COMPUTING SCIENCE

Splitting Atoms with Rely/Guarantee Conditions Coupled with Data Reification

Cliff B. Jones and Ken G. Pierce

TECHNICAL REPORT SERIES

No. CS-TR-1186 **January 2010**

TECHNICAL REPORT SERIES

No. CS-TR-1186 January, 2010

Splitting Atoms with Rely/Guarantee Conditions Coupled with Data Reification

C.B. Jones, K.G. Pierce

Abstract

This paper presents a novel formal development of a non-trivial parallel program: Simpson's implementation of asynchronous communication mechanisms (ACMs). Although the correctness of the "4-slot algorithm" has been shown elsewhere, earlier developments are by no means intuitive. The aims of this paper include both the presentation of an understandable (yet formal) design history and the establishment of another way of "splitting (software) atoms". Using the "fiction of atomicity" as an aid to understanding the initial steps of development, the top-level specification is developed to code. The rely-guarantee approach is, here, combined with notions of read/write frames and "phased" specifications; the atomicity assumptions implied by rely/guarantee conditions are realised by clever choice of data representation. The development method herein is compared with other approaches --in a spirit of cooperation-- as the authors believe that constructive comparison elucidates many of the finer points in the "4-slot" specification/development and of parallel programs in general.

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

Bibliographical details

JONES, C.B., PIERCE, K.G.

Splitting Atoms with Rely/Guarantee Conditions Coupled with Data Reification
[By] C.B. Jones, K.G. Pierce

Newcastle upon Tyne: University of Newcastle upon Tyne: Computing Science, 2010.

(University of Newcastle upon Tyne, Computing Science, Technical Report Series, No. CS-TR-1186)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE
Computing Science. Technical Report Series. CS-TR-1186

Abstract

This paper presents a novel formal development of a non-trivial parallel program: Simpson's implementation of asynchronous communication mechanisms (ACMs). Although the correctness of the "4-slot algorithm" has been shown elsewhere, earlier developments are by no means intuitive. The aims of this paper include both the presentation of an understandable (yet formal) design history and the establishment of another way of "splitting (software) atoms". Using the "fiction of atomicity" as an aid to understanding the initial steps of development, the top-level specification is developed to code. The rely-guarantee approach is, here, combined with notions of read/write frames and "phased" specifications; the atomicity assumptions implied by rely/guarantee conditions are realised by clever choice of data representation. The development method herein is compared with other approaches --in a spirit of cooperation-- as the authors believe that constructive comparison elucidates many of the finer points in the "4-slot" specification/development and of parallel programs in general.

About the authors

Cliff Jones is a Professor of Computing Science at Newcastle University. He is now applying research on formal methods to wider issues of dependability. Until 2007 his major research involvement was the five university IRC on "Dependability of Computer-Based Systems" of which he was overall Project Director - he is now PI of the follow-on *Platform Grant* "Trustworthy Ambient Systems" (TrAmS) (also EPSRC). He is also PI on an EPSRC-funded project "Splitting (Software) Atoms Safely" and coordinates the "Methodology" strand of the EU-funded RODIN project. As well as his academic career, Cliff has spent over twenty years in industry. His fifteen years in IBM saw among other things the creation -with colleagues in Vienna- of VDM which is one of the better known "formal methods". Under Tony Hoare, Cliff wrote his doctoral thesis in two years (and enjoyed the family atmosphere of Wolfson College). From Oxford, he moved directly to a chair at Manchester University where he built a world-class Formal Methods group which -among other projects- was the academic lead in the largest Software Engineering project funded by the Alvey programme (IPSE 2.5 created the "mural"(Formal Method) Support Systems theorem proving assistant). Cliff is a *Fellow* of the Royal Academy of Engineering (FREng), ACM, BCS, and IET. He has been a member of IFIP Working Group 2.3 (Programming Methodology) since 1973 (and was Chair from 1987-96).

