

# COMPUTING SCIENCE

Abstraction as a unifying link for formal approaches to concurrency

Cliff B. Jones

**TECHNICAL REPORT SERIES**

---

No. CS-TR-1339

June 2012

## **Abstraction as a unifying link for formal approaches to concurrency**

**C.B. Jones**

### **Abstract**

Abstraction is a crucial tool in specifying and justifying developments of systems. This observation is recognised in many different methods for developing sequential software; it also applies to some approaches to the formal development of concurrent systems although there its use is perhaps less uniform. The rely/guarantee approach to formal design has, for example, been shown to be capable of recording the design of complex concurrent software in a "top down" stepwise process that proceeds from abstract specification to code. In contrast, separation logics were -at least initially- motivated by reasoning about details of extant code. Such approaches can be thought of as "bottom up". The same "top down/bottom up" distinction can be applied to "atomicity refinement" and "linearisability". Some useful mixes of these approaches already exist and they are neither to be viewed as competitive approaches nor are they irrevocably confined by the broad categorisation. This paper reports on recent developments and presents the case for how careful use of abstractions can make it easier to marry the respective advantages of different approaches to reasoning about concurrency.

## Bibliographical details

JONES, C.B.

Abstraction as a unifying link for formal approaches to concurrency  
[By] Cliff B. Jones

Newcastle upon Tyne: Newcastle University: Computing Science, 2012.

(Newcastle University, Computing Science, Technical Report Series, No. CS-TR-1339)

### Added entries

NEWCASTLE UNIVERSITY

Computing Science. Technical Report Series. CS-TR-1339

### Abstract

Abstraction is a crucial tool in specifying and justifying developments of systems. This observation is recognised in many different methods for developing sequential software; it also applies to some approaches to the formal development of concurrent systems although there its use is perhaps less uniform. The rely/guarantee approach to formal design has, for example, been shown to be capable of recording the design of complex concurrent software in a "top down" stepwise process that proceeds from abstract specification to code. In contrast, separation logics were -at least initially- motivated by reasoning about details of extant code. Such approaches can be thought of as "bottom up". The same "top down/bottom up" distinction can be applied to "atomicity refinement" and "linearisability". Some useful mixes of these approaches already exist and they are neither to be viewed as competitive approaches nor are they irrevocably confined by the broad categorisation. This paper reports on recent developments and presents the case for how careful use of abstractions can make it easier to marry the respective advantages of different approaches to reasoning about concurrency.

### 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. 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 also PI on an EPSRC-funded project "AI4FM" and coordinates the "Methodology" strand of the EU-funded DEPLOY project. He 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) since 1973 (and was Chair from 1987-96).

### Suggested keywords

CONCURRENCY  
FORMAL METHODS  
RELY/GUARANTEE THINKING  
SEPARATION LOGIC  
ATOMICITY REFINEMENT  
LINEARISABILITY

# Abstraction as a unifying link for formal approaches to concurrency

Cliff B. Jones

School of Computing Science, Newcastle University, NE1 7RU, UK  
cliff.jones@ncl.ac.uk

**Abstract.** Abstraction is a crucial tool in specifying and justifying developments of systems. This observation is recognised in many different methods for developing sequential software; it also applies to some approaches to the formal development of concurrent systems although there its use is perhaps less uniform. The rely/guarantee approach to formal design has, for example, been shown to be capable of recording the design of complex concurrent software in a “top down” stepwise process that proceeds from abstract specification to code. In contrast, separation logics were –at least initially– motivated by reasoning about details of extant code. Such approaches can be thought of as “bottom up”. The same “top down/bottom up” distinction can be applied to “atomicity refinement” and “linearisability”. Some useful mixes of these approaches already exist and they are neither to be viewed as competitive approaches nor are they irrevocably confined by the broad categorisation. This paper reports on recent developments and presents the case for how careful use of abstractions can make it easier to marry the respective advantages of different approaches to reasoning about concurrency.

## 1 Introduction

There is much research activity around formal support for concurrency. The reasons for this ought be clear. For non-critical applications, good (semi-formal) engineering methods are sometimes adequate for sequential programs. Such methods borrow much from past formal research and, even here, organisations such as Praxis report that adding formal methods to the development process can bring about a return on investment because of the tighter control and reduction in the late discovery of errors that are expensive to fix because they result from decisions made much earlier in the design process.

