hold\text{-}w \mapsto v\}$
   $commit\text{-}Write()$
      **wr** $fresh\text{-}w$
      **rd** $hold\text{-}w$
      **post** $fresh\text{-}w = hold\text{-}w$

$Read()r\colon Value$
**owns wr** $hold\text{-}r$
   $start\text{-}Read()$
      **wr** $hold\text{-}r$
      **rd** $fresh\text{-}w$
      **guar** $hold\text{-}r \neq \overleftarrow{hold\text{-}r} \;\Rightarrow\; hold\text{-}r = fresh\text{-}w$
      **post** $hold\text{-}r \in \wideparen{fresh\text{-}w}$
   $end\text{-}Read()r\colon Value$
      **rd** $data\text{-}w, hold\text{-}r$
      **rely** $data\text{-}w(hold\text{-}r) = \overleftarrow{data\text{-}w}(hold\text{-}r)$
      **post** $r = data\text{-}w(hold\text{-}r)$
```

Fig. 7. Rely/guarantee specifications on Σ^i

3.3. Proofs

The initial state satisfies invariant:
$inv\text{-}\Sigma^i(\sigma_0^i)$ is immediate

Preservation of $inv\text{-}\Sigma^i$ by each operation:

$$\forall \overleftarrow{\sigma^i} \in \Sigma^i \cdot post\text{-}start\text{-}Write^i(\overleftarrow{\sigma^i}, v, \sigma^i) \;\Rightarrow\; \sigma^i \in \Sigma^i$$

First, *start-Write* cannot reduce **dom** *data-w* (so both *fresh-w* and *hold-r* will be in the domain of the resulting *data-w*); furthermore *hold-w* is of type X and is clearly in **dom** $(\overleftarrow{data\text{-}w} \dagger \{hold\text{-}w\} \mapsto v)$.

$$\forall \overleftarrow{\sigma^i} \in \Sigma^i \cdot post\text{-}commit\text{-}Write^i(\overleftarrow{\sigma^i}, \sigma^i) \;\Rightarrow\; \sigma^i \in \Sigma^i$$

is immediate because *hold-w* was already in **dom** *data-w*.

$$\forall \overleftarrow{\sigma^i} \in \Sigma^i \cdot post\text{-}start\text{-}Read^i(\overleftarrow{\sigma^i}, \sigma^i) \;\Rightarrow\; \sigma^i \in \Sigma^i$$

is immediate because *fresh-w* was already in **dom** *data-w*.

$$\forall \overleftarrow{\sigma^i} \in \Sigma^i \cdot post\text{-}end\text{-}Read^i(\overleftarrow{\sigma^i}, \sigma^i) \;\Rightarrow\; \sigma^i \in \Sigma^i$$

is trivial since $\sigma^i = \overleftarrow{\sigma^i}$.

Respecting rely-conditions (by pairs of operations):

The obvious case is to show that *rely-end-Read* is OK: this follows immediately from *guar-start-Write*.

In fact, the more interesting case is *rely-start-Write*: (*end-Read* does not change any of the relevant variables, so) *guar-start-Read* has to imply *rely-start-Write* which is immediate from their texts and the fact that *fresh-w* cannot change during the execution of *start-Write*.

Reification:
This is a classic situation where the standard VDM "homomorphic" data reification rule does not suffice and Nipkow's version (cf. Section 1.1) is needed to show that the data absent in the representation was not actually essential in the specification.[8]

The key here is to show that any required *Value*s are contained in the smaller $data\text{-}w^i$; this is done by checking that a mapping m exists between the available X indices and the $\mathbb{N}_1$ indices to the full list in $data\text{-}w^a$. So:

$$rel : \Sigma^a \times \Sigma^i \to \mathbb{B}$$

$$\begin{array}{l} rel(mk\text{-}\Sigma^a(d\text{-}w^a, fr\text{-}w^a, ho\text{-}r^a), mk\text{-}\Sigma^i(d\text{-}w^i, fr\text{-}w^i, ho\text{-}r^i, ho\text{-}w^i)) \quad \triangleq \\ \quad \exists m \in (X \xleftrightarrow{m} \mathbb{N}_1) \cdot \\ \quad\quad d\text{-}w^i \subseteq m \circ d\text{-}w^a \wedge \\ \quad\quad m(fr\text{-}w^i) = fr\text{-}w^a \wedge m(ho\text{-}r^i) = ho\text{-}r^a \end{array}$$

The initial states relate:
$rel(\sigma_0^a, \sigma_0^i)$ is immediate with $m = \{\alpha \mapsto 1\}$

It is then necessary to show that each pair of operations preserve this relation. It is often a disadvantage of "Nipkow's rule" that it requires an existential proof; in general, such existence proofs can be troublesome but in this case the Σ^a operations are simple enough (two are deterministic) that it is easy to spot the witness for σ_2^a.

