Newcastle University

COMPUTING SCIENCE

Balancing Expressiveness in Formal Approaches to Concurrency

Cliff B. Jones

TECHNICAL REPORT SERIES

No. CS-TR-1394 September 2013

TECHNICAL REPORT SERIES

No. CS-TR-1394 September, 2013

Balancing Expressiveness in Formal Approaches to Concurrency

C.B. Jones

Abstract

One might think that specifying and reasoning about concurrent programs would be easier with more expressive languages. This paper questions that view. Clearly too weak a notation can mean that useful properties either cannot be expressed or their expression is unnatural. But choosing too powerful a notation also has its drawbacks since reasoning receives little guidance. For example, few would suggest that programming languages themselves provide tractable specifications. Both rely/guarantee methods and separation logic(s) provide useful frameworks in which it is natural to reason about aspects of concurrency. Rather than pursue an approach of extending the notations of either approach, this paper starts with the issues that appear to be inescapable with concurrency and -only as a response thereto- examines ways in which these fundamental challenges can be met. Abstraction is always a key tool and its influence on how the key issues are tackled is examined in each case.

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

Bibliographical details

JONES, C.B.

Balancing Expressiveness in Formal Approaches to Concurrency
[By] C.B. Jones

Newcastle upon Tyne: Newcastle University: Computing Science, 2013.

(Newcastle University, Computing Science, Technical Report Series, No. CS-TR-1394)

Added entries

NEWCASTLE UNIVERSITY
Computing Science. Technical Report Series. CS-TR-1394

Abstract

One might think that specifying and reasoning about concurrent programs would be easier with more expressive languages. This paper questions that view. Clearly too weak a notation can mean that useful properties either cannot be expressed or their expression is unnatural. But choosing too powerful a notation also has its drawbacks since reasoning receives little guidance. For example, few would suggest that programming languages themselves provide tractable specifications. Both rely/guarantee methods and separation logic(s) provide useful frameworks in which it is natural to reason about aspects of concurrency. Rather than pursue an approach of extending the notations of either approach, this paper starts with the issues that appear to be inescapable with concurrency and -only as a response thereto- examines ways in which these fundamental challenges can be met. Abstraction is always a key tool and its influence on how the key issues are tackled is examined in each case.

About the authors

Cliff B. Jones is currently Professor of Computing Science at Newcastle University. As well as his academic career, Cliff has spent over 20 years in industry. His 15 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. 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). He is now applying research on formal methods to wider issues of dependability. He is PI of two EPSRC-funded responsive mode projects “AI4FM” and "Taming Concurrency", CI on the Platform Grant TrAmS-2 and also leads the ICT research in the ITRC Program Grant. 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).

Suggested keywords

CONCURRENCY
RELY/GUARANTEE
SEPARATION LOGIC

Balancing Expressiveness in Formal Approaches to Concurrency

Cliff B. Jones
School of Computing Science, Newcastle University, NE1 7RU, UK

September 19, 2013

Dedication: Wlad Turski 1938–2013

Abstract

One might think that specifying and reasoning about concurrent programs would be easier with more expressive languages. This paper questions that view. Clearly too weak a notation can mean that useful properties either cannot be expressed or their expression is unnatural. But choosing too powerful a notation also has its drawbacks since reasoning receives little guidance. For example, few would suggest that programming languages themselves provide tractable specifications. Both rely/guarantee methods and separation logic(s) provide useful frameworks in which it is natural to reason about aspects of concurrency. Rather than pursue an approach of extending the notations of either approach, this paper starts with the issues that appear to be inescapable with concurrency and –only as a response thereto– examines ways in which these fundamental challenges can be met. Abstraction is always a key tool and its influence on how the key issues are tackled is examined in each case.

1 Introduction

Concurrency has been an issue in computing for a long time but it becomes ever more pressing as hardware evolves: the numbers of "cores" per chip is increasing and "weak memory" architectures are being used. Furthermore computation increasingly involves distributed data;

Concurrency always magnifies difficulties. The near impossibility of achieving acceptable sequential programs without using formal methods becomes absolute in the presence of concurrency. Although this points to deploying apposite formalisms, making them tractable is far more challenging than for purely sequential programs. In particular, the important property of compositionality (see Section 2.1) is harder to achieve. The central concern of this paper is "expressiveness" and this certainly becomes more delicate with concurrency.

1.1 Expressiveness

It can be useful to limit the expressive power of formal notations in order to make them more tractable. Of course, notations are only useful if they can express something of interest so a balance must be sought. An obvious example is the use –in programming languages– of type annotations that provide redundant assertions whose consistency with the rest of a program is decidable and can be checked at compile time.

This paper presents a study of two key issues in shared-variable concurrency: interference and separation. It reviews two well-known approaches and considers how they support reasoning about the aforementioned issues. It is argued that part of the usefulness of the existing methods derives precisely from limitations to their expressiveness. Furthermore, cautions are offered about attempts to take notations –that serve one purpose well– and to artificially bend them in an attempt to express other concepts. New directions for both existing approaches are indicated in the hope that understanding what is really going on in their methods will lead to new ways of combining the underlying concepts.

Another example of where expressive limitations appear to contribute to usefulness can be seen in writing specifications themselves. In say Hoare logic [Hoa69], post conditions are logical expressions that express properties of required results. Several observations can be made here:

- the use of logical operators (e.g. conjunction and negation) makes it possible to extend the range of concepts that can be expressed by the basic operators on types (e.g. GCD in terms of multiplication; sorting in terms of list operators)

- in contrast, the use of sequencing and iteration is what makes programming languages "Turing complete"; such imperative operators are normally avoided in specifications because proving the equivalence of programs is harder than showing that programs satisfy specifications

- there is a minor –but indicative– contrast between methods that use post conditions that are relations (between initial and final states) and those that attempt to get by with post conditions of the final state alone; the latter approach is forced to use free variables to *express* what are intuitively relations (cf. Section 5.3).

The widespread use of Hoare logic indicates that it represents a good balance between expressiveness and tractability.

It is worth taking one last example where limitations on expressiveness can be seen as positive. The topic is data abstraction and it is almost a *leitmotiv* of the current paper playing important roles in particular in Sections 4.1 and 4.2. Perhaps the point can be best made by an anecdote about where data abstraction was not fully used.. The committee that produced the ECMA/ANSI formal definition of PL/I [ANS76] was persuaded to adopt the basic ideas of formal modelling but was nervous of the acceptance by practitioners of abstractions like sets being used in the state of the model; consequently sequence types were employed where the natural model would have used sets. Readers of the resulting PL/I standard who wish to know if the ordering property of a particular sequence has any observable effect are therefore forced to examine all 300 pages of the document; the use of a set type would have been made it immediately apparent in the few pages of state description that no use of ordering was possible. The general point is that abstract objects can be used deliberately in specifications to limit the properties that can be expressed (e.g the elements of a set are not ordered) and that this can make for a clearer specification.

The industrial trends listed at the beginning of this section point to studying shared-variable concurrency but this should not be taken as an argument that communication-based concurrency is uninteresting. Two of the significant approaches to shared-variable concurrency are "Rely/Guarantee thinking" (normally abbreviated below as "R/G") and Separation Logic (abbreviated as "SL"). More is said about both R/G and SL (including pointers to source references) when the issues to which they appear to respond are presented — but the current argument is to view their respective expressive limitations positively: their expressiveness indicates which issues they discuss well and should not be viewed as a prompt to bend useful methods to tasks outside their natural purview.

The analysis in this paper is an early outcome of an attempt to devise balanced expressive power and provide one or more notations in which it is natural to reason about key issues in concurrency.