Once one moves to the design of concurrent systems, the enormous increase in the number of execution paths brought about by thread interaction makes it effectively impossible to have any confidence in correctness without some form of formal proof.

One might ask why designers should be so rash as to venture into such dangerous territory. Unfortunately, there is no choice — the pressures to face concurrency become ever greater. First, the (economic) limits for the extrapolation of “Moore’s law” mean that hardware performance can only be increased by moving from “multi-core” to “many-core” hardware (i.e. numbers of threads likely to measured in hundreds). Secondly, embedded systems often run in parallel with physical phenomena that are varying continuously; control software linked to the physical world by sensors and actuators

cannot ignore these state changes. Thirdly, a class of application has to be implemented by physically distributed sets of processors.

The combination of a realisation that concurrency cannot be avoided with the acknowledgement that its mastery requires formal tools has generated many research strands. Notable activity in the areas of rely/guarantee thinking, separation logic, atomicity refinement and linearisability is addressed in the body of this paper (citations to relevant papers are given below). To apply some of these research ideas to the paper itself, the attempt here is to look for constructive interaction between several threads of research. In particular, this paper looks to tease out the key *concepts* from the various methods and indicate a path to one or more methods that achieve real synergy from what are currently rather distinct approaches. This is a much deeper exercise than just seeking combinations of notations.

A key distinction between top-down and bottom-up approaches is used below. Any such dichotomy must be viewed with care and there is certainly no intention to make a judgement that one approach is “better” than the other. If the task is to improve the quality of millions of lines of legacy code, there is little choice but to use bottom-up methods. On the other hand, faced with the challenge of developing and documenting a large system from scratch, it would be unwise not to record each stage of design “from the top” (i.e. the specification).<sup>1</sup> One particular form of top-down formal development that makes solid engineering sense is known as “posit and prove”. The idea is that each step of design starts by recording an engineering intuition that might be a decomposition of a problem into sub-tasks or the choice of a data representation for something that was previously an abstraction that achieved brevity. In suitable formal methods, such a posited step gives rise to “proof obligations” whose discharge justifies the correctness of the step. It is well understood that redundancy is essential for dependability and this posit and prove approach provides constructive redundancy. As discussed below, there is a technical requirement of “compositionality” for such methods. Although relatively easy to achieve for sequential systems, compositionality is far more elusive in the world of concurrent systems.

It is however important to remember that no judgement is being made here about the relative merits of top-down and bottom-up approaches. There is indeed evidence that, in (complex) bottom-up analysis, it is necessary to recreate abstractions that are hidden in the code [Cou08,LS09]. It is hoped, and anticipated, that any new methods devised from –for example– the combination of concepts from separation logic and rely/guarantee thinking will provide benefit to both top-down and bottom-up approaches.

## 1.1 Rely/guarantee thinking

Specifications of sequential programs are normally given as pre and post conditions.<sup>2</sup> Floyd showed [Flo67] how predicate calculus assertions could be added to a flowchart

---

<sup>1</sup> Of course, there exists the issue of how to obtain the starting specification. This is not the subject of the current paper but some contribution to a resolution of this issue is made in [JHJ07].

Interestingly, this joint work with Ian Hayes and Michael Jackson uses rely/guarantee ideas.

<sup>2</sup> This paper does not waste space making the case for formality in system design but it is worth remembering that formal concepts (e.g. data type invariants) are useful even when used in specifications and developments that are not completely formal.

of a program to present a proof that the program satisfied a specification; Hoare made the essential step [Hoa69] to give an inference system for asserted texts. Few programs can tolerate completely arbitrary starting states and it is important to note that pre conditions effectively grant a developer assumptions about the starting states in which the created software will be deployed; in contrast, post conditions are requirements on the running code.

The essence of concurrency is *interference*. In shared variable concurrency, such interference manifests itself by one thread having to tolerate changes being made to its state by other processes.<sup>3</sup> No useful program can achieve a sensible outcome in the presence of completely arbitrary interference. This is recognised in the rely/guarantee approach [Jon81, Jon83a, Jon83b] by recording the acceptable interference as a relation—known as a rely condition—over pairs of states. (The use of relations fits with the fact that VDM [Jon80] employs relational post conditions.) Like pre conditions, rely conditions can be seen as permission for the developer to make assumptions about contexts in which the final code will be deployed. The commitment as to what interference a running component will impose on its neighbours is recorded in a guarantee condition (again a relation over states).