The pair of *start-Write* operations preserve the relation:
The only one where the existence of the new mapping m requires work is:

$$rel(\sigma_1^a, \sigma_1^i) \wedge post\text{-}start\text{-}Write^i(\sigma_1^i, v, \sigma_2^i) \;\Rightarrow\; \exists \sigma_2^a \in \Sigma^a \cdot post\text{-}start\text{-}Write^a(\sigma_1^a, v, \sigma_2^a) \wedge rel(\sigma_2^a, \sigma_2^i)$$

From $rel(\sigma_1^a, \sigma_1^i)$ we have:

$$\begin{array}{l} \exists m_1 \in (X \xleftrightarrow{m} \mathbb{N}_1) \cdot \\ \quad d\text{-}w_1^i \subseteq m_1 \circ d\text{-}w_1^a \wedge \\ \quad m_1(fr\text{-}w_1^i) = fr\text{-}w_1^a \wedge m_1(ho\text{-}r_1^i) = ho\text{-}r_1^a \end{array}$$

Then (as mentioned) $post\text{-}start\text{-}Write^a(\sigma_1^a, v, \sigma_2^a)$ determines σ_2^a to have:

$$data\text{-}w_2^a = data\text{-}w_1^a \curvearrowright [v]$$

Then $rel(\sigma_2^a, \sigma_2^i)$ follows from:

$$m_2 = m_1 \dagger \{ho\text{-}w^i \mapsto \mathbf{len}\, data\text{-}w_2^a\}$$

because the type of $ho\text{-}w \in X$ gives the first property; the pairing $ho\text{-}w \mapsto \mathbf{len}\, data\text{-}w_2^a$ ensures $d\text{-}w_2^i \subseteq m_2 \circ d\text{-}w_2^a$; and $ho\text{-}w_2^i \notin \{fr\text{-}w_1^i, ho\text{-}r_1^i\}$ (from $post\text{-}start\text{-}Write^i$) shows the last two requirements on m are satisfied.

The pair of *commit-Write* operations preserve the relation:

$$rel(\sigma_1^a, \sigma_1^i) \wedge post\text{-}commit\text{-}Write^i(\sigma_1^i, \sigma_2^i) \;\Rightarrow\; \exists \sigma_2^a \in \Sigma^a \cdot post\text{-}commit\text{-}Write^a(\sigma_1^a, \sigma_2^a) \wedge rel(\sigma_2^a, \sigma_2^i)$$

[8] Note that no adequacy proof is required.

Here, $m_2 = m_1$.

The pair of *start-Read* operations preserve the relation:

$$rel(\sigma_1^a, \sigma_1^i) \wedge post\text{-}start\text{-}Read^i(\sigma_1^i, \sigma_2^i) \;\Rightarrow\; \exists \sigma_2^a \in \Sigma^a \cdot post\text{-}start\text{-}Read^a(\sigma_1^a, \sigma_2^a) \wedge rel(\sigma_2^a, \sigma_2^i)$$

is immediate with $m_2 = m_1$ and the post-conditions both copying *fresh-w* into *hold-r*.

The pair of *end-Read* operations preserve the relation:

$$rel(\sigma_1^a, \sigma_1^i) \wedge post\text{-}end\text{-}Read^i(\sigma_1^i, \sigma_2^i, v) \;\Rightarrow\; \exists \sigma_2^a \in \Sigma^a \cdot post\text{-}end\text{-}Read^a(\sigma_1^a, \sigma_2^a, v) \wedge rel(\sigma_2^a, \sigma_2^i)$$

again is immediate with $m_2 = m_1$; in fact, the only interest here is to see that the result v is the same in each case.