Ken received his BSc (Hons) in Computer Science (Software Engineering) from Newcastle University in 2005.
Ken studied for a PhD under the supervision of Prof. Cliff Jones as part of the EPSRC "Splitting (Software) Atoms Safely" project. His thesis, titled "Enhancing the Usability of Rely-Guarantee Conditions for Atomicity Refinement", was published in December 2009. Ken is currently working on the DESTECS project (destecs.org). The project is a consortium of research groups and companies working on the challenge of developing fault-tolerant embedded systems. Specifically, the aim is to explore collaborative modelling and simulation in the design of embedded systems. The Newcastle team is principally concerned with methodology and how fault-tolerance can be incorporated into these collaborative, multi-disciplinary models.

Suggested keywords

SIMPSONS FOUR SLOT ALGORITHM
VDM
DATA REIFICATION
RELY/GUARANTEE
FORMAL METHODS

Splitting Atoms with Rely/Guarantee Conditions Coupled with Data Reification*

Cliff B. Jones and Ken G. Pierce

School of Computing Science, Newcastle University, UK

Abstract. This paper presents a novel formal development of a non-trivial parallel program: Simpson's implementation of asynchronous communication mechanisms (ACMs). Although the correctness of the "4-slot algorithm" has been shown elsewhere, earlier developments are by no means intuitive. The aims of this paper include both the presentation of an understandable (yet formal) design history and the establishment of another way of "splitting (software) atoms". Using the "fiction of atomicity" as an aid to understanding the initial steps of development, the top-level specification is developed to code. The rely-guarantee approach is, here, combined with notions of read/write frames and "phased" specifications; the atomicity assumptions implied by rely/guarantee conditions are realised by clever choice of data representation. The development method herein is compared with other approaches –in a spirit of cooperation– as the authors believe that constructive comparison elucidates many of the finer points in the "4-slot" specification/development and of parallel programs in general.

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". To do this, it addresses an intricate parallel program to illustrate the novel aspects of an approach to the development of parallel programs.

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 detailed proof obligations to justify such steps. Even if –as here– what is being created 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 2.2 below) extends the formal tools to cover classes of 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.

* This paper appears in *ABZ '08: Proceedings of the 1st international conference on Abstract State Machines, B and Z*, pages 360–377, Berlin, Heidelberg, 2008. Springer-Verlag. Please cite the published version in preference to this one.

More recently, research has looked at using a "fiction of atomicity" as an additional abstraction [Jon03] in the specification of parallel programs; the corresponding development notion is sometimes referred to as "splitting (software) atoms safely"; one example of this approach is Jones' transformation rules for "pobl" as in [Jon96].

This paper uses rely and guarantee conditions in reasoning about "splitting atoms". In particular, the example illustrates the combination of rely/guarantee reasoning with data reification outlined in [Jon07].

Although this paper offers comparisons (see Section 6), it is quite specifically not competitive. In fact, the intention is to write a longer joint journal paper with Abrial and Cansell whose rather different approach [AC05] fits into the evolving research on "Event-B" which is being pursued in the EU "Deploy" project in which the first author is also a player. Moreover, 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.

The application chosen concerns "Asynchronous Communication Methods" — specifically, the four-slot implementation of ACMs devised by Hugo Simpson [Sim97] — see Section 3. The algorithm is ingenious and its correctness by no means obvious.

The real message of the current paper is however the (generic) approach outlined; a key test is whether the reader gains insight by reading the development below. In each major section, there is a sub-section that restates the methods used so that it is clear what the reader can take from the specific example to other specification and design challenges.

2 Background material

This section *briefly* sets out state-of-the-art methods; any reader who is unfamiliar with these areas should consult the cited publications.

2.1 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 complex way (possibly to achieve performance). Separation of these issues results in clearer documentation of design histories. The preferred development rule in VDM [Jon90] works where the chosen reification (representation of the abstraction) can be described 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 simple 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 retains information to express potential non-determinacy and this information is superfluous in a step of development where the non-determinacy is reduced. In a sense this is "intentional bias". 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].

2.2 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 the 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 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 references include [Jon81,Jon83a,Jon83b,Jon96] but a number of other theses extend the basic idea.[1] A notable extension to cover progress arguments is [Stø90]. 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 a specific set of rules. In fact, the general approach can also be applied to communication-based concurrency.

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 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 5 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.

[1] 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

2.3 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 (although Section 2.4 notes a common concern). 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 "refine 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]. Some further comments relating [AC08] to the material in this paper are made in Section 6.3.