Not surprisingly, the proof obligations required in rely/guarantee reasoning are more complex than for sequential programs. There is also scope for more variability and the proof obligations concerned with introducing concurrency differ over various publications. One form (geared to decomposition — see Sect. 2.1) is:

$$\begin{array}{c}
 \{P, R_l\} s_l \{G_l, Q_l\} \\
 \{P, R_r\} s_r \{G_r, Q_r\} \\
 R \vee G_r \Rightarrow R_l \\
 R \vee G_l \Rightarrow R_r \\
 G_l \vee G_r \Rightarrow G
 \end{array}
 \quad
 \frac{\overline{P} \wedge Q_l \wedge Q_r \wedge (R \vee G_l \vee G_r)^* \Rightarrow Q}{\text{Par-I} \quad \{P, R\} s_l \parallel s_r \{G, Q\}}$$

What is crucial is that rely/guarantee systems of rules can be made “compositional” in the same sense that Hoare-like methods for sequential programs enjoy this essential property: a specification with rely/guarantee conditions records all that a developer need know to create acceptable code. De Roever’s exhaustive survey book [dR01] distinguishes between compositional and non-compositional development methods pointing out that the *post-facto* “Einmischungsfrei” proof obligation in the Owicki/Gries method makes it non-compositional.

Examples of rely/guarantee development are postponed to Section 2 where future issues are explored. The most approachable text on past research in this area might be [Jon96]; a proof of the soundness of rely/guarantee methods is given in [CJ07].

Compositionality is key to methods that are to be used in a *top-down* style where a development is started from an overall specification and decomposed step by step

<sup>3</sup> Process algebras might appear to finesse the whole issue of “states” but processes can be constructed that effectively store values that can be changed and read by interaction; the issue of interference reappears as reasoning about the traces of such interaction.

until the finest sub-components have been developed into code. Data abstraction is key to achieving brief and understandable specifications; the corresponding development method of *data reification* is also compositional. Interaction between data reification and rely/guarantee thinking is explored in Section 3.1 below.

## 1.2 Separation logic(s)

Just as in the preceding section, it is neither the aim to offer a complete description nor to present a full history of research on separation logics<sup>4</sup> and the extension to concurrent separation logic here. Some important milestones include [Bur72, OP99, ORY01, Rey00, IO01, Rey02, O’H07, Bro07, OYR09]. In fact, Peter O’Hearn pointed out at the Cambridge meeting to mark Tony Hoare’s 75<sup>th</sup> birthday that the fundamental idea of disjoint parallelism dates back to [Hoa72]. Of interest for the current paper is the notion of “separating conjunction” and the emphasis on reasoning about heap variables.<sup>5</sup>

To show that execution threads do not “race” on access to a particular variable, it is enough to establish that there is mutual exclusion between any reference from those threads to the relevant variable. With standard (“stack”) variables, this can be achieved by ensuring that each variable is visible to at most one thread. Separation logic is, however, more often applied to programs using dynamic (“heap”) variables and it offers ways of reasoning about the dynamic “ownership” of their addresses. The principal tool is the “separating conjunction”: if two conjuncts have disjoint frames, they can be associated with different threads of execution. Thus, two parallel processes can achieve a post condition written as a separating conjunction by each achieving one of the conjuncts. There is no interference and no “race” on addresses.

One important aspect of separation logic is that ownership can change dynamically; this does however appear to be the reason that the “frame” of an operation is determined by the alphabet of its assertions. Perhaps more significant for the objectives of the current paper is that there are—in addition to separating conjunction—a number of other operators in separation logic and that all of the operators are linked by useful algebraic laws. A practical point is that the researchers involved with separation logic have put considerable effort into providing tool support for their ideas.

The majority of papers on separation logic focus on “heap” variables<sup>6</sup> that can be dynamically allocated and freed. The examples chosen are typically of low-level (operating system like) code performing tasks like maintaining concurrent queues and delicate manipulations of tree representations. This creates the impression that separation logic is aimed at “bottom up” analysis of extant code. Furthermore, much of the commendable effort on tool support is aimed at establishing freedom from stated faults such as race conditions in extant code.

As an example (that is useful in Section 2) Reynolds considers a sequential in place list reversal in [Rey02]; the introduction of the problem is:

---

<sup>4</sup> The justification for using the plural of “logic” is [Par10].