3.4. Taking stock again

This section has made progress in that the specification in Figure 7 offers a way to retain only a small number of *Value*s in *data-w*^{i}; but in resolving the atomicity of *data-w*, we have actually introduced new atomicity problems! Recall that in Section 2 it was made clear that –in the final implementation– atomicity is only to be assumed at the level of single bit operations (not even a pair of bits can be accessed and changed atomically).

To be precise, the only requirement taken forward is that **card** $X \geq 3$ — but Section 4 shows that Simpson needs four slots precisely to facilitate communication. We need a way of communicating *hold-r* without assuming that we can assign values of type X atomically otherwise we might have a problem as big as the initial transfer of *Values*. Again, choosing the right representation is the key to achieving the guarantee conditions.

4. The four-slot representation

Section 3 reduces the number of *Value*s that have to be retained. More importantly, it reduces the atomicity requirements providing the fields of *data-w* can be separately accessible. This leaves the issue of atomic operations on the shared variables *fresh-w* and *hold-r* (*hold-w* is not shared). Essentially, this section shows how to encode the "ownership" from the Σ^i level without atomicity assumptions on *hold-r* and *fresh-w*. There is, in fact, a clue to how this can be done in that so far we have only established the need for three distinct places in *data-w* — maybe a couple of bits suffice. But it is part of the atomicity objective of the whole design process that one cannot even "lock" two bits: even this could delay the sibling process.

Simpson's contribution is not, in fact, realising a minimal number of slots but in finding a way to communicate between *Read* and *Write* assuming *only* single bit operations avoid corruption.

4.1. The code

The state of the implementation can be defined as in Figure 8. Although they play no real part in this development, Simpson's terms "pair" and "slot" are used here. This final state introduces local variables for slot and pair information: in the *Write* process these are *wp-w* and *ws-w*; in *Read*, *pair-r* and *rs-r*. All but *pair-r* are strictly local (not even visible to the other process).[9] Notice that viewing pair/slot as the model of X gives **card** $X = 4$. (Also, anywhere in the proofs, we can use, for a field *any* of type P or S, **card** $\widehat{any} \leq 2$.)

The code for Simpson's algorithm is given in Figure 9. For convenience of comparison with earlier papers [Sim97, Hen04], comments are added to the code.

Two pairs of statements (in *commit-Write* and *start-Read*) are marked as being executed atomically for now: this requirement is lifted in Section 4.3. The reason for the temporary assumption is that –in Figure 7– the behaviour of *fresh-w*^{i} is clearly atomic whereas its representation here as a P/S pair could introduce new behaviours. As becomes clear below, these are avoided by a standard concurrent programming technique.

[9] This fits with the locality of *hold-w* in Σ^a.

$$
\begin{array}{lll}
\Sigma^r :: & \textit{data-w} & : P \times S \xrightarrow{m} [\textit{Value}] \\
 & \textit{pair-w} & : P \\
 & \textit{pair-r} & : P \\
 & \textit{slot-w} & : P \xrightarrow{m} S \\
 & \textit{wp-w} & : P \\
 & \textit{ws-w} & : S \\
 & \textit{rs-r} & : S
\end{array}
$$

$$
\begin{array}{l}
\textbf{inv } (mk\text{-}\Sigma^r(\textit{data-w}, \textit{pair-w}, \textit{pair-r}, \textit{slot-w}, \textit{wp-w}, \textit{ws-w}, \textit{rs-r})) \quad \triangleq \\
