COMPUTING SCIENCE

The role of auxiliary variables in the formal development of concurrent programs

Cliff B. Jones

TECHNICAL REPORT SERIES

No. CS-TR-1179 November 2009

TECHNICAL REPORT SERIES

No. CS-TR-1179 November, 2009

The role of auxiliary variables in the formal development of concurrent programs

C.B. Jones

Abstract

So called “auxiliary variables”' are often used in reasoning about concurrent programs. They can be useful - but they can also be undesirable in that they can undermine the hard won property of “compositionality”. This paper explores the issue of auxiliary variables and tries to set concerns about overuse in a wider context; it concludes with an attempt to recommend constraints on their use.

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.

The role of auxiliary variables in the formal development of concurrent programs
[By] C.B. Jones

Newcastle upon Tyne: University of Newcastle upon Tyne: Computing Science, 2009.

(University of Newcastle upon Tyne, Computing Science, Technical Report Series, No. CS-TR-1179)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE
Computing Science. Technical Report Series. CS-TR-1179

Abstract

So called "auxiliary variables" are often used in reasoning about concurrent programs.
They can be useful --- but they can also be undesirable in that they can undermine the hard won property of “compositionality”. This paper explores the issue of auxiliary variables and tries to set concerns about overuse in a wider context; it concludes with an attempt to recommend constraints on their use.

About the author

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). During his time at Manchester, Cliff had a 5-year *Senior Fellowship* from the research council and later spent a sabbatical at Cambridge for the whole of the Newton Institute event on "Semantics" (and appreciated the hospitality of a Visiting Fellowship at Gonville & Caius College. Much of his research at this time focused on formal (compositional) development methods for concurrent systems. In 1996 he moved to Harlequin, directing some fifty developers on Information Management projects and finally became overall Technical Director before leaving to re-join academia in 1999. 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).

Suggested keywords

CONCURRENCY, RELY/GUARANTEE, FORMAL METHODS

The role of auxiliary variables in the formal development of concurrent programs

Cliff B. Jones
School of Computing Science, Newcastle University, UK
cliff.jones@ncl.ac.uk

November 22, 2009

Abstract

So called "auxiliary variables" are often used in reasoning about concurrent programs. They can be useful — but they can also be undesirable in that they can undermine the hard won property of "compositionality". This paper explores the issue of auxiliary variables and tries to set concerns about overuse in a wider context; it concludes with an attempt to recommend constraints on their use.

This is the Technical Report version of a paper which will be printed in a *Festschrift* for Tony Hoare.

1 Introduction

There have been a number of "X considered harmful" papers, the most famous being [Dij68]. The position taken here is that the use –or rather overuse– of "auxiliary variables" (sometimes referred to as "ghost variables") can be harmful in the development of concurrent programs.

The reason that concurrent programs are difficult to think about is the interference that comes from their environment. Interference is the reason that it is difficult to find compositional ways of formally developing concurrent programs; rely/guarantee ideas offer a way to achieve a notion of "compositionality" that enables separate development of programs that run in parallel. In most cases, rely conditions express assumptions about how variables written in another process (or "thread") change. Auxiliary variables are often used in reasoning about concurrent programs; typically each such variable is changed in exactly one process and only used in assertions of other processes. The alternative phrase "ghost variables" emphasises the fact that they can subsequently be erased without affecting the behaviour of a program.

There is, however, a danger inherent in the use of ghost or auxiliary variables in reasoning about concurrency. In the extreme, they can be used to record the entire history of execution of a process; if the rely conditions of other processes use this history, there is no abstraction of the interference. One could not, for example, reuse the proof with a slightly different split. More importantly, there is no sense in which a design decision to split a system into two or more parallel threads would facilitate separate development.

The next two sections include a review of known material. At first sight, this might appear to be a digression but, on the one hand, it builds up to some key issues about concurrency and, on the other hand, identifies via a rather different route a basis for believing that abstraction is best served by minimising auxiliary variables. The discussion of rely/guarantee thinking in Section 2 also serves to make the paper relatively self-contained. Section 4 discusses the search for a general approach to "atomicity refinement" and, finally, Section 5.2 sets out my current views on using auxiliary variables.

2 Rely/guarantee "thinking"

2.1 For comparison: the sequential case

Today, it is second nature to talk about specifications in the form of pre and post conditions but this was not always the case. Tony Hoare's "axiomatic basis" paper led to what might reasonably be classed a "paradigm shift" in the way computer scientists think about programs.[1] The move from the flowcharts of Floyd [Flo67] or King [Kin69] to thinking about programs in non-operational terms is clear in [Hoa69] and crucial for the intellectual shift that has followed.

In fact, the most important point about pre/post conditions was not really explicitly recognised in Tony's papers until he presented a development of *FIND* in [Hoa71] in which he shows how the axiomatic approach offers a useful notion of separating development choices. Items that are specified (and are to be implemented by a program) are referred to as "operations" as in VDM (the B method uses the same term, Event-B uses "events"). Assuming that one has some specified operation S and makes a design decision to split it into a sequential composition of operations $S1$ and $S2$ (thus $S = S1; S2$) — of course specifying $S1$ and $S2$ with pre and post conditions, the key property is that the development of $S1$ can be independent of S and $S2$. Once the proof rule for "semicolon" is discharged, that step of the argument does not need to be revisited (unless there is some broader change to be made). This property of a development method is often referred to as "compositionality" and this term is used below. To find compositional development rules for sequential programs is reasonably straightforward. A top-down documentation of design can introduce design decisions in many layers but the proofs at each layer are independent of each other and the development of sibling operations.

Before the discussion moves to concurrent programming, there are several points to be made about the above approach to sequential programs — the message is that, although facing concurrency magnifies some problems, their seeds are present even with sequential reasoning.

First, there is the issue of whether one should strive for one, definitive, set of rules. Even within Hoare's framework, there are choices about how to present the rules for programming constructs. For example, one version might include a specific rule to weaken pre and post conditions — alternatively, such weakening can be built into the rules for each construct by adding implications.

Another issue is that misnamed "partial correctness" (vs. "total correctness"): termination was not handled in [Hoa69]. Ignoring termination was lampooned by McCarthy's "millionaire's algorithm" (to become a millionaire, walk along the street – pick up every piece of paper on the sidewalk – if it's a check –made out to you– for a million dollars then cash it; otherwise, discard the piece of paper and continue).