<sup>5</sup> In [PB05] a move to high level programming constructs is made — that paper does not, however, link the constructs to concurrency.

<sup>6</sup> In [PBC06], the ideas of separation logic are applied to “stack” variables but the resulting system appears less pleasing than that for heap variables.

The following program performs an in-place reversal of a list:

```
j := nil; while i ≠ nil do
(k := [i + 1]; [i + 1] := j; j := i; i := k).
```

(Here the notation  $[e]$  denotes the contents of the storage at address  $e$ .)

The reasoning then employs “separating conjunction” ( $*$ ) as in

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

to specify that the lists starting respectively at addresses  $i$  and  $j$  encompass separate sets of addresses. The extremely succinct separation logic rule for a parallel construct is

$$\boxed{\text{SL}} \frac{\begin{array}{c} \{P_l\} s_l \{Q_l\} \\ \{P_r\} s_r \{Q_r\} \end{array}}{\{P_l * P_r\} s_l || s_r \{Q_l * Q_r\}}$$

There is a lot going on in this compact rule. The separating conjunction (written as an infix “ $*$ ” operator) is only valid if the frames of the two disjuncts are disjoint. Moreover, since there is no explicit declaration of the read/write frames of either operand, these are determined by the alphabets of the expressions.<sup>7</sup> It is also the norm that the *SL* rule is used on heap variables (i.e. machine addresses that are allocated at run time rather than names of variables that are translated to machine addresses by a compiler).

Before considering the potential for using the core ideas of such ownership logics early in the design process, the next section reviews what has already been done to obtain complementary benefits from the two approaches outlined.

### 1.3 Existing complementarity

The research around rely/guarantee thinking and separation logics is extremely active. Both [VP07] and Viktor Vafeiadis’ Cambridge thesis [Vaf07] propose a combination of rely/guarantee and separation thinking: the “RGSep” rules neatly specialise to either of the original sets of rules.<sup>8</sup>

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

(The brevity of this rule is slightly artificial: see discussion in Sect. 2.1.)

Matt Parkinson’s “Deny/Guarantee” system [DFPV09] extends rely/guarantee ideas to cope with the dynamic forking of threads. Although it belongs to a later discussion, it is also worth mentioning here the “RGSim” approach [LFF12].

<sup>7</sup> This has the surprising consequence that some expression  $E$  and  $E \wedge x = x$  do not have an equivalent effect.

<sup>8</sup> Xinyu Feng’s research on SAGL [Fen09] goes in a similar direction to RGSep — for reasons of space, discussion here is confined to the latter.

An interesting trace of interaction between the two main approaches discussed so far can be seen in a series of papers that all address “Asynchronous Communication Methods” in general — and more specifically Hugo Simpson’s “4-slot” algorithm [Sim90]. Richard Bornat is a key figure in Separation Logic research but both of his papers [BA10, BA11] make use of rely/guarantee descriptions — a fact that is explicit even in the title of the earlier contribution. The current author’s contributions [JP08, JP11] interleave in time with Bornat’s and throw an interesting light on a distinction that has been drawn between methods. The talk by Peter O’Hearn at the 2005 MFPS (published as [O’H07]) suggested that the natural tool for proving the absence of data races is separation logic — in contrast, rely/guarantee reasoning is appropriate for “racy” programs. One key feature of Simpson’s 4-slot algorithm is the avoidance of clashes (or data races) on any of the four slots used as an interface between the entirely asynchronous read/write processes. However, [JP11] uses rely/guarantee conditions to describe the non-interference at an abstract level — in fact, this is done before the number of slots has been determined.

The MFPS categorisation *can be* useful but it is important to remember that any such split must be used judiciously. (This warning applies, of course, also to the use of “top down” versus “bottom up” in the current paper.)

## 2 Seeking further synergy

It is clear that neither rely/guarantee nor separation logic alone can cope with all forms of concurrency reasoning. This is precisely the reason that looking for synergy is worthwhile. Lacunae on the rely/guarantee side certainly include the ability to reason about (dynamic) ownership and discourses about heap variables. (The extent to which the ideas of Sect. 2.4 below can finesse these gaps is yet to be determined.)