1.2 Structure of the paper

The examples above set out a general case for regarding expressive weakness in a positive light; the remainder of the paper specialises this argument to concurrency. Sections 2 and 3 introduce "issues" and then review notations for reasoning about the issues. Peter O'Hearn proposes a useful dichotomy around "data races" in [O'H07] arguing that SL is a natural way of showing race freedom whereas R/G might be the more natural tool for "racy" programs. This useful observation is refined in Sections 2.2 and 4.1.

Section 4 moves towards a goal that two new projects have set themselves: to take inspiration from R/G and SL and to look for new ways of deploying their fundamental insights –together with "abstraction"– to devise one or more new methods. Interestingly, abstraction appears to subvert O'Hearn's neat dichotomy (see both Sections 4.1 and 4.2).

Section 5 broadens the discussion both by listing some other issues and referring to additional approaches.

2 Reasoning about interference (race tolerance)

The most fundamental issue with concurrency is interference.[1] Data races occur when two or more processes can refer to the same data. The easiest case

[1] Although this might be more obvious with shared-variable concurrency, it is easy to reproduce in the communication-based approach.

to present is that of normal, named, variables to which multiple processes read and write values. Even if assignment statements were to be executed atomically, $x \leftarrow x + 1 \parallel x \leftarrow x * 2$ yields non-deterministic results. If –as in most programming languages– there is no way to enforce the atomicity of assignment statements, even more non-determinacy arises. Unguarded conflicts between reads and writes are also problematic and the same issue can be reproduced with heap variables which are referred to via their addresses.

In spite of this low-level unpredictability, it is possible to write programs that satisfy sensible specifications despite "interference". It is pointed out in Section 4 that dealing with interference (in specifications and designs) using abstract objects might be more useful than at the code level but the issue of interference is central to concurrency and any method that can help designers reason about interference warrants some attention.

Section 2.1 presents how R/G was originally formulated; after a motivating example, Section 2.3 sketches a more algebraic formulation of the rely/guarantee idea and revisits the example.

2.1 The original rely/guarantee 5-tuples

VDM was clearly part of the backdrop for the original R/G research. Among the ideas inherited from [Jon80] was the use of post conditions that were relations between initial and final states; using the resulting non-determinism to postpone design decisions; a commitment to proving total correctness (implementations must terminate when started in any state satisfying the pre condition of their specification); the preference for a "posit and prove" use of formalism; a strong commitment to data abstraction/reification; and judging any method against the test of "compositionality".

VDM [Jon90], B [Abr96] and Event-B [Abr10] can be classified as "posit and prove" approaches. They allow a designer to posit a design step which gives rise to "proof obligations" whose discharge justifies the design step. (The Rodin tools [Rod08] are an example of integrating such an approach with theorem proving support.) One of the advantages of such approaches is the inherent redundancy that increases the chances of early detection of design errors.

Closely allied to posit and prove approaches is the property of compositionality. In order for development to be conducted in an organised way, it should be possible to make, say, a design decomposition into sub-components and move on with confidence that everything that needs to be achieved is recorded in the specifications of the sub-components. Of course, mistaken design decisions might require backtracking because a specification is unsatisfiable but a component that meets its specification should never be rejected because of some unstated requirement. Compositionality is relatively easy to achieve with sequential programs but far more difficult in the presence of concurrency.

The issue of developing concurrent programs had not been tackled in the Vienna work; what was widely thought of as a viable approach to concurrent programs was the "Owicki/Gries method". Susan Owicki's thesis [Owi75] (or the more accessible [OG76]) sets out an approach in which the proof that two threads running concurrently satisfy some specification is tackled in two phases: in the first phase, each thread is separately developed to satisfy its own pre/post condition specification; the conjunction of these separate conditions must be such that they imply the required specification of the combined threads; but, before this result can be concluded, each thread must be proved not to interfere with the proof of the other thread. So, in the second of the two phases, one is asked to discharge a number of proof obligations that is the product of the number of statements in the two threads — but this is not the most worrying aspect of the Owicki/Gries approach. Far more serious is that the approach is fundamentally non-compositional in the sense that if this post-facto interference freedom (actually called by its authors the "einmischungsfrei") property is not true, the separate developments must be repeated. Owicki's contribution made progress beyond the earlier work of Ashcroft and Manna [AM71] but failed the test of being compositional in that the pre/post conditions of the two threads *fail* to express all of the requirements for acceptability. In [dR01], de Roever presents an encyclopedic analysis of compositional and non-compositional development methods for concurrency.

There is, in fact, a further limitation of the Owicki/Gries approach (shared with that of Ashcroft and Manna): there is a reliance on a fixed level of granularity. The chosen level happens to be that assignment statements (and expression evaluation) are assumed to be atomic. It is shown in Section 2.2 below how decisions on granularity can be put into the hands of the developer.

The basic idea behind rely/guarantee thinking is simple: interference must be acknowledged and provision made for reasoning about it. Just as few programs will function properly in a completely arbitrary starting state, almost no specification could be fulfilled by a program that experiences arbitrary interference. The familiar way of handling the former challenge is to record a pre condition that defines the set of starting states in which the program must terminate with a final state that is acceptable. Rely relations record the interference that a program must tolerate. It is important to note that both pre and rely conditions are effectively permissions to the designer of an implementation to ignore some deployment environments (*viz.* those that do not satisfy the conditions) they are not things to be tested in the program. Of course, a program is more robust if it satisfies weaker pre and rely conditions but there will always be some limitations to record.

The overall function of a terminating program is recorded (at least in VDM) as a relation between the initial and final states: the post condition is an obligation on the created implementation. The corresponding obligation that records limitations on the interference that a component may inflict on its environment is recorded in a guarantee relation. Figure 1 depicts these roles.

Most people record Hoare triples with the pre and post conditions in braces wrapped around the program constructs which are claimed to satisfy the specification — thus $\{p\}\ S\ \{q\}$. It is easy to extend these judgements to incorporate rely and guarantee conditions: $\{p, r\}\ S\ \{g, q\}$ has the two assumptions in the left braces and the two commitments on the right. For sequential programming constructs, inference rules are typically given in terms of Hoare-triple

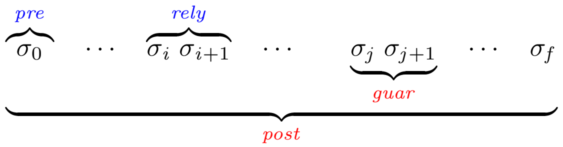

Figure 1: The roles of pre, rely, guarantee and post conditions

judgements. Using the 5-tuple judgements, rules for introduction of parallel constructs can be given — one possible rule is:

$$\boxed{\parallel\text{-}I}\ \frac{\begin{array}{c}\{P, R \vee G_2\}\ S_1\ \{G_1, Q_1\}\\ \{P, R \vee G_1\}\ S_2\ \{G_2, Q_2\}\end{array}}{\{P, R\}\ S_1 \parallel S_2\ \{G_1 \vee G_2, Q_1 \wedge Q_2 \wedge (R \vee G_1 \vee G_2)^*\}}$$

It should come as no surprise that this rule is more complicated than those for sequential constructs but it is actually easy to explain. If the overall combination of statements $S_1 \parallel S_2$ has to be able to achieve its post condition with interference (R) from its environment, then each S_i has to be able to tolerate that degree of interference plus any that can come from the sibling process S_j; the overall guarantee condition is the disjunction of the guarantees of the components; the overall post condition is at least as strong as the conjunction of the post conditions of the components but it is possible to add a conjunct that is the reflexive closure of the guarantees and the overall rely condition.

The simplest class of rely and/or guarantee conditions might state that the values of some variables remain unchanged but, in fact, such properties are better handled by some notation for "framing". The example in the next subsection illustrates conditions that express monotonic change. Interesting examples often combine conditions: Section 3.2 illustrates orderings on flags whose status, in turn, is used on the left of an implication which constrains changes to another variable.