[1] My view of the importance of Hoare's paper led me to take [Hoa69] as the "fulcrum" for [Jon03]; that discussion links the prior work of Floyd, Naur and van Wijngaarden (and remarks on the lack of what could have been an interesting link back to Turing's work).

VDM [Jon90] rules are about "total correctness" — that is, they require termination. They also differ from some other approaches such as "weakest pre conditions" [Dij76, DS90] in that VDM's post conditions are relational (they are predicates of two states: initial and final).[2] Relevant to the issue of auxiliary variables is that relational post conditions have the advantage that they obviate the need to use free (logical) variables in weakest pre conditions approaches to define constraints on the final state that are relative to values in the initial state.

The proof rules in [Jon80] were –in Peter Aczel's polite phrase– "unmemorable"; his unpublished note [Acz82] gave a presentation of the VDM rules that is close to Hoare's original rules but deals with relational post-conditions and termination (these rules are used in [Jon90]):[3]

$$\boxed{\textit{while-I}}\ \frac{\{P \wedge b\}\ S\ \{P \wedge W\}}{\{P\}\ \textbf{while}\ b\ \textbf{do}\ S\ \textbf{od}\ \{P \wedge \neg\, b \wedge W^*\}}$$

The use of post-conditions (especially when presented as relations) yields a natural way of writing specifications that do not determine a unique outcome, somewhat loosely, these are often referred to as "non-deterministic specifications". It has become clear with usage that such specifications are a very good way of structuring the introduction of decisions during the design process. For example, the properties of a free storage manager are easily documented before a specific algorithm is designed.

One last, but important, point (that is magnified considerably by concurrency) is "expressive weakness". In common with some other approaches, it is a requirement in VDM that the set of states defined by the pre condition of any operation should be a subset of the domain of the relation characterised by the post condition of the operation. One might say that if an operation is required to terminate on some state, the post-condition should constrain the result state. This prompts a satisfaction relation that a valid step of development can widen the pre condition or restrict the non-determinacy in the post condition (subject to the aforementioned "satisfiability" condition). In most situations, these guidelines fit and are not even noticed but there are applications like security where non-determinacy has to play a different role and in "action systems" semantics are not preserved by widening "guards" — some of the alternatives are explored in [Bic95].

2.2 Onwards to the concurrent case

It is worth taking a careful look at why it is much harder to achieve compositionality for concurrent, than sequential, program development. Post conditions are enough to characterise sequential operations because the latter can be considered to execute atomically. In contrast, if two parallel processes share variables, each process can have an effect on the other. Such effects (viewed from the recipient) are "interference". Once this point is recognised, it becomes obvious that a development method for concurrent programs must support documentation of –and reasoning about– interference. As with other formalisations of development, the quest is then for tractability: we know that we need to record more than the input/output relation for an operation but recording the full history of execution is clearly not going to yield a compositional development method.

Rely/guarantee "thinking" is about finding this sweet point. A possible way to record a specification of a shared variable program is to add to:

$\textit{pre-}OP_i\colon \Sigma \to \mathbb{B}$
$\textit{post-}OP_i\colon \Sigma \times \Sigma \to \mathbb{B}$

a predicate that records what can happen to the shared state when the environment interferes:

$\textit{rely-}OP_i\colon \Sigma \times \Sigma \to \mathbb{B}$

and one that records the interference that OP_i will inflict on the environment:

$\textit{guar-}OP_i\colon \Sigma \times \Sigma \to \mathbb{B}$

An execution, in which the environment makes the state transition from σ_i to σ_{i+1} and the component makes the transition from σ_j to σ_{j+1}, is pictured in Figure 1.

It was perhaps not fully appreciated at the time of [Hoa69] that the roles of pre and post conditions differ in that a pre condition gives *permission* to a developer to ignore certain possibilities; the onus is on a user to prove that a component will not be initiated in a state that does not satisfy its pre condition. In contrast a post condition is an *obligation* on the code that is created according to the specification. This

[2] Both points were true not only in the early book on program development in VDM [Jon80] but also the earlier IBM reports [Jon72b, Jon72a].

[3] In the rule, P is a predicate of one state; W a predicate of two that is well founded (thus establishing termination without the need for a "variant function"); W^* is the reflexive and transitive closure of W. See [Jon90] for the honest form of this rule which has an additional hypothesis on definedness — but this paper is not about partial functions.

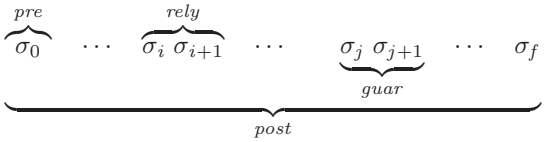

Figure 1: Illustrative execution under interference

Deontic view carries over: just as pre conditions should be viewed as assumptions that the developer can make about where the finished code will be deployed, rely conditions are assumptions that the developer can make on the limit of interference that the code will have to endure (i.e. permission to ignore the possibility of arbitrary interference). Similarly, a guarantee condition is like a post condition in that it is a commitment on the code finally created from the development process; in the case of a guarantee condition, the finished code must not generate interference that does not satisfy the specified relation.

Typical clauses that occur in rely and guarantee conditions are:

- some variable x is unchanged[4]
- a variable changes in some monotonic way — notice that the ordering need not be over numbers, the example in Section 3.3 uses $s \subseteq s'$
- the truth/falsity of some flag implies some condition similar to one of the above
- and, of course, ensuring that such a flag behaves as expected is an example of monotonic change

It is interesting that, even in fairly complicated concurrent programs, most variables are changed in only one thread even though many processes might access their values. The most common exception to this observation is in fact flag-like variables.

In [Jon81] and several subsequent papers, rely and guarantee conditions are constrained to be both transitive and reflexive: this corresponds to the observation that there can be zero or multiple steps of interference. This is only one of the points on which there is flexibility in choosing specific rules for reasoning about interference. This flexibility prompts the use of the term "rely/guarantee thinking" to make clear that we are not limiting the discussion to one specific set of rules.