From the other side, there are concepts with which separation logic appears to struggle because interference can be problematic without races. Consider the innocent looking thread, say  $\alpha$ :

$$x \leftarrow 1; y \leftarrow x$$

Suppose that some thread control variables are added so that parallel processes do not interfere during the execution of either assignment in thread  $\alpha$ . The classical Hoare rule would carry the information that  $x$  has the value 1 to the precondition of the second assignment. If there is a concurrent process that increases the value of  $x$  by an arbitrary amount (e.g. a loop that continues to execute  $x \leftarrow x + 1$ ), then this transfer of “knowledge” is certainly invalid. There are no data races here but, if one is to conclude  $y \geq 1$ , there have to be assumptions (rely conditions) about the context in which thread  $\alpha$  will run.

This section reports on some on-going research and speculates about some further directions where abstraction might make it possible to get underneath the syntactic details of both rely/guarantee and separation logic; the hope is that, by really understanding the conceptual contribution, one or more methods will evolve that are intuitive to the intended users.

## 2.1 Algebraic laws about rely/guarantee

It has never been the intention to fix on one set of proof rules for rely/guarantee reasoning. One cause of differences between rules is the distinction between rules that are convenient for composition versus those best suited to decomposition. This distinction can be illustrated even on the standard Hoare-rule for **while** constructs: the most common form of the rule gives the post-condition of a **while** as the conjunction of the loop invariant (say  $P$ ) with the negation of the test condition written in the loop construct — thus  $P \wedge \neg b$ . This is a useful composition rule but it is unlikely that, when faced with a design step, the post condition will fall neatly into such a conjunction. An equivalent Hoare rule with an arbitrary post condition of  $Q$  to be achieved after the loop requires an additional hypothesis that  $P \wedge \neg b \Rightarrow Q$ .<sup>9</sup> For pre/post conditions, this distinction is small and often glossed over in texts but for more complex rely/guarantee specifications, the difference in presentation between composition and decomposition rules is greater — as can be seen by contrasting the rules in Sects. 1.1/1.3. Of course, the rules are related by suitable “weakening rules” but the choice of an appropriate form of the rule does matter when providing tool support for proof obligation generation.

There are also more interesting differences between versions of rules that generate proof obligations for rely/guarantee reasoning. In [CJ00], for example, an “evolution invariant” is used that can be thought of as relating any state that can arise back to the initial state. Just as (standard, single state) data type invariants have proved useful intuitive aids in developments that are not necessarily formal, the idea of evolution invariants has sparked interest in a number of areas.

These points prompted a desire to find something more basic that could be used to reason about interference in a rely/guarantee style. In [HJC12], inspiration is drawn from the refinement calculus as presented by Carroll Morgan [Mor94] (and, in particular the “invariant command” of [MV94]) to employ **guar** and **rely** constructs written as commands. The move away from fixed format presentation of rely/guarantee as in

$$\{P, R\} s \{G, Q\}$$

brings advantages similar to those of the refinement calculus over Hoare-triples.<sup>10</sup> In addition, it makes clear that there is an algebra of the clauses. For example, the trading of clauses between guarantee and post conditions that appears almost as black magic in earlier papers becomes a law

$$(\mathbf{guar} G \bullet [Q \wedge G^*]) = (\mathbf{guar} G \bullet [Q])$$

Furthermore, the collection of laws in [HJC12] fits with a pleasing refinement calculus top-down development style (the reader is referred to that paper for the full set of rules and a worked example — the style of formal semantics used there is that of [HC12]).

<sup>9</sup> VDM uses relational post conditions that make it possible to express termination (sometimes referred to as “total correctness”) via well-founded relations — this feels more natural than the “variant function” of [Dij76]. Be that as it may, the same distinction between composition and decomposition presentations remains.

<sup>10</sup> Jürgen Dingel has also looked at presenting rely/guarantee ideas in a form of refinement calculus setting — his objective in [Din00, Din02] was not however to separate the commands in the way done in [HJC12].

## 2.2 Framing

The early papers on rely/guarantee reasoning 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, it also prompts the use of a compact notation to specify the write frame of a command. Thus:

$$x: [Q]$$

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 rarely practical to write specifications in a single line.

## 2.3 Possible values

Another interesting development in [JP11] is the usefulness of being able, in assertions, to discuss the “possible values” that a variable can take. This idea actually arose from a flaw in an earlier version of our development of Simpson’s four-slot implementation of *Asynchronous Communication Mechanisms*: at some point there was a need to record in a post condition for a *Read* (sub-)operation that the variable *hold-r* acquired the value from a variable *fresh-w* that could be set by a *Write* process. This was written in the earlier, flawed, version of the development [JP08] as  $hold-r = \overline{fresh-w} \vee hold-r = fresh-w$ . But allowing that *hold-r* acquired the initial or final values of *fresh-w* is not enough because the sibling (*Write*) process could execute many assignments to *fresh-w* whilst the *Read* process was executing. This prompted a special notation for the set of values that can arise and the post condition of the *Read* process can be correctly recorded as  $hold-r \in \overline{fresh-w}$ . 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 many 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 [HBDJ11].

