DOI 10.1007/s00165-014-0310-2
The Author(s) © 2014. This article is published with open access at Springerlink.com
Formal Aspects of Computing (2015) 27: 475–497

Formal Aspects of Computing

Balancing expressiveness in formal approaches to concurrency

Cliff B. Jones[1], Ian J. Hayes[2] and Robert J. Colvin[2]

[1] School of Computing Science, Newcastle University, Newcastle Upon Tyne NE1 7RU, UK
[2] School of Information Technology and Electrical Engineering, The University of Queensland, Brisbane 4072, Australia

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 ubiquitous tool and its influence on how the key issues are tackled is examined in each case.

Keywords: Concurrency, Rely/guarantee reasoning, Separation logic

1. Introduction

Concurrency has been an issue in computing for a long time but finding tractable ways of reasoning about 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: without using formal methods, developing sequential programs that satisfy their specifications is difficult, in the presence of concurrency it is virtually impossible. 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 Sect. 2.1) is harder to achieve. The central concern of this paper is "expressiveness" and this certainly becomes a more delicate balancing challenge with concurrency.

1.1. Suitably expressive abstractions

Finding suitable abstractions both to specify a problem and to derive a program to meet that specification is a fundamental challenge in computing science. Good notations are needed both to specify problems and their solutions, and tractable methods must be found for showing that the solution satisfies the specification. Providing the right balance of expressiveness in such notations is crucial. The better understood realm of specifying and reasoning about sequential programs is discussed first to highlight the important issues, before addressing these issues for concurrent programs.

Dedication: Wlad Turski 1938–2013

Correspondence and offprint requests to: C. B. Jones, E-mail: cliff.jones@newcastle.ac.uk

As an example where restricting the notation is seen as beneficial, consider the debate over structured programming versus programming with **goto**s [Dij68, DDH72]. Programming with **goto**s allows one to build arbitrary (spaghetti) control-flow structures, whereas structured programming constructs restrict a language to a set of structured commands, such as **while** and **case** (or **switch**) statements, which can make programs easier to comprehend. An important aspect of structured statements is that their behaviour is defined in terms of the behaviour of their sub-components.

There is a link from the structured programming debate to program verification. Floyd's original method of assigning meanings to programs [Flo67] was based on arbitrary control-flow graphs, while Hoare's contribution [Hoa69] adapts Floyd's ideas to structured programs. Hoare's inference rules take advantage of the use of structured statements: the proof of a property of a structured statement decomposes into a set of proofs about its component sub-statements. That allows what has become known as *compositional* reasoning, which has become a goal of any proof method for programs.

In Hoare's approach a sub-component statement, no matter how complex, is abstracted to a pre and post condition pair that it must satisfy. Hoare's pre and post conditions facilitate the abstract specification of sequential programs. Compared to program code they are both more restrictive, in that they only define properties of the initial and final states of a program (and say nothing about any of its intermediate states), and more expressive, in that program code does not normally allow expression of assumed preconditions and can use concepts that are absent from the programming language. Furthermore, postconditions allow non-deterministic specifications, whereas (sequential) program code is deterministic. The widespread use of Hoare logic indicates that it represents a good balance between expressiveness and tractability.

Hoare's abstraction of programs as pre and post conditions is sufficient to handle disjoint parallelism [Hoa72] but cannot handle the interference inherent in shared-variable parallelism precisely because it cannot say anything about intermediate states. A key challenge is to devise more expressive formalisms that allow compositional reasoning about shared-variable concurrency. As with Hoare logic, the balance is in devising suitable abstractions that can act as specifications of the components of a concurrent program. The rely-guarantee approach is one such method [Jon81, Jon83a]. It augments Hoare's pre and post conditions with rely and guarantee conditions that allow the specification of assumed properties about the interference that can be relied on and a guarantee that limits the interference the program can impose on its environment. But rely-guarantee is expressively weak in that, for example, it is not rich enough to express progress requirements.

Hoare's parallel rule [Hoa72] relied on partitioning the program variables into disjoint sets used by the component processes. Concurrent separation logic relaxes Hoare's constraint by using separating conjunctions in both the pre and post conditions; that allows the heap to be dynamically partitioned.

Vafeiadis [Vaf07, VP07] combines both the above by using rely/guarantee to bound the interference from concurrent processes and separation logic to handle sharing of the heap.

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 bend them artificially 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.

It is worth taking one further example where limitations on expressiveness can be seen as positive. Data abstraction is almost a *leitmotiv* of the current paper—it plays important roles in particular in Sects. 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 semantics but was unsure that abstractions like sets would be accepted by practitioners; consequently the committee decided to use sequences in the state 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 Sect. 1 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" (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. To address a potential complaint, the current authors' backgrounds lie in the R/G camp and the current paper is in no way claimed to be a completely impartial comparison of SL and R/G approaches.