\quad \textbf{card dom } \textit{data-w} = 4 \wedge \\
\quad \textbf{card dom } \textit{slot-w} = 2
\end{array}
$$

$$
\begin{array}{l}
\textbf{init let } \textit{data-w} = \{(p_0, s_0) \mapsto \mathtt{x}, (p_0, s_1) \mapsto \textbf{nil}, (p_1, s_0) \mapsto \textbf{nil}, (p_1, s_1) \mapsto \textbf{nil}\} \\
\qquad\quad \textit{pair-w} = p_0 \\
\qquad\quad \textit{pair-r} = p_0 \\
\qquad\quad \textit{slot-w} = \{p_0 \mapsto s_0, p_1 \mapsto s_0\} \\
\qquad\quad \textit{wp-w} = p_0 \\
\qquad\quad \textit{ws-w} = s_0 \\
\qquad\quad \textit{rs-r} = s_0 \textbf{ in} \\
\quad mk\text{-}\Sigma^r(\textit{data-w}, \textit{pair-w}, \textit{pair-r}, \textit{slot-w}, \textit{wp-w}, \textit{ws-w}, \textit{rs-r})
\end{array}
$$

Where (**card** P = **card** $S = 2$):

$P, S = \textit{token}\textbf{-set}$

Fig. 8. The final state: Σ^r

```
Write(v: Value)
owns wr data-w, pair-w, slot-w, wp-w, ws-w
  start-Write(v: Value)
    wp-w ← ρ(pair-r);                   writer chooses pair
    ws-w ← ρ(slot-w(wp-w));             writer chooses slot
    data-w(wp-w, ws-w) ← v;
  commit-Write()
    < slot-w(wp-w) ← ws-w;              writer declares slot
    pair-w ← wp-w >                     writer declares pair

Read()r: Value
owns wr pair-r, rs-r
  start-Read()
    < pair-r ← pair-w;                  reader chooses (and declares) pair
    rs-r ← slot-w(pair-r) >;            reader chooses slot
  end-Read()r: Value
    r ← data-w(pair-r, rs-r)
```

Fig. 9. Code for Simpson's algorithm

4.2. Correctness of the code

Initial state satisfies invariant:

$inv\text{-}\Sigma^r(\sigma_0^r)$:
  is immediate[10]

Code (with atomicity) satisfies specs of Section 3.2

[10] There is a small issue here which different authors circumvent in various ways: several authors put the initial `x` value in all four slots; we prefer to view $\textit{data-w}^i = \textit{data-w}^r \rhd \{\textbf{nil}\}$.

Preservation of $inv\text{-}\Sigma^r$:
With the interpretation:

$fresh\text{-}w = (pair\text{-}w, slot\text{-}w(pair\text{-}w))$
$hold\text{-}r = (pair\text{-}r, rs\text{-}r)$
$hold\text{-}w = (wp\text{-}w, ws\text{-}w)$

Re *post-start-Write*
There are essentially three clauses:
1) $hold\text{-}w \neq fresh\text{-}w$
even if $pair\text{-}r = pair\text{-}w$, $ws\text{-}w = \rho(slot\text{-}w(pair\text{-}w))$ ensures $(wp\text{-}w, ws\text{-}w) \neq (pair\text{-}w, slot\text{-}w(pair\text{-}w))$
note that all variables with names $\alpha\text{-}w$ cannot change by interference.
2) $hold\text{-}w \neq hold\text{-}r$
Since $wp\text{-}w = \rho(pair\text{-}r)$, it follows that $(wp\text{-}w, ws\text{-}w) \neq (pair\text{-}r, rs\text{-}r)$
3) Finally,
$data\text{-}w = \overleftarrow{data\text{-}w} \dagger \{hold\text{-}w \mapsto v\}$
is immediate.

Re *guar-start-Write*
The code only changes $data\text{-}w(hold\text{-}w)$
rely-start-Write gives $\widehat{hold\text{-}r} = \{\overleftarrow{hold\text{-}r}, fresh\text{-}w\}$
by the same argument as above, $hold\text{-}w \notin \{\overleftarrow{hold\text{-}r}, fresh\text{-}w\}$

Re *post-commit-Write*
$hold\text{-}w = (wp\text{-}w, ws\text{-}w)$ and $fresh\text{-}w = (pair\text{-}w, slot\text{-}w(pair\text{-}w))$
so the result is immediate (but splitting the atoms is discussed in Section 4.3).

Re *post-start-Read*
$fresh\text{-}w = (pair\text{-}w, slot\text{-}w(pair\text{-}w))$ and $hold\text{-}r = (pair\text{-}r, rs\text{-}r)$
give the exact result (but again splitting the atoms has to be discussed in Section 4.3).

Re *guar-start-Read*
is essentially the same argument.