It is worth looking at a representative rule. If one wishes to decompose the operation S into the parallel composition of S_l and S_r, it is clear that the interference generated by S_l can affect the outcome of S_r. In the spirit of presenting Hoare-like rules as $\{P\}\ S\ \{Q\}$ one can write: $\{P, R\}\ S\ \{G, Q\}$; a sound rule is:

$$\boxed{\textit{Par-I}}\ \frac{\begin{array}{l}\{P, R \vee G_r\}\ S_l\ \{G_l, Q_l\} \\ \{P, R \vee G_l\}\ S_r\ \{G_r, Q_r\} \\ G_l \vee G_r\ \Rightarrow\ G \\ \overleftarrow{P} \wedge Q_l \wedge Q_r \wedge (R \vee G_l \vee G_r)^*\ \Rightarrow\ Q\end{array}}{\{P, R\}\ S_l \parallel S_r\ \{G, Q\}}$$

There are a number of issues that could be addressed at this point but, for the immediate purpose of cuing discussion of auxiliary variables, the most pressing of these issues is the "expressive weakness" of rely/guarantee conditions. Basically, the decision to record potential interference in a single relation makes it difficult or impossible to state certain behaviours. For example, consider a sequence of instructions that can be viewed as progressing through two phases: in the first phase, some variable x is monotonically increased; whereas in the second phase, x is monotonically decreased. The union of the two behaviours $\overleftarrow{x} \leq x$ and $\overleftarrow{x} \geq x$ tells us nothing other than that x might change! Unfortunately, there are contexts that require something more useful than this nugatory information. Section 5.2 indicates that this specific example can often be finessed. Be that as it may, expressive weakness is one of the issues that the rely/guarantee rules for concurrency magnify (over the sequential case) and the question must be faced as to whether this forces the use of auxiliary variables. The answer is, however, deferred to Section 5.2 pending explanation of the distaste of such variables.

2.3 Conclusions so far

It is, perhaps, worth first repeating the point that the attempt here is to learn from "rely/guarantee thinking": the point has been made that there is considerable freedom in the presentation of such rules — far more than there is with Hoare logic of sequential programs (but, freedom –Section 2.2 points out– is already there). There is a significant literature on extensions and variants of rely/guarantee rules (see [Jon09]).

[4]Rely/guarantee conditions are quite capable of recording "no change" but Section 2.3 discusses how the read/write frames of VDM simplify such descriptions.

One interesting extension is the use of "dynamic invariants" in [CJ00]. There are also some odd variants including those that try to get by with predicates of single states for rely and guarantee conditions; [Sti86][5] even restricts post-conditions to being predicates of a single state. In each case, for practical application, the move away from relations creates the need for extra auxiliary or logical variables.

Turning now to the lessons themselves, they fall under the headings of abstraction, compositionality and granularity. The abstraction with *pre*/*post* might be described as "what – not how" (e.g. it is not only easier to write and/or read a specification of *SORT* than an implementation but the latter is also much harder to use for subsequent reasoning because algorithm equivalence is more difficult than showing an algorithm satisfies some property). Rely/guarantee thinking retains this viewpoint in as far as it can but needs to face the extra abstraction of "interference" which it is argued is the essence of concurrency. The question which has to be addressed below (cf. Section 5) is whether the abstraction using relations is well chosen.[6]

The main motivation behind the inception [Jon81] of rely/guarantee conditions was the lack of compositionality in [Owi75]. Both multi-level decomposition and even changes of data representation work compositionally with rely/guarantee conditions. Thus, we have a design method that allows the designer to make and record design decisions in a stepwise form. As with introducing loops and sequential composition in sequential program design, definite design decisions to use parallel composition are difficult to undo in the sense that designers should avoid putting themselves back into the problems of equivalence proofs.

In concurrent program design, the issue of granularity is closely linked to compositionality: a guarantee condition must be respected at the level of granularity at which the final code executes. This, in fact, works well but it must be tackled with awareness. Making rash granularity decisions can necessitate locking of variables and this can destroy the performance advantages of parallel execution. In general, it is far better to avoid locking. Section 3.2 and the example in Section 3.3 offer interesting insight on the topic of granularity; further discussion can be found in [CJ07, Col08].

3 Abstract objects

Although this section covers what might be classed as well-known territory, there is in the first sub-section a useful warning about "clutter" in specifications and an indication of how abstraction can be used to avoid it. Perhaps most importantly for the analysis of auxiliary variables, a precise test is given for where complexity is actually "clutter". Section 3.2 moves on to an important link between data reification and rely/guarantee thinking.

3.1 Why use a relation if a retrieve function will do?

The story of using abstract data objects to obtain short and perspicuous specifications is traced in [Jon89]. In passing, I might comment that I am proud of having included data abstraction in the early book on VDM [Jon80] and of promoting it to its rightful place ahead of operation decomposition in the 1986 first edition of [Jon90]. The essence of the abstraction is to use, in a specification, data types that match the problem rather than the implementation. Typically, these are finite mathematical objects with pleasing algebras. It is revealing that quite diverse specification languages such as Z [Hay93], VDM and SETL [Ano09] all build on some form of sets, sequences, maps and records.

The use of the abstraction leaves the two questions of how good it is (as an abstraction) and how to get from the abstraction to the implementation. It is useful to tackle the second of these issues first for reasons that become clear below.

Given two descriptions of a collection of operations, one needs to be able to determine if they exhibit the same "behaviour". Peter Lucas faced this problem in looking at two operational models of the PL/I programming language: did they give the same semantics? In [Luc68], he used a "twin machine" proof. In essence, he defined a large machine with state elements from both descriptions and linked them by what we might call today a "gluing invariant"; he then proved that the combined machine preserved this data type invariant. The argument was then that either set of variables could be regarded as "ghost variables" and be erased without changing the behaviour. It is possible to argue that this was the mother and father of all auxiliary variable ideas! The contribution of [Jon70][7] was to capitalise on the fact that –in most cases– one model is more abstract than the other in that it "has less information". Where this is the case,

[5] Colin Stirling was interested in meta results more than usability in applications.

[6] There are those who argue that the root of the problem is, in fact, shared variable concurrency. Another of Tony Hoare's major contributions is, of course, the development of CSP [Hoa78, Hoa85]. Although the concept of communicating processes has yielded considerable insight into the nature of concurrency, it is by no means immune from interference. The interference just comes from communication. This is manifest in any process algebra in which shared variables can be simulated by a process that holds their current value.