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. O'Hearn [O'H07] proposes a useful dichotomy around "data races" 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 Sects. 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 be key to refining O'Hearn's dichotomy (see both Sects. 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 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, $x \leftarrow x + 1 \parallel x \leftarrow x \times 2$ 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 numeric 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 Sect. 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 summarises how R/G was originally formulated; after a motivating example in Sects. 2.2 and 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] were 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 use of a "posit and prove" development style; a strong commitment to data abstraction/reification; and judging any development method against the test of "compositionality".

VDM [Jon90], B [Abr96] and Event-B [Abr10] can be all 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. In contrast, one line arguments in the refinement calculus [BvW98, Mor94] work extremely well for small examples but industrial specifications are often long and a keyword style of splitting the parts of a specification is useful.

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

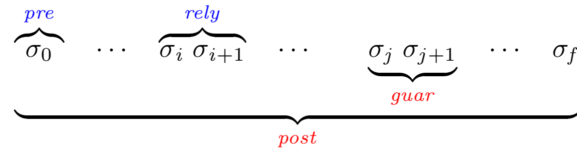

Fig. 1. The roles of pre, rely, guarantee and post conditions

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 VDM research; what was widely thought of as a viable approach to concurrent programs was the "Owicki/Gries method". Owicki's [Owi75] thesis (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, 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 does not hold, the separate developments must be repeated. Owicki's contribution moved 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. de Roever [dR01] 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 Sect. 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. Figure 1 shows σ_0 as a starting state and indicates that those of interest are restricted by the pre condition. The overall function of a terminating program is recorded in a post condition that is (at least in VDM) a relation between the initial and final states (σ_0/σ_f in Fig. 1). The corresponding obligation that records limitations on the interference that a component may inflict on its environment is recorded in a guarantee relation: in Fig. 1, this is shown as the component making a transition from σ_j to σ_{j+1}. Just as a pre condition can be thought of as permission for the designer to make assumptions about the starting environment of the component to be designed, a rely condition invites the assumption that a step made by the environment will satisfy the rely relation (shown as the environment making a step from σ_i to σ_{i+1} in Fig. 1).

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), and are not conditions 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 assumptions to record.

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

$$\boxed{\parallel\text{-}I}\ \frac{\{p, r \vee g_2\}\ s_1\ \{g_1, q_1\} \quad \{p, r \vee g_1\}\ s_2\ \{g_2, q_2\}}{\{p, r\}\ s_1 \parallel s_2\ \{g_1 \vee g_2, q_1 \wedge q_2 \wedge (r \vee g_1 \vee g_2)^*\}}$$

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.

Simple 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" (this topic is resumed in Sect. 3.3). The example in Sect. 2.2 illustrates conditions that express monotonic change. Interesting examples often combine conditions: Sect. 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 Sect. 2.2 and the more complicated asynchronous communication method (ACM) implementation discussed in Sect. 4.1, it should be clear that granularity can be fixed by the designer and is not predefined by the method. Here again, data abstraction and the use of rely conditions on abstract objects is important.

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]—Prensa Nieto [Pre03, Pre01] provided Isabelle-checked soundness proofs 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 Stølen's [Stø90] because it 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), de Roever's [dR01] exhaustive survey 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 Sect. 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[2]: the set of possible primes is stored in a variable s: $\mathbb{N}$-**set**. Initialisation arranges that s contains all natural numbers from 2 up to n. With an obvious outer loop, the post condition of the sequential process $REM(i)$ 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 set data representation; but if $REM(i)$ is to run concurrently with $REM(j)$, its post condition cannot contain the strict equality $s' = s - c_i$ because a concurrent $REM(j)$ might have removed numbers. Suffice it here to say that the post condition sets a lower bound on what is removed, while the guarantee condition defines an upper bound on deletions, and both rely and guarantee conditions end up needing a conjunct to ensure no composite values are added back into the set.

[2] VDM notation [Jon90] is used but should present no difficulty. In contrast to VDM's usage of $\overleftarrow{s}/s$ for old/new values in relations, here s/s' are used.

In [Jon83a] this example is used to introduce juggling tricks with the parts of the specification—a more systematic approach is described in Sect. 2.3. Furthermore, this pattern of weakening a post condition from a sequential specification and re-capturing some of the constraints in a guarantee condition is generally useful.

There are a number of interesting facets of this first level of design for the 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 $REM(i)$ 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 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 Sect. 4.1). The detailed code meets the guarantee condition because it is possible to have a "remove" primitive that is idempotent, i.e. repeated removal of the same element has the same effect as a single removal.

Although this example nicely illustrates the use of rely and guarantee conditions in the development of a concurrent design, it is in no way a claim that the approach can handle all such designs. The form of interference assertions chosen has inherent expressive restrictions. This observation leaves open the questions of whether the chosen expensiveness covers a useful class of design problems and whether useful extensions of "rely/guarantee thinking" can be found that do not reduce the tractability of the approach.