Re *post-end-Read*
follows immediately from $hold\text{-}r = (pair\text{-}r, rs\text{-}r)$

4.3. Final atomicity refinement

The code in Figure 9 has atomic brackets around two pairs of statements: as far as *start-Read* is concerned, while these pairs of statements are linked, there are only two possible behaviours: either $hold\text{-}r = \overleftarrow{fresh\text{-}w}$ or $hold\text{-}r = fresh\text{-}w$. Allowing the steps of the atomic statements in *commit-Write* and *start-Read* to interleave admits no new behaviours. But it is *crucial* that the *slot-w* and *pair-r* are set (read) in *commit-Write* (*start-Read*) in the reverse order: this gives the impression of "atomicity". This is, of course, a standard technique from database locking [WV01, §4] (for an attempt to link views of different communities about "atomicity", see [JLRW05]).

Many authors choose to present the code of Figure 9 above with an additional variable *wp-r* and write *start-Read* is to write
$rp\text{-}r \leftarrow pair\text{-}w$;
$pair\text{-}r \leftarrow rp\text{-}r$
instead of
$pair\text{-}r \leftarrow pair\text{-}w$;
This is not done here since we do not assume assignment statements are executed atomically. But, if they did execute atomically, the use of the extra *wp-r* admits more behaviours. This observation just goes to emphasise the extreme interference/interleaving being considered in ACM implementation.

It is worth emphasising that the residual assumptions on "atomicity" are only at the bit level: any assignment has only one shared variable and affects only a single bit. For a discussion of "meta-stability" at the bit level, see [PHA04].

5. Generality of the techniques

This section pulls out the ideas which –in various subsets/combinations– should be useful in other developments.

In the ACM example, the ideal of the "fiction of atomicity" would be to abstract from all of the details by using a single atomically accessed variable as an abstraction but this does not describe all of the possible behaviours and one has to think harder to obtain a starting specification. The choice here is to make a minimal split of the two parallel processes each into two sub-operations whose behaviour is composed sequentially ("by semicolon"). This "phasing" is of course algorithmic detail in a specification but is claimed to offer a reasonably intuitive description of the permissable behaviours of an ACM. The general suggestion of *tasteful* use of algorithmic operators in specifications is a useful message.

The same phasing idea pays off handsomely when the move is made to specifications with rely and guarantee conditions: if the same essential properties were to be presented for the whole of say *Write* in the ACM example, there would have to be ghost variables to track the phase and implications to present the information about the separate phases as a single predicate. The current authors believe that phasing is a useful specification idea that is explored further in the second author's PhD thesis [Pie09].

In Section 2, the rely and guarantee conditions themselves are fairly standard. Checking that they are consistent between the two parallel threads is made almost trivial by judicious choice of frame markings. Such frame markings are another useful technique familiar from writings on VDM but with additional payoff in concurrency.

The notation for "possible values" is new in this paper and warrants further exploration and exploitation. This links with how mutual exclusion is handled in Section 3 at the abstract level (Σ^i); this is in contrast to the auxiliary variable argument in [Pie09]. This issue is discussed further in [Jon10].

The justification of the data reification from Σ^a to Σ^i cannot be done using the simpler of the two rules in the VDM literature but the rule from Nipkow's thesis covers the (possible) reduction in the size of the state space and this rule is included in VDM: e.g. [Jon90, §9.3]. The use in Section 3 is technically interesting; in fact, its availability makes possible the choice of development from Σ^a to Σ^r via Σ^i. Such careful choice of design strategy is essential but is perhaps the hardest of the techniques to reduce to general rules.

Another key point only sees its completion in Section 4 and that is the use even at this step of rather bold atomicity assumptions. Without Simpson's clever data representation it might be impossible to achieve atomic update (on a reasonable machine architecture) without locking and it is made clear in Section 2 that this is not allowed in ACMs. Such roadblocks (leading to backtracking) cannot be ruled out by any method whether formal or informal. The general observation that data reification has a key role to play in "atomicity refinement" is made in Section 1.3.