2.4 Tracking execution with variables

There are two roles that variables can play that are close to "pseudo instruction counters". The shorthand term "phasing" is used in this paper to refer to either role.

In the (rely/guarantee) approach it is sometimes necessary to delineate different interference in different phases of a program. One way of handling this is by using pseudo-instruction counters and implications whose left-hand side makes appropriate case distinctions. One objective below is to show that using control constructs like "semicolon" provides another way of representing changing assumptions.

The other use of pseudo instruction counters is vividly illustrated in Abrial's event refinement approach. The order of execution of the events with true guards in a given set is non-deterministic. In situations where the correctness depends on a constrained order, pseudo instruction counters are tested in guards and set in the corresponding events.[2] The Abrial/Cansell approach is discussed in Section 6.3.

2.5 Status of the proofs

The authors have checked all of the proof obligations required in the development below. A technical report version of this paper will add appendices that make outline proofs available for scrutiny. The second author's thesis will present proofs at the level of formality used in [Jon90]. Plans to attempt machine checked proofs are currently being considered.

[2] This is reminiscent of the proof of the Boehm/Jacopini theorem that "goto" statements can be avoided.

3 ACMs and their specification

"Asynchronous Communication Methods" (ACMs) address an extremely interesting application scenario. First, imagine two process 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. So it is *not*, for example, possible to use a conventional shared variable –access to which is controlled by some device such as semaphores– since one process could be delayed waiting for a lock to be released. To sharpen the issues, it might be useful to think of *Value* below as being large — something that certainly can't be changed in one machine cycle ("atomically"). ACMs are used in important high speed communication situations such as passing values from sensors to flight control software.

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. The more complicated consequences are shown once a formal specification has been given.

3.1 A specification

The issue of a formal statement of what behaviour a valid ACM is allowed to exhibit is itself non-trivial and different approaches are already distinguished at this starting point. (Alternatives are discussed in Section 3.3.) Sections 4 and 5 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 of a simple (atomic) variable. Unfortunately, this is not an appropriate abstraction because it does not show the full potential behaviour of an ACM. In particular, a read operation could start and –before it delivers a value– several write operations could start and complete. Alternatively, a write operation could start and –before it completes– several different read operations could start and complete. It is necessary to show which values can be delivered to the receiving process. A straightforward way to do this is to distinguish between *start-Read*/*end-Read* and *start-Write*/*end-Write*.[3] An underlying state to characterise these operations (in VDM notation[4]) could be:

[3] For those who feel queasy about this in a specification, Section 3.3 discusses alternatives. Furthermore, the approach of the current section can be proved to fit with such specifications.

[4] Remember that types in VDM are restricted by invariants; so, for example, quantifying over Σ^a only includes records that satisfy its invariant.

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

inv $(mk\text{-}\Sigma^a(\mathit{data\text{-}w}, \mathit{fresh\text{-}w}, \mathit{hold\text{-}r})) \triangleq \mathit{hold\text{-}r} \leq \mathit{fresh\text{-}w}$

The idea here is that *data-w* retains all values written; *start-Write* first stores a new value but only *end-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 indicate whether the reader or writer can change their values; this shows straight away that there are no variables written to by both "sides".)

It is obviously necessary to initialise the state. Most authors who give formal presentations do this by assuming that a value `x` has been written *then* read once. This can be shown as:

$$\sigma_0^a = mk\text{-}\Sigma^a([\mathtt{x}], 1, 1)$$

with the following pseudo-code:

while true do
  start-Write(*v*: *Value*): *data-w* $\leftarrow$ *data-w* $\curvearrowright$ [*v*];
  end-Write(): *fresh-w* $\leftarrow$ **len** *data-w*
od
while true do
  start-Read(): *hold-r* $\leftarrow$ *fresh-w*;
  end-Read()*r*: *Value*: *r* $\leftarrow$ *data-w*(*i*) **for some** $i \in \{\mathit{hold\text{-}r}..\mathit{fresh\text{-}w}\}$
od

The code ensures that old values cannot be read. 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-r* before the choice is made and *hold-r* is never greater than *fresh-r*, the read process cannot return an old value (though the same value may be returned more than once).