Rely conditions discuss interference but do not fix the granularity of operations. This point is difficult to make clear without examples but, both in the sieve design of Section 2.2 and the more complicated ACM implementation discussed in Section 4.1, it should be clear that granularity can be fixed by the designer and is not predefined by the method.

The older references for R/G are [Jon81, Jon83a, Jon83b] but [Jon96] provides an adequate overview. The ugly soundness proof in [Jon81] has been replaced in [CJ07, Col08] — Leonor Prensa Nieto provided Isabelle-checked soundness proofs in [Pre03, Pre01] but the programming language is somewhat restricted (parallel statements cannot be nested) and there is a simplifying assumption on granularity. Among the numerous other theses on R/G, it is worth mentioning Ketil Stølen's because it [Stø90] tackles progress arguments. A different style of R/G rule in [CJ00] uses so-called "evolution invariants". Although now becoming slightly dated (in that many relevant theses post-date its publication), Willem-Paul de Roever's encyclopedic survey [dR01] offers an excellent reference point and carefully argues the distinction between compositional and non-compositional approaches to shared-variable concurrency.

One idea that is better *not* regarded as an extension of R/G is the use of auxiliary (or "ghost") variables; this point is expanded upon in Section 5.

2.2 A racy example

Section 1 above mentions Peter O'Hearn's dichotomy that uses the key distinction between race-free programs and those which are "racy". One example of developing a racy program is known as the "Sieve of Eratosthenes". The specification requires that all primes are identified up to some maximum value n; the algorithm attributed to the worthy Greek simply eliminates all composites by starting at two and progressively eliminating the products of each successive number (this process can terminate at $\lfloor\sqrt{n}\rfloor$); after sieving, only the primes remain. Data abstraction is useful to make the specification and top-level design clear: the set of possible primes is stored in a variable $s\colon \mathbb{N}$-**set**[2]. Initialisation arranges that s contains all natural numbers up to n. With an obvious outer loop, the post condition of the process ($\mathit{REM}(i)$) that removes multiples of i simply requires $s' = s - c_i$ where c_i is all of the multiples of i. It is then straightforward to see that, over the whole loop, all composites are removed and the primes remain.

The interest here is in developing a concurrent version of this sieving process. The design decision to run instances of *REM* concurrently can be described sensibly at the level of the $s\colon \mathbb{N}$-**set** data representation; but if $\mathit{REM}(i)$ is to run concurrently with $\mathit{REM}(j)$, its post condition cannot be the strict equality $s' = s - c_i$ because $\mathit{REM}(j)$ might have removed numbers. In [Jon83a] this example is used to initiate the reader to juggling tricks with the various conditions — a more systematic approach is described in the next sub-section. Suffice it here to say that using the post condition to set a lower bound on what is removed (i.e. $s' \cap c_i = \{\,\}$) has to be reflected in the guarantee condition by defining an upper bound on deletions ($s - s' \subseteq c_i$) and both rely and guarantee conditions end up needing $s' \subseteq s$ as a conjunct.

There are a number of interesting facets of this first level of design for this sieve example.

1. It is important to note that the granularity of the interference is much finer than that of the *REM* operations being specified: many elements could be removed from the set s by the environment of some instance $\mathit{REM}(i)$ during its execution.

2. The specification given has *not* fixed the level of granularity of interaction: a (rather poor) implementation could meet the specification by having each instance of *REM* lock the whole of set s for the duration of its execution. Of course far better implementations for, say, a many-core architecture will avoid this locking but the decision is left open by the

[2] VDM notation [Jon90] is used but should present no difficulty.

R/G description of this first design step; further steps in the design process need to make, record and justify the design decisions.

3. The rely and guarantee conditions are used to advantage on abstract types: they capture the natural intuition of monotonic removal of elements before the detailed representations are discussed.

4. Notwithstanding the previous point, data reification has an essential part to play in achieving the guarantee condition. Assuming the set is finally represented by some indexed vector, the guarantee condition can best be achieved if the atomicity at the vector level works per indexed element; locking would be required if, for example, the representation packed eight bits into a byte and the operations of the machine were at the byte level. (This intimate connection between R/G and data reification was noted, in [Jon07], some time after the initial R/G ideas were proposed.)

5. The code developed for this sieve example does exhibit real races on the final data representation (cf. the example in Section 4.1). The detailed code meets the guarantee condition because it is possible to have a "remove" primitive that is idempotent.

2.3 An algebraic presentation of R/G

Two recently funded research projects are aiming to develop R/G thinking: "Taming Concurrency" is funded by (UK) EPSRC and "Understanding concurrent programmes using rely/guarantee thinking" is led by Ian Hayes and funded by the Australian Research Council. The former project in particular has made an explicit aim to "pull apart" both R/G and SL with a view to understanding what they each express naturally. It is hoped that this understanding can lead to one or more new combinations of notations that work together well. This section indicates how "getting under the (syntactic) skin" of R/G could offer a way forward.

R/G takes the issue of interference head on and uses guarantee conditions to record the interference an implementation can inflict on its environment; correspondingly, rely conditions record the interference that an implementation must tolerate. The fixed format 5-tuple for presenting rely and guarantee conditions is abandoned in [HJC13] in favour of a "refinement calculus" [Mor90, Mor94] style of presentation which is extended to allow rely and/or guarantee statements to be added to either specifications or code.

As in the original refinement calculus, pre/post condition specifications are treated as commands and can be written $[p, q]$ — identically **true** pre conditions are elided as in $[q]$. Rely or guarantee conditions can be added to any command c as follows: $\mathbf{rely}\, r \cdot c$, $\mathbf{guar}\, r \cdot c$. The framing conventions from the refinement calculus are also adopted — this is discussed below in Section 3.3.

Using this notation, the laws relating the various command constructs express pleasing properties that were invisible in the original R/G presentation.

Enough laws are presented to revisit the sieve example.[3]

Three laws that express equalities over commands that involve guarantee commands are

Nested-G $(\mathbf{guar}\, g_1 \cdot (\mathbf{guar}\, g_2 \cdot c)) = (\mathbf{guar}\, g_1 \wedge g_2 \cdot c)$

Trading-G-Q $(\mathbf{guar}\, g \cdot [g^* \wedge q]) = (\mathbf{guar}\, g \cdot [q])$

Distribute-G- $||$ $\mathbf{guar}\, g \cdot (c \parallel d) = (\mathbf{guar}\, g \cdot c) \parallel (\mathbf{guar}\, g \cdot d)$

The first of these should be self-explanatory; that named *Trading-G-Q* reflects the fact that, since a guarantee command requires every atomic step to satisfy g, the overall execution preserves the transitive closure of that condition g^*. The third is one of a collection that permits distribution of guarantee commands over the different program constructs.

The next law is a weakening law in that the right hand side of $\sqsubseteq$ will suffice in any context where the left is acceptable.

Intro-G: $c \sqsubseteq (\mathbf{guar}\, g \cdot c)$

Notice however that the g constraint on the right makes it harder to implement that command than c alone.

Several laws are given in [HJC13] for the introduction of parallel constructs; these laws are closest in intent to the 5-tuple law in Section 2.1; the n-ary parallelism used in the sieve example is symmetric[4] and the following simple form suffices (*Intro-* $\parallel$ *-n* is again a weakening law):

Intro- $\parallel$ *-n*: $[q] \sqsubseteq \; \parallel_i (\mathbf{guar}\, gr \cdot (\mathbf{rely}\, gr \cdot [q_i]))$

providing $\forall i \cdot q_i \;\Rightarrow\; q$.