There is a danger when presenting such a development that a reader will conclude that a claim is being made that the "method" can never lead down false paths. This is certainly not the claim of the current authors. For example, without the clever choice of data representation, the guarantee conditions on Σ^i cannot be met within the atomicity assumptions. What *is* claimed here is (only) that the design decisions can be seen more clearly in a reification process than by staring at the final code.

6. Discussion

This section offers brief descriptions of some other recent justifications of Simpson's algorithm. In making such comparisons, the current authors are not trying to be competitive but to use this intricate algorithm to indicate what insight can be given by various approaches. No attempt is made here to trace the full history of the algorithm that we –in common with most authors– have referred to as "Simpson's algorithm". The interested reader could start at [Pet83a, Pet83b] and certainly read some of Leslie Lamport's many contributions such as [Lam86].

6.1. Henderson's development

Henderson's research (in particular, his thesis [Hen04][11]) has been a key information source. Interestingly, he uses broadly the same set of technical tools as in the current paper. In spite of this, the presentation here looks very different.

First, Henderson's specification attempts to retain a minimal list of *Value*s that could potentially be returned by a *Read*. A cost for this is a pair of "ghost variables" that inform the *Read* operation in which phase the *Write* operation is executing (and *vice versa*). These variables can be eliminated in reification because Henderson also uses "Nipkow's rule". The current authors hold the (biased) view that the specification here is clearer but there would be little difficulty in proving they describe the same behaviour and the choice can be left to the "customer".

A more pervasive difference results in part from the recent development (cf. [Jon07]) of the link between atomicity refinement and data reification. In Section 4 of the current paper, the preceding interference specifications are achieved by capitalising on Simpson's four-slot representation.

6.2. Event decomposition

Jean-Raymond Abrial's extension of his "B" approach [Abr96] to "Event-B" is described in [AC05]. Guarded events are assumed to be executed atomically; selection as to which event can be executed is non-deterministic if multiple guards evaluate to **true**. As such, this approach is completely different from that of rely/guarantee thinking. The approach in [AC05] to increasing concurrency (or "splitting atoms") is to decompose events. When one "splits" an event into sub-events it has to be shown that all but one "refines skip". There are a number of elegant examples of the use of this approach.

Abrial and Cansell have also tackled the "4-slot" implementation of ACMs and have been kind enough to let us see their development as supported by the RODIN tools [Rod08]. They start from a specification in terms of the traces of reading and writing. It is inherent in the ACM problem –rather than a criticism of their specification– that pinning down the exact behaviour is somewhat messy: in essence, they have to reflect the points at which operations start and end. It would be possible to relate the initial specification in Section 2.3 to their specification and prove that the same invariants are satisfied. This then leaves the user to decide which is the most intuitive way of understanding ACM behaviour.

The "event decomposition" method is extremely interesting: Abrial and Cansell avoid the need for rely and guarantee conditions by preserving the atomicity of events at any level of development. This achieves a considerable economy of rules. The use of pseudo instruction counters is vividly illustrated in Abrial's event refinement approach. In those situations where the correctness depends on a constrained order, since the order of execution of the events with true guards in a given set is non-deterministic, pseudo instruction counters are tested in guards and set in the corresponding events.[12]

The current authors do also wonder whether the interesting development of Simpson's algorithm in [AC08] indicates that the atomicity constraint might require a series of difficult-to-invent steps. But their forthcoming publication will admit wider comparison (and by people unbiased by being authors of either approach).

6.3. Comparison with "Separation Logic"

Another exciting avenue in research on concurrent code has been the recent developments around "concurrent separation logic" [Rey02, O'H07, Bro07, OYR09, PB05]. At this time, researchers in Newcastle, London and Cambridge are discussing ways of combining the best features of both separation logic and rely/guarantee reasoning. For example, the second author's thesis builds the bridge with the read/write frames here. There is not space here to do this research full justice; but an excellent recent reference (from which other citations can be found) is [Vaf07].

During the writing of this paper, Richard Bornat sent us his current work on Simpson's algorithm using

[11] The reader is also referred to [HP02] and [PHA04]; the second of these addresses the delicate issue of "meta-stability" of the control bits.
[12] This is reminiscent of the proof of the Boehm/Jacopini theorem that "goto" statements can be avoided.

separation logic. The title of [BA08] alone should indicate why this is exciting. Again, the availability of this in published form will admit proper unbiased comparison.