2.3. An algebraic presentation of R/G

The current authors are engaged in two projects ("Taming Concurrency" in the UK and "Understanding concurrent programmes using rely/guarantee thinking" in Australia) that relate to the research in this paper. The former project in particular has made an explicit aim to get underneath the specific R/G and SL notations with a view to understanding what they each express naturally. It is hoped that this understanding can lead to 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 [HJC14] in favour of a "refinement calculus" [BvW98, Mor94, Mor87] 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 conditions are treated by the command $\{p\}$, that aborts if p does not hold but otherwise terminates immediately, and post condition specifications by the command $[q]$, that allows any behaviour that satisfies q between its initial and final states but aborts if the environment performs any step that modifies the state. Rely or guarantee conditions can be added to any command c as follows: (**rely** $r \bullet c$) and (**guar** $r \bullet c$).

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

[3] The names used here differ from those for the laws in [HJC14]—the choice here is for shorter names that suffice for the current example.

$$\begin{array}{ll} \mathit{Nested\text{-}g}: & (\mathbf{guar}\ g_1 \bullet (\mathbf{guar}\ g_2 \bullet c)) = (\mathbf{guar}\ g_1 \wedge g_2 \bullet c) \\ \mathit{Trading\text{-}g\text{-}q}: & (\mathbf{guar}\ g \bullet \lceil g^* \wedge q \rceil) = (\mathbf{guar}\ g \bullet \lceil q \rceil) \\ \mathit{Distribute\text{-}g\text{-}parallel}: & \mathbf{guar}\ g \bullet (\|_{i \in S}\ c_i) = \|_{i \in S}\ (\mathbf{guar}\ g \bullet c_i) \end{array}$$

The first of these should be self-explanatory; law $\mathit{Trading\text{-}g\text{-}q}$ 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 allows a guarantee to be distributed over a set of commands in parallel (indexed by the finite set S); it is one of a collection of laws that permits distribution of guarantee commands over the different program constructs.

The next law is a refinement rather than an equality in that the right hand side of $\sqsubseteq$ will suffice in any context where the left is acceptable.

$$\mathit{Intro\text{-}g}: \quad c \sqsubseteq (\mathbf{guar}\ g \bullet c)$$

Notice however that the guarantee constraint makes it harder (than c alone) to implement the right-hand side of the ordering.

Several laws are given in [HJC14] for the introduction of parallel constructs—these laws are closest in intent to the 5-tuple law in Sect. 2.1. The multi-way parallelism used in the sieve example has symmetric rely and guarantee conditions[4] and the following simple form, in which S is a finite set, suffices (this is again a refinement):

$$\mathit{Intro\text{-}multi\text{-}parallel}: \quad \lceil \forall i \in S \cdot q_i \rceil \sqsubseteq \|_{i \in S}\ (\mathbf{guar}\ gr \bullet (\mathbf{rely}\ gr \bullet \lceil q_i \rceil))$$

These laws are enough[5] to illustrate the approach for a concurrent version of prime sieving—see Fig. 2. With C as the set of all relevant composite numbers:

$$\begin{aligned} c_i &= \{i \times j \mid 2 \le j \wedge (i \times j) \le n\} \\ \mathit{range} &= \{i \in \mathbb{N} \mid 2 \le i \le \lfloor \sqrt{n} \rfloor\} \\ C &= \textstyle\bigcup_{i \in \mathit{range}} c_i \end{aligned}$$

The overall post condition requires the removal of composites from set s (strictly, there should be a framing constraint on the command but this topic is postponed to Sect. 3.3). The first step of the proof development is justified by set theory but represents a useful strategy for R/G thinking: in a sequential sieving algorithm, it would be easy to specify that the iteration for removing multiples of i had a post condition of $s' = s - c_i$; this equality is too constraining in an environment where concurrent threads are removing elements from s; a common pattern for spotting the post and guarantee conditions in concurrent situations is to weaken an equality so that the post condition gives a bound on the elements that must be removed by the concurrent process (here $s' \cap C = \{\,\}$) and to indicate an upper bound on the elements that can be removed (here $s - s' \subseteq C$) in the guarantee condition; recording that extra conjunct $s' \subseteq s$ both ensures equivalence with the original $s' = s - C$ and records the intuition that no composites can be added back into the set.

Starting the refinement proper, $\mathit{Intro\text{-}g}$ is used to fix the design decision that the set s gets monotonically smaller and that no action removes prime (non composite) numbers from s; $\mathit{Trading\text{-}g\text{-}q}$ can be used to drop the conjunct from the post condition because $s' \subseteq s \wedge s - s' \subseteq C$ expresses a transitive relation; employing $\mathit{Intro\text{-}multi\text{-}parallel}$ requires the insertion of the (matching rely and guarantee) condition on monotonic shrinking of the set s (notice that the post condition would not be realisable were it possible for interfering processes to re-insert values); the penultimate step uses $\mathit{Distribute\text{-}g\text{-}parallel}$; the final step moves to an equivalent specification with the nested guarantees combined.