These laws are enough to develop an implementation of the concurrent version of prime sieving — see Figure 2. As above, set s initially contains all natural numbers up to some n; C is the set of all composite numbers; and

$$c_i = \{i * j \mid 2 \leq j \wedge (i * j) \leq n\}$$
$$C = \bigcup\{c_i \mid 2 \leq i \leq \lfloor \sqrt{n} \rfloor\}$$

The first step of the proof development is justified by set theory; *Intro-G* is used to require that no atomic step removes prime (non composite) numbers from s; given that this condition is transitive, *Trading-G-Q* can be used to drop the conjunct from the post condition because $s - s' \subseteq C$ expresses a transitive relation; employing *Intro-* $\parallel$ *-n* requires the insertion of the (matching rely and guarantee) condition on monotonic shrinking of the set s; the penultimate step uses *Distribute-G-* $||$; the final step moves to an equivalent specification with the nested guarantees combined.

The final line of Figure 2 is essentially the expected rely/guarantee specification for *REM*. The steps of development from there to the detailed code would not be dissimilar to those in [Jon81] but there are now laws for distributing rely and guarantee conditions over loop and sequence constructs (in the earlier version these were taken as "obvious") and proper laws for introducing assignments (which tended to be handled informally in R/G).

[3] The names used here differ from those for the laws in [HJC13] — the choice here is for shorter names that suffice for the current example.

[4] Such symmetric gr is an interesting special case but more interesting parallel decompositions such as that in Section 4.1 use different predicates for the rely and guarantee conditions.

$$
\begin{array}{ll}
 & [s' = s - C] \\
= & \text{by set theory} \\
 & [s' \cap C = \{\,\} \wedge s - s' \subseteq C] \\
\sqsubseteq & \text{by } \mathit{Intro\text{-}G} \\
 & \mathbf{guar}\, s - s' \subseteq C \cdot [s' \cap C = \{\,\} \wedge s - s' \subseteq C] \\
= & \text{by } \mathit{Trading\text{-}G\text{-}Q}(s - s' \subseteq C \text{ is transitive}) \\
 & \mathbf{guar}\, s - s' \subseteq C \cdot [s' \cap C = \{\,\}] \\
\sqsubseteq & \text{by } \mathit{Intro\text{-}} \parallel \mathit{\text{-}n} \\
 & \mathbf{guar}\, s - s' \subseteq C \cdot (\parallel_i \mathbf{guar}\, s' \subseteq s \cdot \mathbf{rely}\, s' \subseteq s \cdot [s' \cap c_i = \{\,\}]) \\
= & \mathit{Distribute\text{-}G\text{-}} \parallel \\
 & \mathbf{guar}\, s - s' \subseteq C \cdot \mathbf{guar}\, s' \subseteq s \cdot (\parallel_i \mathbf{rely}\, s' \subseteq s \cdot [s' \cap c_i = \{\,\}]) \\
= & \mathit{Nested\text{-}G} \\
 & \mathbf{guar}\, s - s' \subseteq C \wedge s' \subseteq s \cdot (\parallel_i \mathbf{rely}\, s' \subseteq s \cdot [s' \cap c_i = \{\,\}])
\end{array}
$$

Figure 2: An (extended) refinement calculus development of *Sieve*

The presentation in the refinement calculus style should not be taken as a step away from "posit and prove" developments. Small examples such as that for prime sieving are seductive but, when one is faced with an industrial post condition that is perhaps a page long, the beauty of a chain of one liners like those in Figure 2 is no longer an option. It should also be clear that laws which are not equalities (i.e. they use $\sqsubseteq$) normally require some design inspiration. Sieve is however a useful illustrative example and the new R/G laws do have an algebraic form hidden by the original 5-tuple presentation. The material in [HJC13] includes an operational semantics, a dozen or so lemmas proved directly from the semantics, over 50 laws derived from the lemmas and an example that is different from the one used here. Hopefully, the new presentation affords a clearer understanding of interference.

3 Reasoning about separation (race avoidance)

As stated above, the planned research programme will also try to "pull apart" separation logic to understand its fundamental contribution. Section 3.1 describes the *issue of separation* and sketches how SL helps reason about the issue; Section 3.2 follows O'Hearn's discussion in [O'H07] in which he moves from separation to "ownership" (Section 4 returns to this issue).

3.1 Concurrent Separation Logic

The issue of separation concerns clarifying which parts of the state are of relevance to different concurrent threads. When considered this broadly, separation can be seen as one way of ensuring (non-)interference. There are two dimensions in which a more focused analysis is needed. Firstly it is useful to look at read vs. write access and secondly the problem takes a different complexion depending on how elements of the state are identified. For the latter dimension, the term "stack variables" is used to refer to the normal identifiers declared in high-level programming languages whereas the phrase "heap variables" is used for access to store via natural number references.

Tony Hoare made a first attempt to extend the "axiomatic basis" [Hoa69] to parallelism in [Hoa72]. That paper considers programs using (normal) stack variables. Assuming separate pieces of code had been proved to satisfy specifications given in terms of their individual pre/post conditions, the question was under which conditions the parallel execution of the code segments would satisfy a specification formed by conjoining their pre/post conditions.[5] Since Hoare was concerned with programs using normal variables, requiring that the threads did not share variables was a simple check of the alphabets of the programs.

Notice that it is not only where two threads write to the same variable that data races can occur: statements proved to satisfy a specification with a pre condition that fixes the value of say x cannot conclude that x still has that value if a concurrent thread can write to x. Read/write conflicts also matter. For stack variables, the separation of alphabets is straightforward. For example, each operation in VDM [Jon90] identifies its **rd/wr** state components and this would support reasoning about separation in the case of normal variables. In fact, it could be argued that separation is just an extreme way of achieving non-interference and that R/G handles more delicate interference requirements.

Separation logic [Rey00, Rey02] tackles the messier case of reasoning about heap variables: where the portion of the state to be read and/or written is determined by a natural number, it is clear that checking separation is more complex.[6] Concurrent Separation Logic [O'H07] resolves several technical challenges in order to get back to a rule that is identical in intent to Hoare's approach in [Hoa72]. Suppose it is necessary (presumably in some larger piece of reasoning) to draw some conclusion about the concurrent execution of two statements that refer to the heap:

$$[x] \leftarrow 3 \parallel [y] \leftarrow 4$$

Little can be concluded if it is unknown whether the values in the stack variables x and y refer to the same address. If however it is a pre-condition that the addresses are distinct, it would be desirable to be able to prove a post condition of the combined statement that conjoins the two individual post conditions. A key SL proof rule permits exactly this reasoning but, rather than normal conjunction, "separating conjunction" (written $P * Q$) is only defined where P and Q are separate. The rule is:

$$\boxed{SL}\ \frac{\begin{array}{c}\{P_1\}\ s_1\ \{Q_1\}\\ \{P_2\}\ s_2\ \{Q_2\}\end{array}}{\{P_1 * P_2\}\ s_1 \mid\mid s_2\ \{Q_1 * Q_2\}}$$

Using $x \mapsto 3$ to mean that the element of the heap whose address is the value

[5] The seeds of [Owi75] and even [AM71] can be detected here.

[6] Using SL for stack variables (e.g. [PBC06]) is, in most cases, overly heavy.

of x holds the value 3 and $x \mapsto \_$ to mean that x holds some value, the above mini-challenge can be proved by an instance of the *SL* rule:

$$\begin{array}{l} \quad\{x \mapsto \_ * y \mapsto \_\} \\ [x] \leftarrow 3 \parallel [y] \leftarrow 4 \\ \quad\{x \mapsto 3 * y \mapsto 4\} \end{array}$$

In both the pre and post condition, the separating conjunction is crucial. One huge benefit of separating conjunction is that the frame rule gives a delightful way of embedding a component in a larger frame:

$$\boxed{\mathit{SL\text{-}frame}}\,\frac{\{P\}\ s\ \{Q\}}{\{P * R\}\ s\ \{Q * R\}}$$

Unsurprisingly, SL is extremely potent for reasoning about disjoint concurrency such as is used in parallel merge sort in [O'H07].