[7] Far too much of the Vienna Lab's work was only published as technical reports.

it is reasonable to take the model with less information as the specification and simplify the reification proof by recording a function from the (more populous) implementation type back to the abstraction. In VDM, these were called "retrieve" functions because they extracted the abstraction from the details of the representation. This homomorphic idea is, of course, the same as in [Mil71][8] and [Hoa72]. The VDM rules for data reification include an "adequacy" proof obligation that determines whether there is at least one representation for each abstract state. Failures of adequacy frequently indicate missed invariants. We have not laboured data type invariants here — although extremely important, they have little to add to the discussion of "auxiliary variables". Suffice it to say that an additional heuristic is to prefer –of two isomorphic models– the one with simpler invariants. Heinrich Hertz wrote:

> "Various models of the same objects are possible, and these may differ in various respects. We should at once denote as inadmissible all models which contradict our laws of thought. We shall denote as incorrect any permissible models, if their essential relations contradict the relations of the external things. But two permissible and correct models of the same external objects may yet differ in respect of appropriateness. Of two models of the same object ... the more appropriate is the one which contains the smaller number of superfluous or empty relations; the simpler of the two."

The question of how good an abstraction is can now be addressed. It is easy to see that the retrieve function idea offers a partial ordering on models: model S is at least as abstract as I if there is a retrieve function from $I \rightarrow S$. There are, however, "equivalently abstract" models where there are retrieve functions in both directions. The question of how to know if one has found one of the "sufficiently abstract" models is settled in [Jon77] by saying that a specification is "biased" if the equality on the underlying states cannot be computed in terms of the operations of the type. (Worked examples are provided in [Jon80, Chapter 15] and [Jon90, Section 9.3]).

One could, in fact, get by with a biased specification by adding "ghost variables" to the implementation and later erasing them as in Lucas' twin machine proofs but there is a real sense in which abstraction can be listed as a virtue — a virtue for which there is a precise test. This situation led to a certain smugness in the model-oriented camp. One should never be smug! The claim that any biased specification could, and should, be replaced by one that is appropriately abstract was challenged by Lyn Marshall who was writing a large VDM specification (of the then standard of the "Graphics Kernel System"). Lyn claimed that bias was required in her specification. After much discussion, she was proved right. The problem boils down to there being non-determinacy in the specification that, once a designer makes design choices, obviates the need for some state values. Together with Tobias Nipkow we boiled this down to a tiny example that illustrated the point beautifully. In parallel (and partially in cooperation) with researchers from Oxford, Tobias came up with a data refinement rule that he proved to be complete in a useful sense (see [Nip86, Nip87, HHH$^+$87]). This rule uses a relation and thus evokes shades of the twin machine idea. More details of this story are given in [Jon89] (and both rules are described in (even the first edition of) [Jon90]); what matters for the discussion of auxiliary variables in concurrency are the points:

- the essence of design is that it introduces "bias" (cf. decisions made in decomposition of operations)
- but for specifications, *prefer the simpler model*
- abstraction should be used to avoid bias because equivalence is harder than reification
- there is a test for "goodness"
- where there is a specific technical problem, it might be possible to devise a new proof method

3.2 Linking rely/guarantee with reification

There is a very interesting connection between rely/guarantee development and data reification. Surprisingly, this was not made explicit in any of the early rely/guarantee proofs. In fact, as far as I'm aware, the first written reference is in [Jon07]. The observation is that *often* obligations from guarantee conditions can only be realised without excessive locking by choosing a clever data representation. So, just like the comment on non-determinacy being a good abstraction of design choices, guarantee conditions are a way of postponing a design decision. Of course, postponement can be perilous if the designer has no idea how to solve the problem.

A very simple example can be made of the *FINDP* problem for Sue Owicki's thesis [Owi75]. The top level specification states that the task is to find the minimum value of an array index such that the indexed element satisfies some predicate.[9] A sequential algorithm simply searches the array indices from

[8] The community was denied a journal version of this paper because it was rejected by JACM.

[9] In the case that there is no such value, the program can either return an indicator or add a sentinel that does have the property.

the minimum index upwards. The interest is how to use concurrency — an "n-fold" split of the indices is no more technically difficult, but the description is shorter if two processes are considered. Suppose one process searches the odd –and the other the even– indices. If these two processes do not communicate, it is easy to see that there is a trap where the parallel algorithm could be slower than the sequential alternative. To avoid this, either process should terminate if its sibling process has detected an array element with a lower index that has the required condition. An obvious abstraction is to have the two processes share a variable, say t, in which they record any index for which it is detected that the array element at that index satisfies the given predicate. At this level of abstraction, both processes need a sub-operation whose specification involves setting t to the minimum of the current value of t and some variable local to that process. Mental warning lights (should) flash when writing down a rely condition that specifies that neither process can live with the other lying (in the sense that they temporarily reduce t then increase it again): the process on which this dishonesty is inflicted might have terminated prematurely. At this level of abstraction, it is not difficult to describe the honesty requirement in a rely condition.

One possible implementation strategy is for both processes to lock t when they need to access it but this could also make a concurrent implementation slower than the simpler sequential approach. In this example, it is not difficult to spot that equipping each process with a local variable means that t can be reified to the minimum of these values. The troublesome guarantee condition of monotonic reduction is now trivial because each local variable is read but not written by the partner process.

The same story of the interplay of reification with satisfying guarantee conditions can be seen with the *SIEVE* example of Section 3.3 — but here it is more interesting:

- each of n processes removes elements from a set s
- assuming the designer does not want to lock s (it's big!)
- the designer must find a representation that helps realise rely/guarantee conditions $s \subseteq \overleftarrow{s}$
- the (less obvious) representation of s as a bit vector meets the need.

This example is spelled out in the next section.

Yet more interesting is the example discussed below in Section 5.1. Simpson's so-called "four-slot" implementation of Asynchronous Communication Methods (ACMs) is an intriguing and very clever piece of programming whose correctness is far from easy to prove. Even more challenging is the task of presenting the development and its formalisation in a way that conveys Simpson's contribution. The claim made in [JP08] is that this is achieved by the use of data reification combined with rely and guarantee conditions. Before more detail is given, Section 4 adds one more idea to our armoury.

The above three examples are identified in [Jon07] and are expanded on in [Pie09].

3.3 An example

The example in this section is a parallel version of the "Sieve of Eratosthenes" which finds all prime numbers –up to some required n– by removing composite numbers. The first reference that I am aware of to a concurrent version is [Hoa75].