The final line of Fig. 2 is essentially the expected rely/guarantee specification for a process $\mathit{REM}(i)$. The steps of development from there to the detailed code would not be dissimilar to those in [Jon81] but there are now formal laws for distributing rely and guarantee conditions over loop and sequence constructs (in the earlier publication these were taken as "obvious") and proper laws for introducing assignments (which tended to be handled informally in R/G). Notice that the initial specification ($s' = s - C$) does not fix that exactly the prime numbers are left in s; this piece of program has a precise (if pointless) behaviour if its starting state is empty. A separate step of reasoning shows that using this operation after appropriate initialisation gives the required set of primes up to n.

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

[5] See Appendix A.3 for a fuller set of laws.

$$
\begin{array}{ll}
 & [s' = s - C] \\
= & \text{by set theory} \\
 & [s' \subseteq s \wedge s - s' \subseteq C \wedge s' \cap C = \{\,\}] \\
\sqsubseteq & \text{by } \mathit{Intro\text{-}g} \\
 & \textbf{guar}\ s' \subseteq s \wedge s - s' \subseteq C \bullet [s' \subseteq s \wedge s - s' \subseteq C \wedge s' \cap C = \{\,\}] \\
= & \text{by } \mathit{Trading\text{-}g\text{-}q} \\
 & \textbf{guar}\ s' \subseteq s \wedge s - s' \subseteq C \bullet [s' \cap C = \{\,\}] \\
= & \text{by set theory as } C = \bigcup_{i \in range} c_i \\
 & \textbf{guar}\ s' \subseteq s \wedge s - s' \subseteq C \bullet [\forall i \in range \cdot s' \cap c_i = \{\,\}] \\
\sqsubseteq & \text{by } \mathit{Intro\text{-}multi\text{-}parallel} \\
 & \textbf{guar}\ s' \subseteq s \wedge s - s' \subseteq C \bullet (\|_{i \in range}\ \textbf{guar}\ s' \subseteq s \bullet \textbf{rely}\ s' \subseteq s \bullet [s' \cap c_i = \{\,\}]) \\
= & \mathit{Distribute\text{-}g\text{-}parallel} \\
 & \|_{i \in range}\ (\textbf{guar}\ s' \subseteq s \wedge s - s' \subseteq C \bullet (\textbf{guar}\ s' \subseteq s \bullet (\textbf{rely}\ s' \subseteq s \bullet [s' \cap c_i = \{\,\}]))) \\
= & \mathit{Nested\text{-}g} \\
 & \|_{i \in range}\ (\textbf{guar}\ s - s' \subseteq C \wedge s' \subseteq s \bullet (\textbf{rely}\ s' \subseteq s \bullet [s' \cap c_i = \{\,\}]))
\end{array}
$$

Fig. 2. An (extended) refinement calculus development of *Sieve*

A further example is contained in [HJC14]. The development there tackles a problem introduced in Owicki's thesis [Owi75] and illustrates that R/G is not bound to the granularity of complete statements. Furthermore, this searching example again illustrates the importance of data reification in R/G thinking: an early design decision uses a shared variable t to which both processes need to write; a way to avoid having to lock t is to choose a representation in terms of two variables which both processes can read but each of which can only be changed by one of the processes.

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 Fig. 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 [HJC14] includes an operational semantics, two dozen or so lemmas proved directly from the semantics and over 50 laws derived from the lemmas. The appendix below gives the flavour of the language, some lemmas, a few laws and an indication of the proof style used in the longer report. Hopefully, the new presentation affords a clearer understanding of interference and the algebraic style makes it easier to combine with reasoning about separation and ownership which is the topic of Sect. 3.

Dingel [Din00, Din02] has also considered a "refinement calculus" view of rely/guarantee thinking. However, unlike the approach outlined here (and detailed in [HJC14]), Dingel does not separate the four conditions (p, r, g, q). There are also technical differences from the current presentation concerned with his use of post conditions of single states which force the use of free variables to capture the relationship between initial and final values.

3. Reasoning about separation (race avoidance)

As stated above, the planned research programme will also try to "get underneath" the notation of separation logic (SL) to understand its fundamental contribution. Section 3.1 describes the *issue of separation* and sketches how SL helps to reason about separation; Sect. 3.2 follows O'Hearn's [O'H07] discussion in which he moves from separation to "ownership" (Sect. 4 returns to this issue). Section 3.3 addresses the issue of specifying the "frame" of variables that can be changed by a specified component.

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 the store via integer addresses.

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 what conditions the parallel execution of the code segments would satisfy a specification formed by conjoining their pre/post conditions.[6] 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 situations.

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 an integer address, it is clear that checking separation is more complex.[7] 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 [Hoa72] approach. 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, where x and y are variables containing addresses and the notation $\left[x\right] \leftarrow 3$ means the value 3 is stored in the address contained in the variable x.