3.2 Ownership

Although the authors writing about separation logic always use that adjective, there is a sense in which it could more usefully be described as "ownership logic". Whether said authors agree or not, the issue of "ownership" is certainly one that has to be faced in many concurrent systems. In particular, there is an interesting class of concurrent system in which the ownership of some part of the shared state is passed between threads. SL appears to be well equipped to deal with such problems and [O'H07] uses the example of passing a value between writer and reader processes by passing its address (O'Hearn adds the important observation that this programming pattern is essential to achieve performance in low-level code).

Interestingly, transfer of ownership of stack variables can easily be specified using R/G. For example, a rely condition for a reader process might record that a buffer (b) does not change when the r flag is set together with the fact that the environment cannot set r to false:

$$(r \Rightarrow b' = b) \wedge (r \Rightarrow r')$$

the writer process must have a corresponding guarantee condition and might rely on the fact that its environment cannot make r true ($r' \Rightarrow r$).

Given that such ownership exchanges are needed for both stack and variables, it feels as though there ought be one way of expressing the idea in either case rather than asking users to employ R/G in the former case and SL in the latter.

It would, in fact, be possible to represent the heap as one component of an overall state and to code assertions about the heap being unchanged using the various map operators in, say, VDM. This is *not* the line proposed in this paper; the interest here is in teasing out the fundamental issues and finding natural ways of handling the issues.

Probing a little deeper into the issue of ownership, it is worth establishing exactly what is intended. Complete ownership of a variable might be taken to mean that only the owner has write or read access. In other situations, it might be useful to express finer distinctions. One of many extensions to SL concerns "fractional permissions" [Boy03] and these can be used to express ownership distinctions. Fractional permissions do, however, look like a way of "coding" something deeper. It is for example interesting to compare the use of fractional permissions in [dRPDYD$^+$11, §4.3] and abstract predicates (see Section 5.2) to tackle the sieve problem of Section 2.3.

Matt Parkinson's [Par10] has the title "The next 700 separation logics"[7] and is a hint of how versions of SL are proliferating to meet new challenges. One laudable property of nearly all SL extensions is the concern shown for algebraic properties of their operators (e.g. "magic wand"). Hopefully, the developments in Section 2.3 will help bring SL and R/G researchers even closer together.

3.3 Framing

The early papers on R/G used VDM's keyword style to define the **rd/wr** frames. The move to a refinement calculus presentation not only gives a more linear notation for assertions, it also prompts the use of a compact notation to specify the write frame of a command. Thus:

$$x\colon [Q]$$

requires that the relational post condition Q is achieved with changes only being made to the variable x. This makes a small step towards the compact notation of separation logic. Rather than go to the complete determination of frames from the alphabets of assertions used there, a sensible intermediate step might be to write pre and post conditions as predicates with explicit parameter lists and have the arguments of the former determine the read frame and the extra parameters of the latter determine the write frame. The indirection of having named predicates would pose little overhead in large applications because it is impractical to write specifications in a single line.

4 Abstraction as a Key Tool

As well as focusing on the issues around concurrency and what needs to be expressed in order to cope with them, this paper (and its predecessor [Jon12a]) presents the case that "abstraction" can be a key tool in tackling the issues. It is pointed out in Section 2.2 above that data abstraction and reification already play an important role in rely/guarantee methods. This is certainly not surprising: VDM has emphasised data abstraction in specification and reification in design since its inception (see [Jon03b, §3.2]) and [Jon80] was probably the first book to put equal emphasis on data in design and programming constructs. The current section goes further, both showing that abstraction appears to extend the domain of R/G (Section 4.1) and offering a new view on reasoning about separation (Section 4.2).

[7] Obviously echoing Peter Landin's [Lan66].

4.1 Abstract race avoidance

The data race that occurs in the sieve example (cf. Section 2.2) is real in the sense that multiple threads execute assignments to shared variables without explicit synchronisation. In other words, only the hardware memory synchronisation behaviour defines the granularity. That algorithms can be designed to work in such cases depends on some form of idempotence — in the sieve case, no harm is done setting a portion of storage to a null value multiple times. A far more subtle example is treated in [Jon81]: the application is the Fisher/Galler algorithm for recording equivalence relations (sometimes known as the "union/find" problem); a concurrent clean-up algorithm that compresses trees was designed in the expectation that some software locking would be required — the analysis using R/G showed that this can be avoided. The property on which this proof is built is far more subtle but, in some very general sense, can again be seen as an idempotent change.

In contrast, this section outlines a case where what appears to be a data race at an abstract level of design actually disappears in later design decisions giving rise to a race free implementation. The example is rather intricate and cannot be fully described here but enough can be sketched to convey the essential point and cited papers contain the supporting details.

The application is the implementation of so-called "Asynchronous Communications Mechanisms" (ACMs). Logically, these are just one place buffers with one writer and one reader but the difficulty derives from the adjective "asynchronous": neither reader nor writer can ever be delayed and, of course, the reader must never see incoherent data that is being changed. If the asynchronous property is to be achieved, it should be obvious that the logical idea of a single buffer cannot be realised by a single shared piece of store. A little more thought shows that two pieces of shared store are also inadequate. An ingenious "four slot" design is due to Hugo Simpson [Sim90]. A strength of his solution is that synchronisation between reader and writer depends only on two single bits (or control wires). ACMs are used in applications where sensors are writing into the buffer and control programs are extracting the values when required (the independence of the two processes giving rise to the asynchronous requirement). In such applications, the use of multiple slots must not be such that the reader ever sees "stale" values. In other words, a value being read must be at least as fresh as that from the most recent write that completed before the read commenced. In particular, it could be disastrous if the reader were ever able to read a value older than one that it had already seen.

There have been many attempts to offer both correctness arguments and, more usefully, understandable design explanations of Simpson's algorithm. Relevant publications that use R/G and/or SL include [JP08, BA10, JP11, BA11, WW10] and several interesting observations are made below on these attempts. First, [JP11, §3] is considered because it exhibits an "abstract race". The initial specification (of which, more anon) is given in terms of an abstract state (Σ^a) that, as well as some pointers, contains a sequence in which is recorded every value written. An intermediate state (Σ^i) is used to explain one set of design decisions: Σ^i retains, in general, far fewer values which are stored in a mapping whose domain is (for now) some unspecified index set X and whose range is the values being passed.[8] The writer and reader processes race on access to the mapping in the sense that both can make changes to the same map; it is precisely the role of the rely and guarantee conditions to record enough information to show that the same range element of the mapping is never read at the same time as it is being written; the values of auxiliary pointers are used to express these assertions. Of course, further conjuncts in the rely and guarantee expressions state which process can change which pointers and when they can do so.

The overall effect is that what look like races on the abstraction actually get removed in the final step of development. So, in this example, R/G is being used to reason about a program whose whole purpose is to avoid races! The argument for using R/G is that –at least for the layers of design abstraction chosen in [JP11]– races on the abstract objects appear to support a convenient abstraction. There are several further aspects of the development given in [JP11] that might be worth reviewing but the interest here is in raising the question whether O'Hearn's interesting dichotomy actually places R/G correctly. Perhaps it would be more accurate to say that R/G indeed supports reasoning about data races but that such races can be abstractions of race-free implementations.