Figures 1, 2 and 3 give possible executions of the code (giving the operation name and corresponding final sate). Figure 1 is a simple sequential write and read: `y` is added to *data-w*, marked as fresh and subsequently read. In Figure 2, the read begins before the write ends and the read yields `x`.

The more complex case in Figure 3 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.

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

Fig. 1. Sequential case

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

Fig. 2. Interleaved case

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

Fig. 3. Non-deterministic case

The pseudo-code above is brief and offers the intuition of what can happen but for the development that follows, this needs to be presented as formal (VDM) specifications of the four operations. These are straightforward (see Figure 4)[5].

Write(v: *Value*)
  start-Write(v: *Value*)
    wr *data-w*
    post $\textit{data-w} = \overleftarrow{\textit{data-w}} \curvearrowright [v]$
  end-Write(v: *Value*)
    rd *data-w*
    wr *fresh-w*
    pre $\textit{data-w}(\textbf{len}\ \textit{data-w}) = v$
    post $\textit{fresh-w} = \textbf{len}\ \textit{data-w}$

Read()r: *Value*
  local $\textit{hold-r}: \mathbb{N}$
  start-Read()
    wr *hold-r*
    rd *fresh-w*
    post $\textit{hold-r} = \textit{fresh-w}$
  end-Read()r: *Value*
    rd *data-w*, *fresh-w*
    post $\exists i \in \{\textit{hold-r}..\textit{fresh-w}\} \cdot r = \textit{data-w}(i)$

Fig. 4. Specification in terms of four sub-operations

[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).

In these operation specifications, the standard VDM style of marking the read/write access is used. This proves particularly valuable below when interference is considered. One non-standard extension is also used and that is the declaring *hold-r* as local to the two *Read* operations. This is essentially marking it as invisible to the two *Write* operations. (The efficacy of these markings is addressed in the (draft) thesis of the second author.)

Notice that rely/guarantee conditions are not *yet* necessary because the four operations are assumed to be atomic. The fiction of atomicity is used to achieve a simple specification. At this point, the specifications imply big assumptions about atomic update of *data-w*; this is addressed in the following sections. Perhaps of more interest is the decision to use semicolon as a tool in specifications — again, this is addressed below.

Note that *pre-end-Write* is required to pass information between the two write processes. The proof showing that this is implied by *post-start-Write* is immediate.

3.2 Splitting atoms in Σ^a

As observed, the operations in the preceding section are assumed to execute atomically. The process of "splitting atoms" can begin by considering the overlap of *Read* and *Write* sub-operations. This could be very difficult to describe. Indeed, the cleverness of the final code is all about finding a way to do this safely whilst achieving "asynchroneity". Here, we postpone the key property that there is no delay to either process since it can be achieved by further splitting of atoms.

For now, Figure 5 contains exactly the post conditions of the preceding section and adds rely and guarantee conditions that represent the possible interference. Notice first that the state Σ^a is unchanged. Rely/guarantee assertions are easy to add because all that is necessary is to make sure that results required in the post conditions cannot be subverted by interference.

It is a simple task to check that the rely and guarantee conditions in the two threads are consistent. The work involved is almost syntactic because of the limitations on read and write access marked in the VDM operation specifications. For example, both *rely-start-Write* and *rely-end-Write* follow immediately from the fact that neither *Read* component has write access to the relevant variables.

An astute reader might be very 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 here is to show that the splitting atoms development 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 a specification of an 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.

$Write(v\colon Value)$
  $start\text{-}Write(v\colon Value)$
    rd $fresh\text{-}w$
    wr $data\text{-}w$
    rely $fresh\text{-}w = \overleftarrow{fresh\text{-}w} \wedge data\text{-}w = \overleftarrow{data\text{-}w}$
    guar $\{1..fresh\text{-}w\} \lhd data\text{-}w = \{1..fresh\text{-}w\} \lhd \overleftarrow{data\text{-}w}$
    post $data\text{-}w = \overleftarrow{data\text{-}w} \curvearrowright [v]$
  $end\text{-}Write(v\colon Value)$
    rd $data\text{-}w$
    wr $fresh\text{-}w$
    pre $data\text{-}w(\mathbf{len}\, data\text{-}w) = v$
    rely $fresh\text{-}w = \overleftarrow{fresh\text{-}w} \wedge data\text{-}w = \overleftarrow{data\text{-}w}$
    post $fresh\text{-}w = \mathbf{len}\, data\text{-}w$