We can use an abstract object containing a set of numbers to make the overall problem clear:[10]

$$\textit{post-PRIMES}(\overleftarrow{s}, s) \triangleq s = \{1 \leq i \leq n \mid \textit{is-prime}(i)\}$$

It is equally straightforward to make (and record) the decision to split *SIEVE* into two sequentially decomposed sub-operations: one for initialisation of s to contain all natural numbers up to the required limit(n); the other operation removes all composites from s. One might think that the specification of the sub operations *INIT* and *SIEVE* is best written as follows:

$(\textit{INIT}; \textit{SIEVE})$ **satisfies** *PRIMES*

$$\textit{post-INIT}(\overleftarrow{s}, s) \triangleq s = \{1, \ldots, n\}$$

$$\textit{pre-SIEVE}(s) \triangleq s = \{1, \ldots, n\}$$
$$\textit{post-SIEVE}(\overleftarrow{s}, s) \triangleq \textit{post-Primes}(\overleftarrow{s}, s)$$

But this would be a mistake — in two ways *SIEVE* is being too tightly specified to fit its context. A better split is to recognise that sieving can be performed on any set and that the removal need not necessarily end up with all primes (consider the case where the starting state for *SIEVE* is the empty set) — so:

$$\textit{pre-SIEVE} \triangleq \textbf{true}$$
$$\textit{post-SIEVE}(\overleftarrow{s}, s) \triangleq s = \overleftarrow{s} - \bigcup\{\textit{mults}(i) \mid 2 \leq i \leq \lfloor\sqrt{n}\rfloor\}$$

[10] VDM notation [Jon90] is used; the only item that might be unfamiliar is the use of $\overleftarrow{s}$ for the initial (and undecorated s for the final) state in relational post conditions. Furthermore, the predicate *is-prime* should be obvious and the function *mults* delivers the set of multiples (by 2 and above) of its argument.

creates a much cleaner separation of *SIEVE* from its context. Here, in the sequential case, the earlier definition might not be disastrous but in more complex cases it could be; moreover, the issue of separation is certainly one that is magnified by the move to concurrency.

As pointed out in Section 2, the step of introducing sequential constructs as in $PRIMES = (INIT; SIEVE)$ marks a clear design decision. If a sequential implementation is sought, it is now straightforward to make –and justify– further design decisions for *SIEVE* to use nested loops as in:

for $i \leftarrow \cdots$
  post-BODY: $s = \overleftarrow{s} - mults(i)$
  for $j \leftarrow \cdots$
    $s \leftarrow s - mults(i * j)$

The fact that repeated execution of the removal of composites $(i * j)$ eventually ensures the post condition $s = \overleftarrow{s} - mults(i)$ relies on there being no interference and this is a reasonable assumption in sequential programs.

The real interest here is to use the design of a concurrent *SIEVE* to illustrate points about the trade-off between the various predicates in a rely/guarantee specification. So, implementing *SIEVE* as:

$||_i REM(i)$

One might first try copying the idea from *post-BODY* above and write:

$post\text{-}REM(\overleftarrow{s}, s) \triangleq s = \overleftarrow{s} - mults(i)$

but this exact definition of the elements to be removed cannot be achieved in the situation where it is the intention that sibling processes are removing elements of s. This points to the idea of specifying in the post condition only that certain elements must be absent after $REM(i)$ has executed:

$post\text{-}REM(\overleftarrow{s}, s) \triangleq s \cap mults(i) = \{\}$

A moment's thought however indicates that even the lower bound on removal of elements can be achieved in the presence of arbitrary interference — the reliance on the fact that no sibling will re-insert deleted elements can be easily recorded in

$rely\text{-}REM(\overleftarrow{s}, s) \triangleq s \subseteq \overleftarrow{s}$

An attempt to use (an n-ary form of) the *Par-I* proof rule of Section 2.2 shows that too much was given away in the above relaxation to the post condition of *REM*: this lower bound on removal could in fact be achieved by setting s to the empty set which will clearly not lead to satisfying the specification of the overall *SIEVE* process. So the guarantee condition can be used to outlaw such over zealousness

$guar\text{-}REM(\overleftarrow{s}, s) \triangleq (\overleftarrow{s} - s) \subseteq mults(i) \wedge \cdots$

This pattern of shifting conditions that might fit the post condition of a sequential process back into the guarantee conditions of concurrent specifications is both common and useful.

Finally, since the sibling processes of $REM(i)$ are actually twins, the guarantee condition is completed by conjoining a copy of the rely condition (cf. *Par-I*) giving the overall specification of each $REM(i)$ to be

$REM(i)$
pre true
rely $s \subseteq \overleftarrow{s}$
guar $(\overleftarrow{s} - s) \subseteq mults(i) \wedge s \subseteq \overleftarrow{s}$
post $s \cap mults(i) = \{\}$

Thus far, the example has been used to illustrate both the fact that –even in sequential programs– care in divorcing a sub-operation from its context produces a more useful specification; and furthermore the sometimes delicate trade-off between the predicates used in the description of a concurrent program (not surprisingly, this balance is more interesting in complex examples — see [CJ00]).

The largest lesson from this example is however to illustrate the point made in Section 3.2. To set the scene, Hoare writes in his discussion of the problem in [Hoa75]:

> Of course, when a variable is a large data structure, as in the example given above, the apparently atomic operations upon it may in practice require many actual atomic machine operations. In this case an implementation must ensure that these operations are not interleaved with some other operation on that same variable"

As Hoare goes on to mention, placing all updates to s in critical regions is certainly one way of ensuring that the guarantee condition is met but it is an implementation that is unlikely to give high performance.

An alternative is to choose a data representation in which such updates can be made safely without locking. As hinted in Section 3.2, this can be achieved by representing the set as a vector of n bits.

To emphasise how subtle the issue of granularity can be, it is worth mentioning that there could still be a dependency on the machine architecture if the implementer packs bits in such a way that it is impossible to set one bit atomically.

4 Abstraction using a "fiction of atomicity"

Just as post conditions abstract from "how" to achieve an objective, and abstract data objects offer a way to abstract from details of machine representations honed for efficiency, a "fiction of atomicity" can be a powerful abstraction that achieves far more perspicuous descriptions than is possible when considering the actual interleaving of steps in an algorithm. Far from being a completely new idea, this very convenient fiction is well known in computing. In particular, it is key to the whole idea of database transactions: a user of a DBMS can picture transactions as "atomic" and it is the responsibility of the system both to overlap transactions and to disguise that fact that it has done so. (Furthermore, it has to do so in the presence of failures in hardware.)