## 2.4 Separation as an abstraction

Thus far, several ways in which abstraction can facilitate both cleaner developments and, more generally, useful concepts for developing programs have been shown. In this more speculative section, a way of viewing the core concept of separation *as an abstraction* is explored.

Returning to Reynolds’ example of reversing a sequence, a top-down development might start with a post-condition built around the function:

$$\begin{aligned}
rev &: X^* \rightarrow X^* \\
rev(s) &\triangleq \mathbf{if } s = [] \mathbf{ then } s \mathbf{ else } rev(\mathbf{tl } s) \overset{\curvearrowright}{\mathbf{hd } s}
\end{aligned}$$

The post condition itself only has to require that some variable, say  $s$ , is changed so that

$$r, s: [r = rev(\overleftarrow{s})]$$

(Notice that this specification gives the designer the permission to overwrite the variable  $s$ .)

This can be achieved by the following abstract program:

```

r ← [];
while s ≠ [] do
  r, s: [r = hd  $\overleftarrow{s}$ ]  $\overset{\curvearrowright}{r} \wedge s = \mathbf{tl } \overleftarrow{s}$ 
  {r  $\overset{\curvearrowright}{rev}(s) = rev(\overleftarrow{s})$ }
od

```

Thus far,  $s$  and  $r$  are assumed to be distinct variables. That they are separate is a useful and natural abstraction. A design *decision* to choose a representation in which both variables are stored in the same vector must maintain the essential points of that abstraction! 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 it will again be a step towards combining the advantages of separation logic thinking with ideas from rely/guarantee and data reification. (Sect. 3.1 describes existing links between rely/guarantee reasoning and data reification.)

## 2.5 Fiction of atomicity as an abstraction for linearisability

There is insufficient space here to go into a complete exploration of a further pair of approaches but it is worth mentioning that there are other issues that fit the analysis of two ideas that have evolved from top-down and bottom-up views and look ripe for reconsideration.

Research on linearisability was put on a firm foundation by [HW90]; recent interesting papers include [GY11, BGM12]. The basic idea is to look at detailed sub-steps and to find a larger atomic operation that would have the same effect.

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 enhanced concurrency. 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 more subtle 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.

### 3 Further observations on abstraction

Much is being made about the virtues of “abstraction” in this paper. It is useful to look both at some past successes and issues around the use of this panacea in software design. Section 2 starts out with a confession that the ideas in that section are speculative; it is likely that “issues” will arise in their development and that past experience might be useful in their resolution.

#### 3.1 Rely/guarantee thinking and data reification

It is difficult to exaggerate the importance of data abstraction in specifying computer systems. Whilst it is true that there are cases like sorting where a post condition is much easier to write than an algorithm, most complex computing tasks can only be described in a brief and understandable way if their description is couched in terms of abstract objects that match the problem in hand. Of course, mathematical abstractions<sup>11</sup> are not necessarily available in programming languages. The top-down, stepwise, process of “reifying” abstractions to data types that are available in implementation languages is a key tool of formal methods (one of the first books to emphasise this is [Jon80] and it has been given its due prominence in VDM ever since).

Because the current author had seen this importance, it was completely natural that—even in the earliest rely/guarantee developments— data abstraction and reification were deployed. What was less apparent was the strength of the link between the ideas. In fact, it was not until [Jon07] that full recognition was given to the extent to which the ability to find a representation affected whether or not granularity assumptions could be met without locking. Having noticed this—on a range of examples— it can now be used to help guide the choice of rely and guarantee predicates.

#### 3.2 What happens when abstraction fails?

It is illuminating to sketch the history of data abstraction/reification in VDM.<sup>12</sup> Peter Lucas’ first proof of the equivalence of two distinct (VDL) operational semantic formulations in fact used a “twin machine” idea that amounted to a relation between the two models. It was only later that the research on VDM focused on the use of a “retrieve” function that was in effect a homomorphism from the representation back to the abstraction. This idea became the standard in VDM (e.g. [Jon80]) and there was even a test devised for “implementation bias”. Everything was rosy in the abstraction garden until

---

<sup>11</sup> It is interesting to remember that all of the formal specification notations VDM [Jon80], Z [Hay93] and B [Abr96] employ the same collection of abstract objects: sets, sequences, maps and some form of record.