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

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:

$$\boxed{SL}\;\frac{\{P_1\}\; s_1\; \{Q_1\} \quad \{P_2\}\; s_2\; \{Q_2\}}{\{P_1 * P_2\}\; s_1 \;||\; s_2\; \{Q_1 * Q_2\}}$$

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

$$\{x \mapsto _ * y \mapsto _\} \;\; \left[x\right] \leftarrow 3 \parallel \left[y\right] \leftarrow 4 \;\; \{x \mapsto 3 * y \mapsto 4\}$$

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:

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

As such, SL is extremely potent for reasoning about disjoint concurrency as is used in the parallel merge sort in [O’H07]. Again, returning to the theme of judicious choice of expressive power, the balance here appears to offer a very compact way of describing separation. The next section explores the related topic of ownership.

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”

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

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

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 express the assertions in 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 also be expressed with R/G conditions. For example, a rely condition for a reader process might record that a buffer (b) does not change when the rd flag is set together with the fact that the environment cannot set rd to false:

$$(rd \;\Rightarrow\; (b' = b)) \wedge (rd \;\Rightarrow\; rd')$$

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

Given that such ownership exchanges are needed for both stack and heap 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 them.

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 Sect. 5.2) to tackle the sieve problem of Sect. 2.3.

Parkinson's [Par10] has the title "The next 700 separation logics"[8] 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 Sect. 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.[9] 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 $x\colon \left[q\right]$ requires that the relational post condition q is achieved with changes only being made to the variable x. (The example in Fig. 2 could be written $s\colon \left[s' = s - C\right]$.) This makes a small step towards the compact notation of separation logic. In fact, the exclusion of identifiers from the write frame is just a shorthand for a guarantee condition. Further notations are offered in [JP11] (**owns**) and [HJC14] (**uses**) that express other constraints on the visibility of identifiers.

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 those issues, 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 Sect. 2.2 above that data abstraction and reification play an important role in rely/guarantee methods. This is not surprising: VDM has emphasised data abstraction in specification and reification in design and [Jon80] was probably the first book to put equal emphasis on data reification and

---

[8] Obviously echoing Landin's [Lan66].

[9] This point appears to be missed by some SL researchers who suggest that R/G conditions can only apply to the global state.

operation decomposition. The current section goes further, both showing that abstraction appears to extend the domain of R/G (Sect. 4.1) and offering a new view on reasoning about separation (Sect. 4.2).

Echoing the *leitmotiv* of this paper, the use of data abstraction is a form of deliberate economy of expressiveness—for example, using a set in a specification makes it immediately clear that no use can be made of expressions about the order of its elements.

4.1. Abstract race avoidance

The data race that occurs in the sieve example (cf. Sect. 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 observing some form of idempotence—in the sieve case, no harm is done by 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 but the analysis using R/G showed that this can be avoided. The property on which this relies is more subtle than in the prime sieve.

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 ACMs. Logically, these are just one place buffers with one writer and one reader but the difficulty derives from the adjective "asynchronous": neither the reader nor the writer can ever be delayed and, of course, the reader must never read 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 reified to 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 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 allow the reader ever to see "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, BA13, WW10] and several observations can be made on these attempts. 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 (at this abstraction level) some unspecified index set X and whose range is the values being passed.[10] 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 a 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 could be reviewed but the interest here is in raising the question of 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.

[10] 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.

There are 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 to and fro between the writer and reader processes. Following the train of argument from separation to ownership in Sects. 3.1/3.2, this would make Simpson's algorithm look to be perfect territory for SL. In fact, [BA13] does not use separation or ownership transfer—it does use a logic that is an interesting combination of R/G and SL known as "RGSep" [VP07, Vaf07] (more is said about this in Sect. 5); the authors also use "linearisability" (cf. Sect. 5) in a novel way. The only publication that appears to use SL to reason about ownership exchange in Simpson's algorithm is [WW10] which confines itself to coherence and stops short of proving the "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 Sect. 4.2 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 of expression in most approaches.

An interesting concept that was used 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 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 ($hold\text{-}r$) acquired the value from another variable ($fresh\text{-}w$) that could be changed by a *Write* process. This was recorded in the earlier, flawed, version of the development by stating that either the initial or final value of $fresh\text{-}w$ 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 new notation in [JP11] for the set of values that can arise and the post condition of the *Read* process can be correctly recorded as $hold\text{-}r \in \widehat{fresh\text{-}w}$. To give a trivial example of the use of this notation, consider writing a post condition for an operation whose implementation sets some local variable x to the value of some global variable y that can be altered by other processes; a trivial implementation is $x \leftarrow y$; but how would one specify this operation? With possible values, the post condition is simply $x \in \widehat{y}$.

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 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.

A critical reader might protest at this point that the *leitmotiv* of expressive weakness appears to have been forgotten. In defence, it can only be pointed out that the possible values of a variable that is being read might indeed be an extra issue to be considered.

4.2. Reasoning about separation as an abstraction

In view of the added value that abstraction gives to R/G approaches, it is worth investigating how much benefit can be drawn by using that same powerful generic idea to tackle the issues where SL appears to be most useful. The proposal here is more speculative than that outlined in Sects. 2.3 and 4.1 but it does indicate that the issue of separation (even in the case of heap variables) can be tackled with little or no extra notation. This, of course, does not argue against the use of SL but it contributes to "pulling apart" of issues from notation.