$Read()r\colon Value$
  $start\text{-}Read()$
    rd $fresh\text{-}w$
    wr $hold\text{-}r$
    rely $hold\text{-}r = \overleftarrow{hold\text{-}r}$
    post $hold\text{-}r \in \{\overleftarrow{fresh\text{-}w}, fresh\text{-}w\}$
  $end\text{-}Read()r\colon Value$
    rd $data\text{-}w, fresh\text{-}w, hold\text{-}r$
    rely $hold\text{-}r = \overleftarrow{hold\text{-}r} \wedge \forall i \in \{hold\text{-}r..\overleftarrow{fresh\text{-}w}\} \cdot data\text{-}w(i) = \overleftarrow{data\text{-}w}(i)$
    post $\exists i \in \{hold\text{-}r..\overleftarrow{fresh\text{-}w}\} \cdot r = \overleftarrow{data\text{-}w}(i)$

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

3.3 Alternative specifications

The most surprising decision in the specification used here is the retention of values (in *data-w* of σ^a) that can no longer be accessed. Henderson goes to pains to delete "old" values after they have been overtaken by subsequent reads. The cost in [Hen04] is that both *Read* and *Write* need to have a record of where the other process is in its execution. True, this record keeping is eliminated in the subsequent development; but so are our superfluous values. In both cases, the same technical rule comes to the rescue. Our current view is that leaving the extra values results in a clearer specification.

Abrial and Cansell [AC08] 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. In the journal version of this paper a proof will be added that the initial "specification" in Section 3.1 satisfies their specification. This then leaves the user to decide which is the most intuitive way of understanding ACM behaviour.

3.4 Summary of specification methods used

The ideal of the "fiction of atomicity" would be to abstract from all of the details of ACMs by using a single atomically accessed variable as an abstraction. Since this does not describe all of the possible behaviours, 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 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*, 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 recognise the arguments for a specification in terms of traces but believe phasing is sometimes easier to understand.

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.

4 Retaining less history

The first real reification is to an intermediate representation in which it is possible to retain fewer *Value*s than in Σ^a. Not only can Σ^i get away with fewer values, it is also clear that it might be possible to lock only parts of its *data-w* component and thus increase concurrency. Thus this step moves towards the idea of multiple slots without being specific as to how many there must be to make the algorithm work. 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.

4.1 The data representation

The state representation for this reification is:

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

$$\textbf{inv } (\mathit{mk}\text{-}\Sigma^i(\mathit{data}, \mathit{fresh}, \mathit{hold\text{-}r}, \mathit{hold\text{-}w})) \triangleq \{\mathit{fresh}, \mathit{hold\text{-}r}, \mathit{hold\text{-}w}\} \subseteq \textbf{dom } \mathit{data}$$

At this step of development, the (indexing) set X is arbitrary. In order to show the initial state assume that $X \in \{\alpha, \beta, \ldots\}$. Then:

$$\sigma_0^i = \mathit{mk}\text{-}\Sigma^i(\{\alpha \mapsto \mathtt{x}\}, \alpha, \alpha, \alpha)$$

4.2 Relating Σ^i to Σ^a

The fact that the chosen state in the specification (Σ^a) cannot be "retrieved" from the representation as in the simple VDM reification rule means that the connection between elements of Σ^a/Σ^i has to be given by a relation:

$$
\begin{array}{l}
r : \Sigma^a \times \Sigma^i \to \mathbb{B} \\
r(mk\text{-}\Sigma^a(data\text{-}w^a, fresh\text{-}w^a, hold\text{-}r^a), \\
\qquad\qquad mk\text{-}\Sigma^i(data\text{-}w^i, fresh\text{-}w^i, hold\text{-}r^i, hold\text{-}w^i)) \quad \triangleq \\
\quad \textbf{rng}\ data\text{-}w^i \subseteq \textbf{elems}\ data\text{-}w^a \wedge \\
\quad data\text{-}w^a(fresh\text{-}w^a) = data\text{-}w^i(fresh\text{-}w^i) \wedge \\
\quad data\text{-}w^a(hold\text{-}r^a) = data\text{-}w^i(hold\text{-}r^i)
\end{array}
$$