Although by no means the first attempt, what is perhaps unusual is the extent to which [Jon07] attempts to elevate the atomicity abstraction –and approaches around splitting– to tools to be used alongside, and in concert with, the other development approaches of Sections 2 and 3. It is indicated below that compositionality can be achieved by deploying rely/guarantee conditions.

Some of the ideas here were enhanced by two Schloß Dagstuhl workshops[11] on the topic of atomicity. The objective was to bring together researchers from different fields that use atomicity in one form or another. In particular [JLRW05] draws up a "manifesto" that compares and contrasts views and approaches from database, hardware, fault-tolerance and formalism research.

The genesis of my own research on splitting atoms was an acceptance that rely/guarantee reasoning was bound to be heavier than proofs in terms of pre and post conditions. More generally, it is inevitable that development of algorithms that interfere will be more difficult than those that (appear to) run in isolation. The higher level advice must be to use concurrency only where it is really required either by the problem itself or to make really telling performance gains.

The acceptance that one needed to be able to limit the areas of reasoning using rely/guarantee conditions came when trying to write a joint paper with Ketil Stølen after he submitted his PhD thesis [Stø90]. Our paper was never completed — but we learned a lot. In fact, it was the start of my search for ways of limiting interference.[12] In particular, the power of object oriented (OO) programming languages to control interference appeared promising: Pierre America's POOL language [Ame89] proved to be a good basis for further investigation.

The avenue I followed in the $\pi o \beta \lambda$ research was to offer "equivalence rules" that facilitated transforming OO programs with large (atomic) steps into equivalent programs where many objects were active concurrently. The argument was that, if there were many processors to run threads for different objects, performance would improve. The notion of "equivalence" was, of course, crucial: the $\pi o \beta \lambda$ language was designed to be expressively weak so that its observations were a sensible approximation to what a user might want. The work on proving these equivalences correct showed that being precise about acceptable observations was crucial. This research (and pointers to more detailed papers — especially on the semantics to justify the equivalences) is summarised in [Jon96].

The general proposal in [Jon07] is to use the "fiction of atomicity" as an abstraction with the corresponding development method called "splitting (software) atoms safely" (or "atomicity refinement"). In the $\pi o \beta \lambda$ proposal, the fission was supported by equivalence rules. More generally, if one starts with an abstraction of atomicity, it is essential to have a notion of observation power to determine whether decomposed (and overlapping) sub-operations offer the expected behaviour to an observer. After all, to an all powerful observer, the behaviour is manifestly different.

Recall also the emphasis in earlier sections on compositionality: there will be cases where splitting can occur at more than one stage of design. If we look for cases where separation is not the answer, it will clearly be necessary to have a handle on any potential interference that can occur with the decomposed sub-operations.

The foregoing observations all point to reasons to investigate how useful rely and/or guarantee conditions can be in atomicity refinement. The general argument looks quite strong: guarantee conditions state what the outside world can rely on — any decomposition must preserve this but is at liberty to decompose operations on any variables not so constrained. To give a trivial example, an operation whose post condition requires the value of a variable x to be increased by, say, 10 can be decomposed into any number of assignments whose accumulated effect is that increment if there are no guarantee conditions on x; if there

[11] Some papers from the 2004 workshop appear in *Journal of Universal Computing Science*, Vol. 11, No.5; similarly (and in the same journal), Vol. 13, No. 8 for the 2006 workshop.

[12] My valued friends working on Separation Logic [Rey00, IO01] for concurrency [Rey02, O'H07, Bro07, OYR09, PB05] should remember that this was back in the early 1990s.

is a guarantee condition that x increases monotonically some decompositions such as $x \leftarrow x-2; x \leftarrow x+12$ are ruled out.

The far more complex example in Section 5.1 offers more evidence for the usefulness of rely and guarantee conditions in atomicity refinement. It is perhaps worth contrasting this approach to the interesting "event refinement" in [Abr10]. The need to introduce the concept of some events "refining skip" is a consequence of not having rely conditions.

5 Limiting the use of auxiliary variables

The essence of the argument here is that abstraction is a better tool than auxiliary variables. The example in Section 5.1 not only illustrates the techniques outlined above; the frank account of two attempts to present an informative development underlies the conclusions in Section 5.2.

5.1 Development of ACMs

This section indicates how the ideas in Sections 2–4 are used in concert to provide a rational reconstruction of a very intricate algorithm. A development of Simpson's "four-slot" implementation of "Asynchronous Communication Mechanisms" (ACMs) is given in [JP08]; the fact that the authors discovered flaws in the original development and the investigation of whether auxiliary variables are needed to complete the proof is of relevance to the key message of the current paper.

The objective in writing yet another paper on Simpson's algorithm was precisely to provide insight as to what is going on in its design. The extremely tight code is difficult to prove correct, but somehow treatments like [Sim97, Hen04] (and even the more recent [AC08, BA08]) fail to utilise fully abstraction in their proofs.

ACMs are used to communicate values between two processes which are asynchronous in the sense that it is not allowed for either to hold up the other. Thus, in the ACM world, locking a shared variable is certainly not an option. A little thought shows that it is possible for the *Read* process to see the same value more than once and for the reader to miss values that are written. There is a requirement that the reader gets the freshest reasonable value and most importantly that the reader never sees a value older than one already read.

The first challenge is to provide a specification to act as a reference point that is clear enough that a user can have confidence that he/she understands the properties. We based the specification in [JP08] around an abstraction (Σ^a) of a sequence of all *Value*s written. This is clearly redundant but precisely in the way discussed in Section 3.1: the redundancy admits non-determinacy and the state can be specialised once the choices are narrowed.

More controversially, the specifications of *Read* and *Write* in [JP08] are each split into two phases. In the terms of the current paper, this is a design commitment: it would be messy to justify further development that was not a specialisation of these phases. In fact, the split *start-Write*, *commit-Write*, *start-Read* and *end-Read* not only holds good for the development, it also makes the behaviour of reader/writer values easy to comprehend. Furthermore, this "phasing" makes it possible to record simpler rely and guarantee conditions than would work with the unsplit operations. Another key aid to clear rely and guarantee conditions is the use of VDM's read/write frames.

The first step of development in [JP08] shows that it is not necessary for the state to hold the whole history of values input. This is a classic example of using Nipkow's rule [Nip87] to show that an otherwise "inadequate" representation gives acceptable behaviour. The states Σ^i are mappings from some arbitrary index set X to the *Values* to be transmitted. In fact, if X is the natural numbers, this model could be identical to Σ^a but the intermediate step establishes properties required of the set X. It is clear from the formal descriptions that the cardinality of X must be at least three. The key to Simpson's choice of *four* slots is actually about communication between the *Read* and *Write* threads: the only atomicity assumption is on the setting and reading of single control bits.