Reynolds [Rey02] considers a sequential algorithm for in-place reversal of a singly-linked list. He introduces the problem via the program in Fig. 3. Each item in the list is represented by two consecutive memory locations with $\left[e\right]$ containing the data value and $\left[e+1\right]$ the address of the next element in the list.[11] The program is assumed to start with i pointing to the head of the list to be reversed; after execution, j should point to the head of the reversed list; k is a temporary variable that could be avoided by using a multiple assignment statement.

[11] The size of the brackets for $\left[i\right]$ has been increased so as to reduce the risk of confusion with their use in refinement calculus specifications.

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

$$j := \textbf{nil};\ \textbf{while}\ i \neq \textbf{nil}\ \textbf{do}\ (k := [i+1]\,; [i+1] := j; j := i; i := k).$$

The notation $[e]$ denotes the contents of the storage at address e.

Fig. 3. In-place list reversal from [Rey02]

Reynolds' reasoning employs "separating conjunction" as in

$$\exists s, r \cdot list(s, i) * list(r, j)$$

to derive a specification, where $list(s, i)$ states that the heap contains a list structure s for which i is the address of the head of the list. The fact that Reynolds chose to start with the implementation fits with the fact that SL is most commonly used to analyse programs ("bottom up").

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 its final value contains the reverse of the initial value of s.

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

Notice (cf. Sect. 3.3) that this specification gives the designer the permission to overwrite the variable s. The function that reverses a sequence (rev) is obvious.[12]

The specification is satisfied by the following abstract program

$$\begin{array}{l} r \leftarrow \textbf{emptyseq};\\ \textbf{while}\ s \neq \textbf{emptyseq}\ \textbf{do}\\ \quad r, s\colon [s \neq \textbf{emptyseq},\ r' = (\textbf{hd}\, s){:}{:}\, r \wedge s' = \textbf{tl}\, s]\\ \textbf{od} \end{array}$$

To see that this design satisfies the specification, simply note that the loop maintains the value of $rev(s) \frown r$; with the final value of s and the initial value of r as the empty sequence, the result follows immediately.

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 the algorithm in Fig. 3. 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 heap must maintain the essential point of the abstraction of separation.

The requirement to maintain the abstraction of separation thus moves to a data reification step. A representation can be given (for example as a VDM record containing a map and two pointers) and a retrieve function written. The requirement for separation can be added as a conjunct in the invariant of the representation—it can either be written as a normal predicate or use a separating conjunction.

To reiterate the point here: separation is an issue with concurrency—there might be a range of notations that support reasoning about separation—obviously, SL is one possibility.

5. Other approaches

Research on SL is extremely active and, perhaps more surprisingly because of its much earlier inception, R/G research also appears to be accelerating. This section discusses only a small portion of this literature and, even with the publications mentioned, it must be understood that their authors probably had different aims than those of the current paper.

5.1. SLs meet R/G

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, in which logical disjunction of predicates is represented by "∪"

[12] In order to avoid confusion with the two other uses of square brackets, use of this shape of bracket for VDM sequences is avoided: $e{:}{:}\,s$ is used in place of $[e] \frown s$; **emptyseq** is used in place of VDM's normal [ ]; and the definition of rev is omitted since its semantics should be obvious.

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

Despite the current authors' admiration for this valuable contribution, it must be said that the *RGSep* rule presents a rather "syntactic" combination of the two approaches in that bringing different "issues" into one rule does not offer any insight as to their application. The "Taming Concurrency" project is aiming to combine more fundamental insights from SL and R/G.

In the same vein, Feng's SAGL [FFS07] argues that SL can be viewed as a specialisation of "assume guarantee" methods for a class of programs. More recently, Feng [Fen09] has proposed "local rely-guarantee reasoning" which does tackle broadly the same issues as are addressed in the current paper. It is not completely clear that the point raised in Sect. 3.3 above about even classical VDM restricting the frames by **rd**/**wr** clauses is appreciated. However, the idea of applying an operator like separating conjuction to relations (rather than just single-state predicates) is certainly worth further study. Another interesting contribution to R/G thinking from Matt Parkinson and his colleagues is the "Deny/Guarantee" idea in [DFPV09].

Another paper that refers to R/G and SL is [SB14]. The proposed "impredicative concurrent abstract predicates" tackle sharing using both higher-order concepts and fractional permissions to achieve modular reasoning although Wickerson's [Wic13] thesis makes a better case for modular reasoning. A recent contribution ([Lia14]) introduces "Rely-Guarantee-based simulation".

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" (CAPs) [DYDG+10] sounds as though it might be in the same groove as the case being made for abstraction in Sect. 4 above. In fact, the relationship is certainly more subtle with CAPs being used to handle complicated 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. Another important avenue is the research on concurrent Kleene algebras which is linked to R/G in [HMSW11]. This chimes with the recent research by the current authors.