Because the representation here has (potentially) less information than the abstraction, it is necessary to use the refinement rule given in [Nip86,Nip87]. For the current case it is necessary to show for each operation that:

$$r(\sigma_1^a, \sigma_1^i) \wedge post^i(\sigma_1^i, \sigma_2^i) \ \Rightarrow\ \exists \sigma_2^a \in \Sigma^a \cdot post^a(\sigma_1^a, \sigma_2^a) \wedge r(\sigma_2^a, \sigma_2^i)$$

Generalisations of this rule to add inputs or outputs are obvious.

4.3 Specifications of the sub-operations

The specifications of the four sub-operations over the Σ^i states are shown in Figure 6. There is masses of non-determinism here — in fact, one valid implementation is to have $X = \mathbb{N}$ and retain the whole sequence as in Section 3.

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.[6]

4.4 Justifying this step

The technical report version of this paper contains proofs of the (initialisation, and) four sub-operations in an appendix. These proofs will be presented formally in the second author's forthcoming PhD thesis.

Checking the coherence of the rely and guarantee conditions between the two sub-operations of *Read* and of *Write* is somewhat more work than in Section 3 but the effort required is still drastically reduced by the read and write frames (coupled with the **local** in *Write*).

4.5 Summary of development methods used in this stage

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 has been included in VDM since [Jon90, §9.3]. The use here is technically interesting;

[6] The argument why that is not enough is set out in [Hen04] and is not repeated here.

$Write(v\colon Value)$
  local $hold\text{-}w\colon X$
  $start\text{-}Write(v\colon Value)$
    rd $hold\text{-}r, fresh\text{-}w$
    wr $data\text{-}w$
    rely $fresh\text{-}w = \overleftarrow{fresh\text{-}w} \wedge data\text{-}w = \overleftarrow{data\text{-}w}$
    guar $\{\overleftarrow{hold\text{-}r}, hold\text{-}r\} \lhd data\text{-}w = \{\overleftarrow{hold\text{-}r}, hold\text{-}r\} \lhd \overleftarrow{data\text{-}w}$
    post $hold\text{-}w \in (X - \{fresh\text{-}w, \overleftarrow{hold\text{-}r}, hold\text{-}r\}) \wedge$
      $data\text{-}w = \overleftarrow{data\text{-}w} \dagger \{hold\text{-}w \mapsto v\}$
  $end\text{-}Write(v\colon Value)$
    rd $data\text{-}w$
    wr $fresh\text{-}w$
    pre $data\text{-}w(hold\text{-}w) = v$
    rely $fresh\text{-}w = \overleftarrow{fresh\text{-}w} \wedge data\text{-}w = \overleftarrow{data\text{-}w}$
    post $fresh\text{-}w = hold\text{-}w$

$Read()r\colon Value$
  $start\text{-}Read()$
    rd $fresh\text{-}w$
    wr $hold\text{-}r$
    rely $hold\text{-}r = \overleftarrow{hold\text{-}r}$
    post $hold\text{-}r \in \{\overleftarrow{fresh\text{-}w}, fresh\text{-}w\}$
  $end\text{-}Read()r\colon Value$
    rd $hold\text{-}r, data\text{-}w$
    rely $hold\text{-}r = \overleftarrow{hold\text{-}r} \wedge data\text{-}w(hold\text{-}r) = \overleftarrow{data\text{-}w}(hold\text{-}r)$
    post $r = data\text{-}w(hold\text{-}r)$

Fig. 6. Rely/guarantee specifications on Σ^i

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 part of the method to reduce to general rules.

Another key point only sees its completion in Section 5 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 3 that this is not allowed in ACMs. Such roadblocks (leading to backtracking) cannot be ruled out by any method whether formal or informal.

There are key links from this section to the second author's upcoming PhD thesis. In particular, one sees even more clearly in this section than the last how rely and guarantee conditions are simpler to express because of the read and write frames. Furthermore, without "phasing", there would be much more to write with implications all over the place.