There are, moreover, further interesting comparisons to be made between the collection of papers relating to Simpson's algorithm. One natural view of the 4-slot algorithm is that the ownership of the slots is passed between the writer and reader processes. Following the train of argument from separation to ownership in Sections 3.1/3.2, this would make it look to be perfect territory for SL. It is, therefore, informative to look at some of the relevant papers from authors who are associated with SL. Most clearly in [BA11], it is stated that "We don't use separation or ownership transfer, . . . ". In fact, this paper uses a logic that is a combination of R/G and SL known as "RGSep" [VP07, Vaf07][9] but concedes in the Acknowledgements "And finally we are grateful to the referees, in particular for forcing us to recognise that we weren't exploiting separation logic and should recast our proofs without it." This is not to claim that [BA11] uses only R/G; in fact, they also use "linearisability" (cf. Section 5) in a novel way. The only publication that appears to use SL to reason about ownership exchange in Simpson's algorithm is [WW10] but this unfortunately confines itself to coherence and stops short of proving the essential "freshness" property.

A thesis of the current paper is that one should be clear about the issues that need to be addressed in concurrency before apposite notations are chosen for their expression. The subsidiary thesis of this section is that the powerful tool of abstraction can help most approaches. Before turning in the next subsection to how this might be seen achieved with SL, a brief aside is made about a concept that appears to be useful and which does not appear to have a mode

[8] One useful bonus of this layering of design decisions is that one can show at this step that at least three slots are essential. In the final step of the explanatory design history, the set X is reified as the cross product of two Boolean values thus indexing Simpson's four slots.

[9] More is said about this in Section 5.

of expression in most approaches.

An interesting concept that needed expression in [JP11] is the ability, in assertions, to discuss the "possible values" that a variable can take. This actually came from spotting a flaw in an earlier version of our development of Simpson's 4-slot implementation: at some point in [JP08] there was a need to record in the post condition for a *Read* sub-operation that one of the pointer variables (*hold-r*) acquired the value from another variable (*fresh-w*) that could be set by a *Write* process. This was written in the earlier, flawed, version of the development by stating that either the initial or final value of *fresh-w* could be captured. But this is not actually general enough because the sibling (*Write*) process could be executed any number of times and make many assignments to its variable whilst the *Read* process was executing. This prompted the creation of a special notation in [JP11] for the set of values that can arise and the post condition of the *Read* process can be correctly recorded as $hold\text{-}r \in \widehat{fresh\text{-}w}$. The possible values notation is equally useful in, say, guarantee conditions and the full payoff comes in proofs.

An encouraging sign for the utility of the possible values notation ($\widehat{x}$) is that several other uses have been found for the same concept. Furthermore, a pleasing link with Ian Hayes' on-going research on non-deterministic expression evaluation is formalised in [HBDJ13].

Both [JP08] and [JP11] use a "phased specification" in which the *Read* and *Write* processes are each expressed as the sequential composition of two sub-operations. The overall system being expressed as the parallel combination of these two sequential compositions. Despite the fact that the authors claim that the use of "semicolon" as a specification operator offers a clear intuition of the freshness requirement in ACMs, it has been shown in [Jon12b] that the possible values notation can yield a specification without such "phased specifications"; the possible values notation gives exactly the required expressiveness.

4.2 Separation as abstraction

In view of the added value that abstraction gives to R/G approaches, it looks worth investigating how much benefit can be drawn by using that same powerful generic idea to tackle the issues where SL appears to be useful. The proposal here is more speculative than that outlined in Sections 2.3 and 4.1 but it does appear to point to a similar "pulling apart" of issues from notation.

In [Rey02], John Reynolds considers a sequential in-place list reversal algorithm. He actually introduces the problem (using the implementation!) as follows:

> The following program performs an in-place reversal of a list:
>
> $j := \textbf{nil};\ \textbf{while}\ i \neq \textbf{nil}\ \textbf{do}$
> $(k := [i+1]; [i+1] := j; j := i; i := k).$
>
> (Here the notation $[e]$ denotes the contents of the storage at address e.)

Reynolds' reasoning then employs "separating conjunction" as in

$$\exists \alpha, \beta \cdot \mathit{list}(\alpha, i) * \mathit{list}(\beta, j)$$

to derive the expected specification.

In contrast a top-down development might start with a post condition that only has to require that some variable, say r, is changed so that

$$r, s \colon [r' = \mathit{rev}(s)]$$

Notice (cf. Section 3.3) that this specification gives the designer the permission to overwrite the variable s. The obvious *rev* function is defined

$$\begin{aligned} &\mathit{rev} : X^* \to X^* \\ &\mathit{rev}(s) \quad \triangleq \quad \textbf{if } s = [\,] \textbf{ then } s \textbf{ else } \mathit{rev}(\textbf{tl}\, s) \curvearrowright [\textbf{hd}\, s] \textbf{ fi} \end{aligned}$$

The specification is satisfied by the following abstract program

```
$r \leftarrow [\,]$;
while $s \neq [\,]$ do
   $r, s \colon [r' = [\textbf{hd}\, s] \curvearrowright r \wedge s' = \textbf{tl}\, s]$
od
```

The argument that this satisfies uses the fact that the loop maintains: $r' \curvearrowright \mathit{rev}(s') = r \curvearrowright \mathit{rev}(s)$.

At this stage of design, s and r are assumed to be distinct variables. That they are separate is a useful and natural abstraction but, of course, fails to embody the clever part of Reynold's algorithm. The step from the simple abstract algorithm to the clever pointer reversal can now be viewed as a step of data reification. A design *decision* to choose a representation in which both variables are stored in the same vector must maintain the essential points of the abstraction of separation.

The requirement to maintain the abstraction of separation thus moves to a data reification step. It is yet to be worked out what form of "separation logic" best suits this view but it is hoped that is will again be a step towards combining the advantages of separation logic thinking with ideas from rely/guarantee and data abstraction/reification.

5 Other approaches

5.1 SLs meet R/G

Research on SL is extremely active and, perhaps more surprisingly because of its much earlier inception, R/G research also appears to be accelerating. One pioneering attempt to look at combining the two approaches is described in [VP07, Vaf07]. In his extremely clear thesis, Viktor Vafeiadis gives the following combined rule:

$$\boxed{RGSep}\ \frac{\begin{array}{c}\{P_l, R \cup G_r\}\ s_l\ \{G_l, Q_l\} \\ \{P_r, R \cup G_l\}\ s_r\ \{G_r, Q_r\}\end{array}}{\{P_l * P_r, R\}\ s_l \mid\mid s_r\ \{G_l \cup G_r, Q_l * Q_r\}}$$

(The brevity of this rule comes in part from the fact that it is presented in a composition (rather than a decomposition) style.)

Despite the current author's admiration for this valuable contribution, it must be said that the *RGSep* rule presents a rather "syntactic" combination of the two approaches; the "Taming Concurrency" project is aiming to combine more fundamental insights from SL and R/G.

In the same vein, Xinyu Feng's SAGL [FFS07] argues that SL can be viewed as a specialisation of "assume guarantee" methods for a class of programs. More recently, Xinyu Feng has proposed "local rely-guarantee reasoning" in [Fen09].

Another interesting contribution to R/G thinking from Matt Parkinson and his colleagues is the "Deny/Guarantee" idea in [DFPV09]. Jürgen Dingel has also considered in [Din00, Din02] a "refinement calculus" view of rely/guarantee thinking. However, unlike the approach sketched in Section 2.3 (and worked out in [HJC13]), Dingel does not separate the four conditions (P, R, G, Q). There are also technical details concerned with his avoidance of relational post conditions.

5.2 Another 700 SLs

If it was tempting to regard the "next 700" in the title of [Par10] as a joke, keeping track of the many developments around SL is becoming a full-time task and the comments here are only intended to mention those items that might be candidates for consideration in bridging between SL and R/G. The research on "(concurrent) abstract predicates" [DYDG$^+$10] sounds as though it might be in the same groove as the case being made for abstraction in Section 4 above. In fact, the relationship is certainly more subtle with CAPs being used to handle subtle ownership questions that there is no obvious way of capturing with R/G.