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. 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 Moszkowski's [Mos85] "interval temporal logic" (ITL) with R/G in 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 [BA13] touches on linearisability and another of the impressive aspects of [Vaf07] is 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 [Jon03, 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.

The recent paper on CaRESL [TDB13] makes some similar points to the earlier work on atomicity refinement but comes to a rather different solution which requires further study. In particular, CaReSL employs higher-order predicates to tackle what these authors call "granularity abstraction".

6. Conclusions

The current authors have 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 as 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 previously recorded 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 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 Sect. 5.5; an interesting limitation of the form of temporal assertion needed is given in [GCPV09]. Another limitation of R/G is identified in Wickerson's [Wic13] thesis: he makes the point that compositionality does not ensure that a method (specifically R/G) can handle modular development (where a component can be used in various contexts).

Moving forward, the authors' goal will *not* be to make arbitrary extensions to existing R/G notation but rather to understand an issue 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 Sect. 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. Short-term objectives include the analysis of more examples (particularly those from SL), a more careful analysis of data abstraction/reification and—ultimately—the provision of machine support for the ideas in [HJC14].

Acknowledgments

The invitation for the first author to speak at SEFM gave rise to [Jon12a]; little of that paper remains here but the trip to Thessaloniki and the hospitality received do remain warm memories. A subsequent invitation to ICECCS was also made enjoyable by thoughtful and generous hosts; scientifically, it provided the opportunity to try out the ideas on expressiveness presented here. It is a pleasure to thank our research collaborators: 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, Oxford and York. Our colleagues Leo Freitas and Diego Machado Dias both provided detailed comments on an earlier draft of the current paper; the anonymous referees used by the journal raised many questions that have hopefully resulted in a more readable paper. Matt Parkinson's August 2013 visit to give a seminar in Newcastle (based on [DYBG+13]) was timely and prompted clarification of a number of remarks about SL—as well as providing other stimulating discussions. The authors of this paper gratefully acknowledge the funding for their research from the EPSRC Platform Grant "TrAmS-2", the EPSRC responsive mode grant on "Taming Concurrency" and the ARC grant DP130102901 on "Understanding concurrent programmes using rely/guarantee thinking".

Open Access This article is distributed under the terms of the Creative Commons Attribution License which permits any use, distribution, and reproduction in any medium, provided the original author(s) and the source are credited.

A. Precis of a refinement calculus for rely/guarantee reasoning

This section contains a summary of the refinement framework developed in the associated technical report [HJC14]. A subset of the wide-spectrum language that contains specification constructs and code is presented first, then a set of fundamental lemmas giving the basic properties of the language and finally a subset of the higher-level refinement laws that may be applied directly in derivations such as the *Sieve* in Fig. 2. The technical report [HJC14] defines an operational semantics in which the primitives of the language are defined and with respect to which the lemmas are proved sound. It also presents a much richer set of refinement laws and their proofs; this appendix provides only some illustrative proofs, with a focus on those used in Fig. 2.

Basic commands. Let p be a predicate, q be a relation, C a set of commands, b a boolean expression, X a set of variables, x a variable, and v a value.

$$c ::= \langle p, q \rangle \quad | \quad [q] \quad | \quad \{p\} \quad | \quad \bigsqcap C \quad | \quad c_1 \Cap c_2 \quad | \quad c_1 \,;\, c_2 \quad | \quad c_1 \parallel c_2 \quad | \quad [[b]]$$

Fig. 4. Syntax of basic command

A.1. Specification language

The basic commands are given in Fig. 4; they include an *atomic specification* $\langle p, q \rangle$, which may take a single step that satisfies q provided p holds, and otherwise may *abort*, that is, engage in any behaviour. The atomic step may be preceded or succeeded by interference from the envionment. A *specification command* $[q]$ represents any command which, when considered from the initial state to the final state, satisfies the relation q provided interference from the environment consists only of stuttering steps (i.e. steps that do not change the state). The process of refinement is essentially about reducing the nondeterminism inherent in $[q]$ to a particular implementation. A *precondition* $\{p\}$ does nothing if the current state satisfies p, but aborts otherwise. A *nondeterministic choice* $\bigsqcap C$ may behave as any of the individual commands in the set C. The more familiar binary choice is a special case where C contains exactly two elements. A *strict conjunction* $c_1 \Cap c_2$ behaves in a manner that is consistent with both commands, with the caveat that if either aborts then the conjunction aborts; each atomic step made by the conjunction must be an atomic step that both can make, unless one can abort in which case the conjunction aborts. This operator simplifies the theory and underlies many of the important definitions and proofs. Sequential composition is the usual sequencing of execution. Binary parallel composition $c_1 \parallel c_2$ is also the usual concept of interleaving of actions. The multi-way parallel composition $\|_{i \in S}\, c_i$ used in the body of the paper is its generalisation to multiple processes executing in parallel, and is defined using binary parallel composition inductively over the finite index set S. Finally, a *test* $[[b]]$, where b is a boolean expression, non-atomically evaluates b and may progress if b evaluates to true. Tests are used in the definition of conditional and while commands, below.