5 The four-slot representation

In purely formal terms, the task remaining after Figure 6 (which uses Σ^i) is to find a representation that admits atomic changes in a sensible machine. The crucial contribution of Simpson's "4-slot" algorithm is to achieve control over where the reader and writer find or change values with only two single bit control variables. This is where the link between "splitting atoms" and reification (cf. [Jon07, §3]) comes into play. These control variables keep the reader and writer from "colliding" while never delaying each other. This is the ingenuity in Hugo Simpson's contribution and there is absolutely no claim here that the formalism is a substitute for such design inspiration. Used by a designer (which it wasn't), formalism can establish that proceeding to the next design step leaves no hostage to fortune on correctness; used as here, the formalism can provide a clear understanding, documentation (and appreciation) of an intricate piece of code.

5.1 The data representation

The size of the domain of the *data-w* field of Σ^i is not constrained. Simpson's "4-slot" approach shows that the domain need only have cardinality four. Furthermore, he shows that treating the *data* map as two pairs (P below) of two slots (indexed by S below) makes their bookkeeping atomic.

So the essential difference between Σ^i of Section 4 and Σ^r here is that the general index set X of the former is represented here as a pair (P, S). The other changes are to control variables whose role becomes clear in Section 5.2.

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

where:

$$P, S = \mathit{token}\text{-}\mathbf{set}$$

These two sets can be identical and each has two elements: $P = S, \mathbf{card}\, P = 2$, with an inverter function, ρ (for "reverse")[7], such that $\rho(i) \neq i$.

5.2 Justifying the step from Σ^r to Σ^i

Figure 7 reflects the differences between the two state spaces. The justification of this step of development requires showing that the combination of index values in Σ^r can be used to justify the properties of X etc. in Σ^i. (This is the process

[7] Many authors use $\mathbb{B}$ for P and then employ negation — to us, this is a coding trick!

Write(v: *Value*)
local *wp-w*: P
local *ws-w*: S
start-Write(v: *Value*)
rd *pair-r*, *slot-w*
wr *data-w*
rely $slot\text{-}w = \overleftarrow{slot\text{-}w} \wedge data\text{-}w = \overleftarrow{data\text{-}w}$
guar $\{(\overleftarrow{pair\text{-}r}, slot\text{-}w(\overleftarrow{pair\text{-}r}), (pair\text{-}r, slot\text{-}w(pair\text{-}r)\} \lhd data\text{-}w =$
$\{(\overleftarrow{pair\text{-}r}, slot\text{-}w(\overleftarrow{pair\text{-}r}), (pair\text{-}r, slot\text{-}w(pair\text{-}r)\} \lhd \overleftarrow{data\text{-}w}$
post $wp\text{-}w = \rho(\overleftarrow{pair\text{-}r}) \wedge ws\text{-}w = \rho(slot\text{-}w(wp\text{-}w)) \wedge$
$data\text{-}w(wp\text{-}w, ws\text{-}w) = v$
end-Write()
wr *pair-w*, *slot-w*
rely $pair\text{-}w = \overleftarrow{pair\text{-}w} \wedge slot\text{-}w = \overleftarrow{slot\text{-}w}$
guar $slot\text{-}w(pair\text{-}r) = \overleftarrow{slot\text{-}w}(pair\text{-}r)$
post $slot\text{-}w(wp\text{-}w) = ws\text{-}w \wedge pair\text{-}w = wp\text{-}w$

Read()r: *Value*
local *rs-r*: S
start-Read()
rd *pair-w*, *slot-w*
wr *pair-r*
rely $slot\text{-}w(pair\text{-}r) = \overleftarrow{slot\text{-}w}(pair\text{-}r) \wedge pair\text{-}r = \overleftarrow{pair\text{-}r}$
post $pair\text{-}r = \overleftarrow{pair\text{-}w} \wedge rs\text{-}r = \overleftarrow{slot\text{-}w}(pair\text{-}r)$
end-Read()r: *Value*
rd *pair-r*, *data-w*
rely $pair\text{-}r = \overleftarrow{pair\text{-}r} \wedge data\text{-}w(pair\text{-}r, rs\text{-}r) = \overleftarrow{data\text{-}w}(pair\text{-}r, rs\text{-}r)$
post $r = data\text{-}w(pair\text{-}r, rs\text{-}r)$