The recent research on "Views" [DYBG$^+$13] offers a generic way of establishing the soundness of logics but the way in which the concurrency structure is created from the base semantics of atomic constructs would not handle situations as general as in, say, [CJ07]. It would however be worth pursuing the direction of general properties for soundness since undertaking such proofs on each logic is time consuming.

5.3 Ghost variables

An approach that is often adopted to extend the natural scope of a notation is to employ "ghost" or "auxiliary" variables. General words of warning about this escape route are offered in [Jon10] and need not be repeated here. The reservations can be summarised with an anecdote. Early in the history of R/G, a respected colleague claimed to have a "completeness proof" for the then current rely/guarantee rule. As has been made clear in this paper, the current author has never made such a claim. In fact, it was always clear that rely and guarantee conditions were expressively weak (even if it is only now that the move is made to view such weakness as a positive attribute). The resolution of the disagreement revolved around the use of auxiliary variables. Their addition can certainly make it possible to express properties that are not stateable with relations over the shared state. Unfortunately it is easy to see that the use of auxiliary variables can lose the key property of compositionality: auxiliary or ghost variables can be employed to record arbitrary amounts of information about a process — but relying on that information means that the implementation of that process is severely constrained. Sacrificing compositionality is far too high a price to pay for the cheap thrill of extending a notation to cover issues for which it was not intended. Whilst not being able to prove that auxiliary variables can always be avoided, [Jon10] sets out the case for finding sound reasons for their use.

5.4 Actions/Events

Employing "Actions" [BS91] or "Events" [Abr10] can offer an extremely neat framework for modelling systems. In [HA10], the authors seek to extend "Event-B" to mimic rely/guarantee style reasoning. It is possible to add environment events whose post conditions record interfering actions but it is equally clear that this can only mirror what really goes on in the rely/guarantee approach by making sure that all events (or actions) are at the granularity of the interference. In R/G reasoning itself, this is certainly not the case: post conditions express the overall effect of an "operation" (cf. event) but the granularity of the interference can be much finer.

5.5 RGITL

The combination of Ben Moszkowski's "Interval Temporal Logic" (ITL)[Mos85] with R/G in Gerhard Schellhorn's RGITL [STER11] provides a seductive combination. On the one hand, temporal logic offers a way of arguing about progress conditions and even various notions of fairness. In keeping with the concern of the current paper with expressiveness however it might be the case that RGITL –or even raw ITL– is too expressive. The fact that a user can write specifications in a language that can express complete programs may be dangerous because, in the hands of the unskilled, it moves the task of proving specification satisfaction to that of program equivalence.

5.6 Linearisability

The discussion above of [BA11] touches on linearisability and another of the impressive aspects of [Vaf07] it that it addresses this way of reasoning about interleaving. Research on linearisability was put on a firm foundation by [HW90];

further recent interesting papers include [GY11, BGMY12]. The basic idea is to look at detailed sub-steps and to find a larger atomic operation that would have the same effect.

It can be argued that the normal presentations of this idea are "bottom-up": they look at the code and try to find a linearised version. In keeping with the emphasis here on abstraction, it might be preferable to approach interleaving "top down" from a specification of acceptable behaviours. Earlier work on trying to do just this throws light clearly on the observational power of programming languages. The idea that it is possible, in a top-down design process, to use a "fiction of atomicity" is discussed in [Jon03a, Jon07] (for the origins of the ideas see references in these papers). The development process that links the abstraction to its realisation is known as "atomicity refinement" (or "splitting (software) atoms safely"). In one particular version of this process, equivalences were found that justified the introduction of concurrency primitives. What was crucial to the justification of these equivalences (see, for example, [San99]) was a careful analysis of the language in which observations can be made. (To make the point most simply, if the observation language can observe timings, parallel processes are likely to be seen as running faster; but there are much subtler dependencies to be taken into account as well.)

It must again be worthwhile to look at how these top-down and bottom-up views of varying the level of atomicity of processes can benefit from each other. Furthermore, both the basic idea of separate sets of addresses and of rely/guarantee-like assumptions about the effect of the processes look likely to be important when reasoning about the different granularities.

6 Conclusions

The current author has a number of prejudices whose exposure might make these conclusions clearer. Although the case is clear for doing something with the huge store of "legacy code" on which all users indirectly depend, the real payoff for formal methods is in the design process. Trying to prove that a finished program has properties such a deadlock freedom might make sense but deriving its full post condition would, in general, be impossible even for "correct" programs and is completely futile with programs that contain errors.

Related to the preceding point, so-called "partial correctness" is inadequate: if a program is intended to terminate, that fact must be part of its specification.

Also related to the argument for the use of formalism in the design process is the view that abstractions are best discovered in a "top down" view. Complexity can only be mastered with abstraction; clever tools might be able to detect abstractions "bottom up" from code; but, as a careful reading of [Cou08] shows, useful abstractions have to be discovered top-down.

The case for "posit and prove" methods is also strong in that they permit engineering intuition to be checked by the discharge of proof obligations. The inherent redundancy of such methods leads to productive use.

Referring back to the title of the current paper, the main argument here is not to regard restrictions on expressiveness as signs of weakness: well-judged restrictions on the expressive power of notations give might focus on the issues that can be handled naturally and increase the tractability of reasoning with said notations. The converse argument is that it is not necessarily an advantage to employ a notation that is more expressive: it might just result in intractability — especially in untutored hands. A particular plea has been made to abstain from using auxiliary variables as a cavalier way of extending the power of notations. This sin appears to be committed most commonly when authors try to extend a notation beyond the issues that it handles naturally. Experience suggests that "abstraction" is not only a key intellectual tool but that its judicious use can sometimes specifically avoid the need for ghost variables (the material on "when abstraction fails" in [Jon12a, §3.2] is relevant here).

Much remains to be done to arrive at notations that express naturally the key issues in concurrency but it is a corollary of the plea to find "natural" notations that researchers should be explicit about the *issues* that are being tackled. One issue not discussed in this paper is that of "progress" arguments. Other than [Stø90], little work has been done on such reasoning in the R/G framework; [Mid93] allows the use of temporal logic but reservations about being too general are covered in Section 5.5; an interesting limitation of the form of temporal assertion needed is given in [GCPV09]. Another limitation of R/G is identified in John Wickerson's thesis [Wic13]: he makes the point that compositionality does not ensure that a method (specifically R/G) can handle modular development.

Moving forward, this author's goal will *not* be to make arbitrary extensions to existing R/G notation but rather to understand the issue itself and then look for apposite notations — almost certainly guided by "abstraction".

There remains work to be done on the new presentation of R/G but there is far more to be done to take the initial steps in Section 4.2 to a full analysis of the issues of separation and ownership. Given the role of abstraction in these tentative steps on SL and the proven part that abstraction plays in R/G, a more general theory of abstraction needs investigation.

Two short-term objectives are the analysis of more examples (particularly those from SL) and the provision of machine support for the ideas in [HJC13].

Acknowledgements

The invitation to speak at SEFM gave rise to [Jon12a]; little of that paper remains here but the trip to Thessaloniki and the hospitality received remain warm memories. A subsequent invitation to ICECCS was also made enjoyable by my thoughtful and generous hosts; scientifically, it provided the opportunity to try out the ideas on expressiveness presented here. A discussion with Viktor Vafeiadis at ITP about his [Vaf13] reminded me that I had used the "union find" problem as a concurrency challenge in [Jon81]. Similarly, a discussion with Ralph-Johan Back at the 2013 WG 2.3 meeting in Saint Petersburg prompted the addition of Section 5.4.