At the Σ^i stage of development, there are still unacceptable atomicity assumptions on updates to the state. The step to Simpson's actual design (Σ^r) uses the fact that four variables (with clever control flags) suffice. The final essence of Simpson's inspiration is presented (cf. Section 3.2 above) as choosing a representation that makes the guarantee conditions realisable. The residual atomicity assumptions are limited to the ability to update single bit flags without corruption.

Unfortunately, when filling in more detailed proofs to write a journal version of [JP08], we detected flaws in two of the proofs. Initially, I could only see how to fix these by adding deprecated auxiliary variables and this led me on this odyssey to understand how to constrain their use. The revised development has been submitted to a journal and a version (pre refereeing) is available at [JP09].

One flaw in the development in [JP08] was a post condition which stated that a local variable could acquire either the initial or final value of a variable changed by the other thread. In fact, because the

relative progress of the threads is not synchronised, the other thread could potentially change this value many times. A short term fix is to use an auxiliary variable to record all possible values. This is the resolution presented in [Pie09]. The solution in [JP09] is more radical: a new specification concept of the set of possible values of a variable is introduced. Thus, in a post condition, $\widehat{var}$ is a set of all values that this variable has during the execution of the specified operation. This concept not only avoids the need to introduce an auxiliary variable in this case, it also provides a specification concept that is of use in other circumstances.

The other place where [Pie09] and [JP09] differ is the way they handle the crucial avoidance of a clash of the *Read* and *Write* processes on a single position in the four slots. This also points to a surprising conclusion about the respective strengths of the rely/guarantee approach and separation logic.

The classical idea of *mutual exclusion* is one of the core concepts in concurrency. Interestingly, Simpson's algorithm does not fit the classic pattern. One reason for this is that mutual exclusion leads to blocking which is inimical with the requirements of ACMs. There is however an issue that might be called "mutual data exclusion": the *Read* and *Write* processes must not interfere on the same cell. In [Pie09], this is proved by adding auxiliary variables at the Σ^r level; the approach in [JP09] establishes the relevant conditions at the more abstract level (Σ^i). This is a useful illustration of the proposal to use abstraction in preference to auxiliary variables.

The deeper aspects of this are even more interesting. John Reynolds pointed out verbally at MFPS in 2005: "separation logic lets one reason about avoiding races; rely/guarantee conditions support reasoning about racy programs". Like so much that Reynolds says, this shows great insight but the example in hand points to a further observation. Simpson's final code does not "race"; in fact, the whole point is to avoid conflicting reads/writes. The use of rely/guarantee conditions makes it possible to present a key abstraction in which there appear to be races.

5.2 Auxiliary variables: a position

The expressive weakness of rely/guarantee conditions is conceded in Section 2 but it is also made clear in discussing pre/post condition specifications that there is always a trade-off between being able to express everything and having a tractable method that makes good engineering sense: abstraction must have a part. In particular, compositionality dictates that detail must be postponed by abstraction.

Coming back to auxiliary variables, it is worth looking in more detail at what we have learnt. It should be clear from Sections 2 and 3 that it is easier to show that a reification step fills in detail than to prove that two detailed algorithms are equivalent. Specifically, with respect to abstract data types, one feels on solid ground if there is a precise test for unnecessary detail.

Before going further, it is worth pinning down the origin of the expressive weakness: is it a facet of rely/guarantee conditions? I concede that I assumed this to be the case for some time. In fact, this is incorrect! If one considers Owicki's "Einmischungsfrei" proof obligation [Owi75], it requires that no step of s_r can interfere with the proof of a step between any two statements of s_l. In fact, the issue is already there in the earlier approach proposed in [AM71]. Even though neither of these methods is compositional in the sense set out in Section 2, they have no way of describing different behaviour during the progress of a sibling process.[13]

So, given the widespread expressive weakness, is it acceptable to plug it with auxiliary variables and can we put precise limits on their use? Two further data points are given before I, tentatively, give positive answers to both questions.

The apparent weakness of rely conditions has an interesting role in the soundness proof for rely/guarantee rules given in [CJ07]. Essentially, the fact that a rely condition must be broad enough to capture *any* interference means that it can be used in the induction proof of a parallel construct even though interference can come either from a sibling process *or from any contextual process.* (The language in [CJ07], unlike that in the Isabelle-checked proofs of [Pre01], permits nested parallelism; our paper and [Col08] also accommodate fine grained interleaving of expression evaluation.)

What is the evidence that one can *not* avoid auxiliary variables? It is plausible that a process will go through phases in which different conditions are guaranteed. For example, one process might, under the control of a flag p, guarantee

$$p \;\Rightarrow\; \overleftarrow{x} \leq x$$
$$\neg\, p \;\Rightarrow\; \overleftarrow{x} \geq x$$

Although this might be viewed as internal information of the process, if the joint behaviour depends on it, the rely condition must record it. Actually, so far, there is no problem in its recordability. In fact, it is

[13] This is the source of the difficulty in [Owi75] in proving that two parallel instances of $< x \leftarrow x + 1 >$ achieve the obvious result.

reasonable to see it as an extension of the "phasing" idea of Section 5.1. The problem comes where there is no convenient variable p to demarcate the phases.

The position taken here is that, in such circumstances, it is reasonable to add an auxiliary variable in place of the missing p. This introduces no more dependence on the other process, no more loss of compositionality, than if the variable were actually present in the first place. Clearly, parallelism that does not depend on distinct phasing in its sibling processes is more robust but, in examples like that in Section 5.1, the mutual dependencies are very intimate.

In spite of conceding this use of auxiliary variables, in all cases, I would prefer a better abstraction to the use of such coding tricks. The key reason for this preference is that it is difficult to retain compositionality without severe constraints on the use of ghost variables.

There is one remaining question –prompted by the history above of reification of data types– and that is whether the whole issue of auxiliary variables points to new proof rules and/or languages. I venture to suggest that process algebras will not resolve the issue. Nor do I expect anything like our current temporal logics to be the source of a solution; but Amir Pnueli[14] who heard the talk from which this paper is derived did make the point that past-time temporal logic could cover some cases.