<sup>12</sup> This is done more fully, and in the context of other research, in [Jon03b].

a few contrary examples appeared where, in each, it was impossible to find an unbiased state that covered all allowable implementations. The essence of the problem was that, for these rare applications, the abstract state had to contain information that allowed non-determinacy; but once design decisions removed the non-determinacy, the states could be simplified in a way that meant they had *less* information than the abstraction; this meant that no homomorphism could be found.

The problem of justifying such design decisions was overcome by adding a new data reification rule to VDM that derives from the research of Tobias Nipkow [Nip86] (a parallel development in Oxford led to [HHH<sup>+</sup>87]). The essential point here is that one should strive for “bias freedom” but that it might not always be attainable. If this is genuinely the case, there may be a need to devise new proof methods.

One particular “trick” that is often used in reasoning about concurrent programs is to add “auxiliary” (or “ghost”) variables that record information that is not (readily) available in the actual variables of a program. The temptation to do this is often strong but this author has doubts about the wisdom of giving in to it. The danger is that it is difficult to put precise limits on what it is legitimate to record in ghost variables. Compositionality can be completely destroyed by recording information about a thread that one might wish to revise without changing the design of any concurrent threads. More is said on this topic in [Jon10].

## 4 Conclusions and next steps

Clearly, much remains to be done to bring about intellectual and software tools that will contribute to the work of software design for concurrent systems.

In some senses, what marks out a useful formal method is not its ability to express *anything* but rather its expressive weakness! One seeks a notation that can cover a useful class of applications but be weak enough to be tractable. For some applications rely/guarantee conditions –coupled with liberal amounts of abstraction– fit this pattern. The basic rely/guarantee relations make it possible to go through a top-down development of non-trivial algorithms that allow (at least abstract) races on variable access. Ketil Stølen showed in [Stø90] that the same basic framework can be extended to handle progress arguments. Similarly, separation logic approaches employ a notation for which useful tool support has been developed. What one has to seek is a sweet spot where much can be handled with a (close to) minimum of formal overhead.

Although the current author finds compact notations attractive (and remember the point made by Christopher Strachey that it is far easier to manipulate a string of symbols that fit on a line than multi-line texts), it is unavoidable that specifications of large systems get recorded using long formulae. This author has written many papers extolling the advantages of abstraction but has seen enough formal specifications of systems such as “cruise control” to avoid setting “single line specifications” as an objective. What is important is to have notations whose operators are linked with useful algebraic properties: separation logic clearly achieves this and, for rely/guarantee, [HJC12] makes a first step in this direction but the objective must be kept in mind.

Another area where separation logic researchers have been wise is in their emphasis on tool support for their ideas. This must –and will– be an objective of our research to bring together different approaches to concurrency reasoning.

It is sadly the case that most currently available programming languages are poor vehicles for expressing (safe) concurrency. There is, therefore, a temptation to plan to embody the ideas from the research adumbrated in the body of this paper into yet-another programming language. Such is not the immediate objective of the current author who has seen too many languages that offer at most one new idea but implement many other concepts less well than existing languages. The first step is tractable design concepts (that might be used to develop programs into patterns in existing languages); it would be pleasing if these patterns were adopted by some careful language designer(s).

## Acknowledgements

It is a pleasure to thank many research collaborators. The most relevant and active contacts on the research reported here are Ian Hayes and Rob Colvin (particularly Sect. 2.1); Matt Parkinson, Viktor Vafeiadis and Richard Bornat (particularly Sect. 1.3); Hongseok Yang and Alexey Gotsman (particularly Sect. 2.5); and all attendees at the productive series of concurrency meetings held in London, Cambridge, Newcastle and Dublin (and Oxford in July is eagerly anticipated).

The author of this paper gratefully acknowledges the funding for his research from the EPSRC Platform Grant TrAmS-2.