There is a sense in which the dynamic ownership (by the two processes) of the indices of $data\text{-}w^i$ ought be made for reasoning with concurrent separation logic. As far as the current authors can determine, no paper has fully exploited this observation. In contrast, the approach here is to show this as a representation of a carefully thought out abstraction. This distinction goes to the heart of John Reynold's comment at MFPS (Birmingham, OK, 2005) that "separation logic lets one show avoidance of races and rely/guarantee facilitates reasoning about races" (this is only an approximate quote — sadly, John has not put it in a published paper). This aspect is the subject of on-going discussions between the first author and Matt Parkinson.

6.4. Model checking

Between the initial submission to this journal and the final revision of the paper, its first author has discussed the specific application with Bill Roscoe. His [Ros10] and the earlier [Rus02] certainly provide insight into aspects of Simpson's algorithm. In particular, the conditions under which one can even relax the atomicity constraints on the control bits are interesting. Although the current authors prefer a developmental presentation, it is clear that:

- this algorithm uses (basically) finite data structures and lends itself to model checking
- model checking is a good tool to investigate code
- this is another example of diverse formal approaches providing different insight

6.5. Our own path

The current paper is an extensive revision of [JP08]. Already in that paper the role of the intermediate abstraction (Σ^i) is clear. There was however an error in the description at that level of abstraction that is corrected here by the use of the new notation for "possible values".

More importantly, the step from Σ^i to Σ^r is completely different here making much more use of the results at the Σ^i level than in [JP08]. The second author's thesis [Pie09] illustrates an approach to showing mutual exclusion on $data\text{-}w^r$ using auxiliary variables. The difference between Pierce's solo argument and that here reflects Jones' strong preference for arguing via abstraction rather than backwards from code.

Another interesting insight into the evolution of Jones' thinking is that the paper [Jon10] was written between [JP08] and the current paper.

7. Conclusions

As made clear at the outset, ACMs are complex; Simpson's algorithm is ingenious; and its correctness requires delicate reasoning. The development in Figures 5, 7 and 9 is key to providing an intuitive grasp of the correctness. The authors hope that the reader finds this a clear design rationale. (The material in Figures 1 and 6 is really there to provide intuition about the behaviour.)

The intention, however, was not just to add yet another correctness argument of one specific algorithm but instead to use this example to illustrate how a number of ideas can be used in concert to move from a "fiction of atomicity" using a development approach that can be called "splitting (software) atoms safely". The notes in Section 5 can be summarised as:

- The authors present an understandable and tractable reworking of the "4-slot" algorithm, with a clear design history.
- The "fiction of atomicity" is a good place to begin.
- Rely/guarantee reasoning is greatly simplified by the use of frames and phasing arguments.
- While rely/guarantee conditions allow us to reason about the interference, a clever data reification is required (which Simpson gives us).

Acknowledgements

The authors are grateful to Jean-Raymond Abrial and Dominique Cansell for sharing their ongoing work in this area. Similarly, the preview of the paper by Richard Bornat and Hasan Amjad is gratefully acknowledged. Thanks also go to Peter O'Hearn, Hongseok Yang, Viktor Vafeiadis, Mike Dodds and Matt Parkinson for general and ongoing discussions on development methods for concurrency. These exchanges have been amplified by the London concurrency meeting in January 2009, its Northern edition in the same format in November 2009 and are to be continued in Cambridge.

The authors are grateful for the reviewers comments obtained anonymously by the journal.

Of course, our original inspiration of the specific algorithm comes from Hugo Simpson. Neil Henderson and Steve Paynter made us aware of the challenge of this tiny but intriguing problem.

Our research is supported by the EPSRC Platform Grant on "Trustworthy Ambient Systems" and the EU FP7 "DEPLOY project".

References

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

[AC05] Jean-Raymond Abrial and Dominique Cansell. Formal construction of a non-blocking concurrent queue algorithm. *Journal of Universal Computer Science*, 11(5):744–770, 2005.

[AC08] Jean-Raymond Abrial and Dominique Cansell. Development of a concurrent program, 2008. private communication.