Acknowledgements

I have had the pleasure of knowing Tony Hoare since the 1960s and my DPhil research was done under his supervision in 1979–81. The process of editing "Essays" [HJ89] enhanced our collaboration after I left Oxford. As was said (repeatedly) at the Cambridge meeting in April, 2009 Tony has inspired and supported many of us over decades.

This paper was not actually presented at the Cambridge meeting to mark Tony's birthday because Bill Roscoe and I had held ours as "makeweights" in case any speakers could not get there. The material was actually presented at the PSY workshop at CAV Grenoble (June 2009).

I am grateful for comments on drafts of this paper from Joey Coleman, Linas Laibinis, Thai Son Hoang and Bill Roscoe; and to my ever-patient proof reader Ms. Allison. This is also a nice opportunity to give belated thanks to Schloß Dagstuhl for (among other pleasurable visits) the two on "Atomicty". The staff in Dagstuhl, the environment and the stimulating participants always make trips there rewarding and refreshing.

My research is currently funded by the EU "Deploy" project, the (UK) EPSRC "TrAmS" platform grant and the ARC project (that brings together Ian Hayes, Keith Clark, Alan Burns and myself) "Time Bands for Teleo-Reactive Programs".

[14] Amir was of course co-author of the first paper to apply rely/guarantee thinking to temporal logic [BKP84].

References

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

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

[Acz82] P. Aczel. A note on program verification. (private communication) Manuscript, Manchester, January 1982.

[AM71] E. A. Ashcroft and Z. Manna. Formalization of properties of parallel programs. In B. Meltzer and D. Michie, editors, *Machine Intelligence, 6*, pages 17–41. Edinburgh University Press, 1971.

[Ame89] Pierre America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4), 1989.

[Ano09] Anon. SETL: main page, Oct 2009. www.setl-lang.org.

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

[Bic95] Juan Bicarregui. *Intra-Modular Structuring in Model-Oriented Specification: Expressing Non-Interference with Read/Write Frames*. PhD thesis, Manchester University, 1995.

[BKP84] H. Barringer, R. Kuiper, and A. Pnueli. Now you can compose temporal logic specification. In *Proceedings of 16th ACM STOC*, Washington, May 1984.

[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).

[CJ00] Pierre Collette and Cliff B. Jones. Enhancing the tractability of rely/guarantee specifications in the development of interfering operations. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction*, chapter 10, pages 277–307. MIT Press, 2000.

[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.

[Col08] Joseph William Coleman. *Constructing a Tractable Reasoning Framework upon a Fine-Grained Structural Operational Semantics*. PhD thesis, Newcastle University, January 2008.

[Dij68] E. W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[DS90] Edsger W Dijkstra and Carel S Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990. ISBN 0-387-96957-8, 3-540-96957-8.

[Flo67] R. W. Floyd. Assigning meanings to programs. In *Proc. Symp. in Applied Mathematics, Vol.19: Mathematical Aspects of Computer Science*, pages 19–32. American Mathematical Society, 1967.

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

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

[HHH+87] C. A. R. Hoare, I. J. Hayes, J. He, C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sørensen, J. M. Spivey, and B. A. Sufrin. The laws of programming. *Communications of the ACM*, 30:672–687, 1987. see Corrigenda in *ibid* 30:770.

[HJ89] C. A. R. Hoare and C. B. Jones. *Essays in Computing Science*. Prentice Hall International, 1989.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, October 1969.

[Hoa71] C. A. R. Hoare. Proof of a program: FIND. *Communications of the ACM*, 14:39–45, January 1971.

[Hoa72] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[Hoa75] C.A.R. Hoare. Parallel programming: An axiomatic approach. *Computer Languages*, 1(2):151–160, June 1975.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, August 1978.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[IO01] S. Isthiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *28th POPL*, pages 36–49, 2001.

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

[Jon70] C. B. Jones. A technique for showing that two functions preserve a relation between their domains. Technical Report LR 25.3.067, IBM Laboratory, Vienna, April 1970.

[Jon72a] C. B. Jones. Formal development of correct algorithms: an example based on Earley's recogniser. In *SIGPLAN Notices, Volume 7 Number 1*, pages 150–169. ACM, January 1972.

[Jon72b] C. B. Jones. Operations and formal development. Technical Report TN 9004, IBM Laboratory, Hursley, September 1972.

[Jon77] C. B. Jones. Implementation bias in constructive specification of abstract objects. typescript, September 1977.

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

[Jon81] C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981. Printed as: Programming Research Group, Technical Monograph 25.

[Jon89] C. B. Jones. Computer-aided formal reasoning for software design, March 1989. talk at: TAPSOFT'89, Barcelona.

[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] Cliff B. Jones. The early search for tractable ways of reasonning about programs. *IEEE, Annals of the History of Computing*, 25(2):26–49, 2003.

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

[Jon09] Cliff B. Jones. Annotated bibliography on rely/guarantee conditions, Oct 2009. http://homepages.cs.ncl.ac.uk/cliff.jones/ftp-stuff/rg-hist.pdf.

[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.

[JP09] Cliff B. Jones and Ken G. Pierce. Elucidating concurrent algorithms via layers of abstraction and reification. Technical Report CS-TR-1166, School of Computing Science, Newcastle University, 2009.

[Kin69] J. C. King. *A Program Verifier*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, 1969.

[Luc68] P. Lucas. Two constructive realizations of the block concept and their equivalence. Technical Report TR 25.085, IBM Laboratory Vienna, June 1968.

[Mil71] R. Milner. An algebraic definition of simulation between programs. Technical Report CS-205, Computer Science Dept, Stanford University, February 1971.

[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.

[Owi75] S. Owicki. *Axiomatic Proof Techniques for Parallel Programs*. PhD thesis, Department of Computer Science, Cornell University, 1975.

[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.

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

[Pre01] Leonor Prensa Nieto. *Verification of Parallel Programs with the Owicki-Gries and Rely-Guarantee Methods in Isabelle/HOL.* PhD thesis, Institut für Informatic der Technischen Universitaet München, 2001.

[Rey00] J. C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In Jim Davies, Bill Roscoe, and Jim Woodcock, editors, *Millennial Perspectives in Computer Science*, pages 303–321, Houndsmill, Hampshire, 2000. Palgrave.

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

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

[Sti86] C. Stirling. A compositional reformulation of Owicki-Gries' partial correctness logic for a concurrent while language. In *ICALP'86*. Springer-Verlag, 1986. LNCS 226.

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