It is also a pleasure to thank other research collaborators: Ian Hayes and Rob Colvin (particularly Sect. 2.3); Matt Parkinson and Richard Bornat (particularly Sect. 2.2); Hongseok Yang and Alexey Gotsman (particularly Sect. 5.6); all attendees at the productive series of concurrency meetings held in London, Cambridge, Newcastle, Dublin and Oxford. My colleagues Leo Freitas and Diego Machado Dias both provided detailed comments on an earlier draft of the current paper.

Matt Parkinson's August visit to give a seminar (based on [DYBG^{+}13]) in Newcastle was timely and prompted clarification of a number of remarks about SL — as well as providing other stimulating discussions.

The author of this paper gratefully acknowledges the funding for his research from the EPSRC Platform Grant TrAmS-2, the EPSRC responsive mode grant on "Taming Concurrency" and the ARC grant on "Understanding concurrent programmes using rely/guarantee thinking".

References

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

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

[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.

[ANS76] ANSI. Programming language PL/I. Technical Report X3.53-1976, American National Standard, 1976.

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

[BA11] Richard Bornat and Hasan Amjad. Explanation of two non-blocking shared-variable communication algorithms. *Formal Aspects of Computing*, pages 1–39, 2011.

[BGMY12] Sebastian Burckhardt, Alexey Gotsman, Madanlal Musuvathi, and Hongseok Yang. Concurrent library correctness on the TSO memory model. In *ESOP*, 2012.

[Boy03] John Boyland. Checking interference with fractional permissions. In Radhia Cousot, editor, *Static Analysis*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.

[BS91] Ralph-Johan Back and Kaisa Sere. Stepwise refinement of action systems. *Structured Programming*, 12:17–30, 1991.

[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.

[Cou08] Patrick Cousot. The verification grand challenge and abstract interpretation. In Bertrand Meyer and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments*, volume 4171 of *Lecture Notes in Computer Science*, pages 189–201. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-69149-5_21.

[DFPV09] Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. In Giuseppe Castagna, editor, *Programming Languages and Systems*, volume 5502 of *Lecture Notes in Computer Science*, pages 363–377. Springer Berlin / Heidelberg, 2009.

[Din00] Jürgen Dingel. *Systematic Parallel Programming.* PhD thesis, Carnegie Mellon University, 2000. CMU-CS-99-172.

[Din02] J. Dingel. A refinement calculus for shared-variable parallel and distributed programming. *Formal Aspects of Computing*, 14:123–197, 2002.

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

[dRPDYD+11] P. da Rocha Pinto, T. Dinsdale-Young, M. Dodds, P. Gardner, and M. Wheelhouse. A simple abstraction for complex concurrent indexes. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 845–864. ACM, 2011.

[DYBG+13] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. Views: compositional reasoning for concurrent programs. In *Proceedings of the 40th* *annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 287–300. ACM, 2013.

[DYDG$^+$10] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *Proceedings of the 24th European conference on Object-oriented programming*, pages 504–528, Berlin, Heidelberg, 2010. Springer-Verlag.

[Fen09] Xinyu Feng. Local rely-guarantee reasoning. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 315–327, New York, NY, USA, 2009. ACM.

[FFS07] Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP: Programming Languages and Systems*, pages 173–188. Springer, 2007.

[GCPV09] Alexey Gotsman, Byron Cook, Matthew Parkinson, and Viktor Vafeiadis. Proving that non-blocking algorithms don't block. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 16–28, New York, NY, USA, 2009. ACM.

[GY11] Alexey Gotsman and Hongseok Yang. Liveness-preserving atomicity abstraction. In *ICALP*, 2011.

[HA10] Thai Son Hoang and Jean-Raymond Abrial. Event-B decomposition for parallel programs. In Marc Frappier, Uwe Glaesser, Khurshid Sarfraz, Regine Laleau, and Steve Reeves, editors, *ABZ*, volume 5977 of *LNCS*, pages 319–333. Springer, 2010.

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

[HJC13] Ian J. Hayes, Cliff B. Jones, and Robert J. Colvin. Refining rely-guarantee thinking. *Transactions on Programming Languages and Systems*, (submitted), 2013.

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

[Hoa72] C.A.R. Hoare. Towards a theory of parallel programming. In *Operating System Techniques*, pages 61–71. Academic Press, 1972.

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

[Jon80] C. B. Jones. *Software Development: A Rigorous Approach.* Prentice Hall International, Englewood Cliffs, N.J., USA, 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.

[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.

[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.

[Jon03a] C. B. Jones. Wanted: a compositional approach to concurrency. In Annabelle McIver and Carroll Morgan, editors, *Programming Methodology*, pages 5–15. Springer Verlag, 2003.

[Jon03b] 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*, 375(1–3):109–119, 2007.

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

[Jon12a] Cliff B. Jones. Abstraction as a unifying link for formal approaches to concurrency. In G. Eleftherakis, M. Hinchey, and M. Holcombe, editors, *Software Engineering and Formal Methods*, volume 7504 of *Lecture Notes in Computer Science*, pages 1–15, October 2012.

[Jon12b] Cliff B. Jones. A specification for ACMs. Technical Report CS-TR-1360, Newcastle University, November 2012.

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

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

[Lan66] P. J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3):157–166, March 1966.

[Mid93] Cornelius A. Middelburg. *Logic and Specification: Extending VDM-SL for advanced formal specification.* Chapman and Hall, 1993.

[Mor90] Carroll Morgan. *Programming from Specifications.* Prentice-Hall, 1990.

[Mor94] C. C. Morgan. *Programming from Specifications.* Prentice Hall, second edition, 1994.

[Mos85] Ben Moszkowski. Executing temporal logic programs. In Stephen D. Brookes, Andrew William Roscoe, and Glynn Winskel, editors, *Seminar on Concurrency*, volume 197 of *LNCS*, pages 111–130. Springer Berlin Heidelberg, 1985.

[OG76] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.

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

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

[Par10] Matthew Parkinson. The next 700 separation logics. In Gary Leavens, Peter O'Hearn, and Sriram Rajamani, editors, *Verified Software: Theories, Tools, Experiments*, volume 6217 of *Lecture Notes in Computer Science*, pages 169–182. Springer Berlin / Heidelberg, 2010.

[PBC06] M. Parkinson, R. Bornat, and C. Calcagno. Variables as resource in Hoare logics. In *Logic in Computer Science, 2006 21st Annual IEEE Symposium on*, pages 137–146, 2006.

[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.

[Pre03] Leonor Prensa Nieto. The rely-guarantee method in Isabelle/HOL. In *Proceedings of ESOP 2003*, volume 2618 of *LNCS*. Springer-Verlag, 2003.

[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.

[Rod08] Rodin. Event-B and the Rodin Platform, 2008. `www.event-b.org`.

[San99] Davide Sangiorgi. Typed π-calculus at work: a correctness proof of Jones's parallelisation transformation on concurrent objects. *Theory and Practice of Object Systems*, 5(1):25–34, 1999.

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

[STER11] G. Schellhorn, B. Tofan, G. Ernst, and W. Reif. Interleaved programs and rely-guarantee reasoning with ITL. In *Temporal Representation and Reasoning (TIME), 2011 Eighteenth International Symposium on*, pages 99–106, 2011.

[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.

[Vaf13] Viktor Vafeiadis. Adjustable references. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *ITP 2013*, volume 7998 of *LNCS*. Springer, 2013.

[VP07] Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In Luís Caires and Vasco Vasconcelos, editors, *CONCUR 2007 – Concurrency Theory*, volume 4703 of *LNCS*, pages 256–271. Springer, 2007.

[Wic13] John Wickerson. *Concurrent verification for sequential programs.* PhD thesis, Cambridge, 2013.

[WW10] S. Wang and X. Wang. Proving Simpson's four-slot algorithm using ownership transfer, 2010. VERIFY Workshop, Edinburgh.