[BA08] Richard Bornat and Hasan Amjad. Inter-process buffers in separation logic with rely-guarantee, 2008. (private communication) Submitted to Formal Aspects of Computing.

[Bro07] S. D. Brookes. A semantics of concurrent separation logic. *Theoretical Computer Science (Reynolds Festschrift)*, 375(1-3):227–270, 2007. (Preliminary version appeared in CONCUR'04, LNCS 3170, pp16-34).

[CJ07] J. W. Coleman and C. B. Jones. A structural proof of the soundness of rely/guarantee rules. *Journal of Logic and Computation*, 17(4):807–841, 2007.

[dR01] W. P. de Roever. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press, 2001.

[dRE99] W. P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and Their Comparison*. Cambridge University Press, 1999.

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

[HP02] N. Henderson and S. E. Paynter. The formal classification and verification of Simpson's 4-slot asynchronous communication mechanism. In L.-H. Eriksson and P.A Lindsay, editors, *FME 2002*, volume 2391 of *Lecture Notes in Computer Science*, pages 350–369. Springer Verlag, 2002.

[JLRW05] C. B. Jones, D. Lomet, A. Romanovsky, and G. Weikum. The atomic manifesto. *Journal of Universal Computer Science*, 11(5):636–650, 2005.

[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.

[Jon83a] C. B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332. North-Holland, 1983.

[Jon83b] C. B. Jones. Tentative steps toward a development method for interfering programs. *Transactions on Programming Languages and System*, 5(4):596–619, 1983.

[Jon89] C. B. Jones. Data reification. In J. A. McDermid, editor, *The Theory and Practice of Refinement*, pages 79–89. Butterworths, 1989.

[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.

[Jon03] C. B. Jones. Wanted: a compositional approach to concurrency. In Annabelle McIver and Carroll Morgan, editors, *Programming Methodology*, pages 1–15. Springer Verlag, 2003.

[Jon07] C. B. Jones. Splitting atoms safely. *Theoretical Computer Science*, 357:109–119, 2007.

[Jon10] Cliff B. Jones. The role of auxiliary variables in the formal development of concurrent programs. In Cliff Jones and Bill Roscoe, editors, *Reflections on the work of C. A. R. Hoare*. Springer, 2010. in press.

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

[Lam86] Leslie Lamport. The mutual exclusion problem: part i—a theory of interprocess communication. *J. ACM*, 33(2):313–326, 1986.

[Nip86] T. Nipkow. Non-deterministic data types: Models and implementations. *Acta Informatica*, 22:629–661, 1986.

[Nip87] T. Nipkow. *Behavioural Implementation Concepts for Nondeterministic Data Types*. PhD thesis, University of Manchester, May 1987.

[O'H07] P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science (Reynolds Festschrift)*, 375(1-3):271–307, May 2007. Preliminary version appeared in CONCUR'04, LNCS 3170, 49–67.

[OYR09] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. *ACM TOPLAS*, 31(3), April 2009. Preliminary version appeared in 31st POPL, pp268-280, 2004.

[PB05] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 247–258, New York, NY, USA, 2005. ACM.

[Pet83a] Gary L. Peterson. Concurrent reading while writing. *ACM Trans. Program. Lang. Syst.*, 5(1):46–55, 1983.

[Pet83b] Gary L. Peterson. A new solution to lamport's concurrent programming problem using small shared variables. *ACM Trans. Program. Lang. Syst.*, 5(1):56–65, 1983.

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

[Pie09] Ken Pierce. *Enhancing the Useability of Rely-Guaranteee Conditions for Atomicity Refinement.* PhD thesis, Newcastle University, 2009.

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

[Rod08] Rodin. Rodin tools can be downloaded from SourceForge, 2008. http://sourceforge.net/projects/rodin-b-sharp/.

[Ros10] A. W. Roscoe. *Understanding Concurrent Systems.* Springer, 2010.

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

[Sim97] H. R. Simpson. New algorithms for asynchronous communication. *IEE, Proceedings of Computer Digital Technology*, 144(4):227–231, 1997.

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

[WV01] Gerhard Weikum and Gottfried Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery.* Morgan Kaufmann Publishers Inc., 2001.