## References

- [Abr96] J.-R. Abrial. *The B-Book: Assigning programs to meanings*. Cambridge University Press, 1996.
- [BA10] Richard Bornat and Hasan Amjad. Inter-process buffers in separation logic with rely-guarantee, 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.
- [Bro07] S. D. Brookes. A semantics of concurrent separation logic. *Theoretical Computer Science*, 375(1-3):227–270, 2007.
- [Bur72] R.M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.
- [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.
- [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.
- [Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J., USA, 1976.
- [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 Non-compositional Methods*. Cambridge University Press, 2001.
- [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.
- [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.
- [GY11] Alexey Gotsman and Hongseok Yang. Liveness-preserving atomicity abstraction. In *ICALP*, 2011.
- [Hay93] Ian Hayes, editor. *Specification Case Studies*. Prentice Hall International, Englewood Cliffs, N.J., USA, second edition, 1993.
- [HBDJ11] Ian J. Hayes, Alan Burns, Brijesh Dongol, and Cliff B. Jones. Comparing models of nondeterministic expression evaluation. Technical Report CS-TR-1273, School of Computing Science, University of Newcastle, September 2011. Submitted to Computer Journal, visible on-line at [www.cs.ncl.ac.uk/research/pubs/trs/papers/1273.pdf](http://www.cs.ncl.ac.uk/research/pubs/trs/papers/1273.pdf).
- [HC12] Ian J. Hayes and Robert J. Colvin. Integrated operational semantics: Small-step, big-step and multi-step. In J. Derrick et al., editors, *ABZ*, volume 7316 of *LNCS*, pages 21–35. Springer Verlag, 2012.
- [HHH<sup>+</sup>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.
- [HJC12] I. J. Hayes, C. B. Jones, and R. J. Colvin. Refining rely-guarantee thinking. Technical Report CS-TR-1334, Newcastle University, May 2012. submitted to Formal Aspects of Computing visible on-line at [www.cs.ncl.ac.uk/research/pubs/trs/papers/1334.pdf](http://www.cs.ncl.ac.uk/research/pubs/trs/papers/1334.pdf).
- [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.
- [IO01] S. Isthiaq and P. W. O’Hearn. BI as an assertion language for mutable data structures. In *28th POPL*, pages 36–49, 2001.
- [JHJ07] Cliff B. Jones, Ian J. Hayes, and Michael A. Jackson. Deriving specifications for systems that are connected to the physical world. In Cliff B. Jones, Zhiming Liu, and Jim Woodcock, editors, *Formal Methods and Hybrid Real-Time Systems*, volume 4700 of *Lecture Notes in Computer Science*, pages 364–390. Springer Verlag, 2007.
- [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.
- [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 reasoning 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.
- [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.
- [LFF12] Hongjin Liang, Xinyu Feng, and Ming Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 455–468, New York, NY, USA, 2012. ACM.
- [LS09] Dirk Leinenbach and Thomas Santen. Verifying the microsoft hyper-v hypervisor with vcc. In Ana Cavalcanti and Dennis Dams, editors, *FM 2009: Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*, pages 806–809. Springer Berlin / Heidelberg, 2009.
- [Mor94] C. C. Morgan. *Programming from Specifications*. Prentice Hall, second edition, 1994.
- [MV94] C. C. Morgan and T. N. Vickers. Types and invariants in the refinement calculus. In C. C. Morgan and T. N. Vickers, editors, *On the Refinement Calculus*, pages 127–154. Springer Verlag, 1994.
- [Nip86] T. Nipkow. Non-deterministic data types: Models and implementations. *Acta Informatica*, 22:629–661, 1986.
- [O'H07] P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1-3):271–307, May 2007.
- [OP99] P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, June 99.
- [ORY01] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *15th CSL*, pp1–19, 2001.
- [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.
- [Par10] Matthew Parkinson. The next 700 separation logics. In Gary Leavens, Peter O'Hearn, and Sriram Rajamani, editors, *Verified Software: Theories, Tools, Experiments*, vol-

- ume 6217 of *Lecture Notes in Computer Science*, pages 169–182. Springer Berlin / Heidelberg, 2010.
- [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.
- [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.
- [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 Reynolds. A logic for shared mutable data structures. In Gordon Plotkin, editor, *Proceedings of the Seventeenth Annual IEEE Symp. on Logic in Computer Science, LICS 2002*. IEEE Computer Society Press, July 2002.
- [San99] Davide Sangiorgi. Typed  $\pi$ -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.
- [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.
- [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 *Lecture Notes in Computer Science*, pages 256–271. Springer Berlin / Heidelberg, 2007.