Fig. 7. Final (Σ^r) rely/guarantee specification of code

described in [Jon07, §3].) Thus one shows that each of the conditions of Figure 7 corresponds to those of Figure 6. This follows from:

Σ^i	represented in Σ^r by
$data\text{-}w^i$	$data\text{-}w^r$
$fresh^i$	$(pair\text{-}w^r, slot\text{-}w^r(pair^r(pair\text{-}w^r)))$
$hold\text{-}r^i$	$(pair\text{-}r^r, slot\text{-}w^r(pair^r(pair\text{-}r^r)))$
$hold\text{-}w^i$	$(wp\text{-}w^r, wp\text{-}s^r)$

The proofs will be given in an appendix to the technical report version of this paper.

5.3 The code

It is straightforward to show that the code in Figure 8 satisfies the specifications of the sub-operations in Section 5.2.

```
Write(v: Value)
  local wp-w: P
  local ws-w: S
    wp-w ← ρ(pair-r);
    ws-w ← ρ(slot-w(wp-w));
    data-w(wp-w, ws-w) ← v;
    slot-w(wp-w) ← ws-w;
    pair-w ← wp-w

Read()r: Value
  local rs-r: S
    pair-r ← pair-w;
    rs-r ← slot-w(pair-r);
    r ← data-w(pair-r, rs-r)
```

Fig. 8. Code for Simpson's algorithm

5.4 Summary of development methods used in this stage

Finally, the usefulness of the intermediate data abstraction becomes clear in this step: it is relatively easy to see the pair/slot mapping as a way of simplifying a mapping from the arbitrary set X. Moreover, the whole thrust of "splitting atoms safely" is clear in this step.

6 Conclusions

This section both summarises the general methodological messages of the paper and offers brief descriptions of some other recent justifications of Simpson's algorithm. In making such comparisons, the authors are not trying to be competitive but to use this intricate algorithm to indicate what insight can be given by various approaches.

6.1 Summary

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

However, the intention was not to add yet another correctness argument of one specific algorithm but instead to use this development 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 Sections 3.4, 4.5 and 5.4 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.
- While rely/guarantee conditions allow us to reason about the interference, a clever data reification is required (which Simpson gives us).
- Rely/guarantee reasoning is greatly simplified by the use of frames and phasing arguments.

6.2 Brief comments on Henderson's development

Henderson's research (in particular, his thesis [Hen04]) 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*. As mentioned in Section 3.3, 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 5 of the current paper, the preceding interference specifications are achieved by capitalising on Simpson's four-slot representation.

The reader is also referred to [HP02] and [PHA04]; the second of these addresses the delicate issue of "meta-stability" of the control bits.

6.3 Comparison with event decomposition

The "event decomposition" method described in, for example, [AC05] is extremely interesting because it is general. Attention has already been drawn above to its use of a "pseudo instruction counter" which is related to the "phasing" idea used here. They avoid any 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 current authors do 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).

As indicated, it is the hope of the current authors that a comparison paper might be written together with Jean-Raymond Abrial and Dominique Cansell.

6.4 Comparison with "Separation Logic"

Another exciting development in research on concurrent code has been the recent developments around "concurrent separation logic". 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 current work on Simpson's algorithm. The title of [BA08] alone should indicate why this is exciting. Again, the availability of this in published form will admit proper unbiased comparison.

Acknowledgments

The authors are grateful to Jean-Raymond Abrial and Dominique Cansell for sharing ongoing work in this area. Similarly, the preview of the paper by Richard Bornat and Hasan Amjad is gratefully acknowledged even though time has not yet permitted a full comparison. Thanks also go to Peter O'Hearn, Hongseok Yang, Viktor Vafeiadis and Matt Parkinson for general and ongoing discussions on development methods for concurrency.

Of course, the original inspiration of the specific algorithm comes from Hugo Simpson's contribution. 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 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 comcurrent 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.

[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.

[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.

[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.

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

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

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

[Stø90] K. Stølen. *Development of Parallel Programs on Shared Data-Structures*. PhD thesis, Manchester University, 1990. Available as UMCS-91-1-1.

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