From these basic commands, the set of specification and code constructs as given in Fig. 5 is derived. A command may be iterated a finite number of times (c^* for zero or more) and either a finite or infinite number of times (c^ω). Iterations of commands are defined via greatest (ν) and least (μ) fixed points with respect to the refinement ordering [vW04]. The familiar if-statement **if** b **then** c_0 **else** c_1 is a choice between non-atomically evaluating the test b to true and executing c_0 and non-atomically evaluating b to false ($\neg b$ to true) and executing c_1. A loop **while** b **do** c is defined as a (potentially infinite) iteration of evaluating b to true and executing the body of the loop, c, followed by an evaluation of b to false.

An assignment $x := e$ is a choice over all possible values v to which e may be evaluated, followed by atomically setting x to v.

The guarantee command (**guar** $g \bullet c$) is defined so that each atomic step of c must satisfy g or stutter (i.e. satisfy the identity relation id), using program conjunction to restrict the behaviours of c to only those that satisfy the constraint. For brevity, the definition of the rely command (**rely** $r \bullet c$) is omitted here but is given in [HJC14]; the following property however demonstrates the interaction of a rely with a specification command.

$$[p,\ q] \sqsubseteq (\mathbf{rely}\ r \bullet [p,\ q]) \parallel \langle r \vee \mathrm{id} \rangle^*$$

It states that (**rely** $r \bullet [p,\ q]$) is an implementation of the specification command $[p,\ q]$ when it executes in parallel with interference satsifying r.

A.2. Lemmas

This section includes a subset of the basic properties of the language as lemmas, which in [HJC14] are proved sound with respect to an operational semantics. These lemmas provide the foundation for algebraic proofs of higher-level refinement laws.

Some of the lemmas use a parameterised version of refinement $\sqsubseteq_r$, where $c \sqsubseteq_r d$ states that d refines c when considering only interference satsifying r. The notation $p_1 \Rrightarrow p_2$ means that the implication holds for all states.

Derived commands. Let p be a predicate, q, g and r be relations, c, c_0 and c_1 be commands, b a boolean expression, e an expression, x a variable, and Val the set of values. For a set of variables X, the relation $\mathrm{id}(X)$ is the identity relation on X. For a variable x, $\bar{x}$ is the set of all variables other than x.

$$\langle q \rangle == \langle \mathbf{true}, q \rangle \tag{1}$$
$$[p,\ q] == \{p\}\,[q] \tag{2}$$
$$c_0 \sqcap c_1 == \bigsqcap \{c_0, c_1\} \tag{3}$$
$$c^* == \nu x \bullet \mathbf{skip} \sqcap c\,;x \tag{4}$$
$$c^\omega == \mu x \bullet \mathbf{skip} \sqcap c\,;x \tag{5}$$
$$\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1 == ([[b]]\,;c_0) \sqcap ([[\neg b]]\,;c_1) \tag{6}$$
$$\mathbf{while}\ b\ \mathbf{do}\ c == ([[b]]\,;c)^\omega\,;[[\neg b]] \tag{7}$$
$$x := e == \bigsqcap \{v \in Val \bullet [[e = v]]\,;\langle x' = v \wedge \mathrm{id}(\bar{x})\rangle\} \tag{8}$$
$$\mathbf{guar}\ g \bullet c == \langle g \vee \mathrm{id}\rangle^\omega \Cap c \tag{9}$$

The notation $\{v \in V \bullet f\}$ stands for the set of values of the expression f for v in the set V. The notation $\{v \in V \bullet f\}$ is often written $\{f \mid v \in V\}$ but it is preferable not to use the latter because it is ambiguous as to whether v is bound within the set comprehension or a free variable being tested for membership of V.

Fig. 5. Derived commands

Lemma 1 (Precondition) *For any predicates p, p_0 and p_1, relation r and commands c and d,*

$$\{p_0\}\{p_1\} \;=\; \{p_0 \wedge p_1\} \tag{10}$$
$$(p_0 \Rrightarrow p_1) \;\Rightarrow\; (\{p_0\} \sqsubseteq \{p_1\}) \tag{11}$$
$$(\{p\}c \sqsubseteq_r \{p\}d) \;\Leftrightarrow\; (\{p\}c \sqsubseteq_r d) \tag{12}$$

Lemma 2 (Parallel-precondition) *For any predicate p, and commands c and d,*

$$\{p\}(c \parallel d) = (\{p\}c) \parallel (\{p\}d)$$

Lemma 3 (Make-atomic) *For any predicate p and relation q, $\left[p,\ q\right] \sqsubseteq \langle p, q\rangle$.*

Lemma 4 (Consequence) *For any predicates p_0 and p_1, and relations q_0 and q_1, provided $p_0 \Rrightarrow p_1$ and $p_0 \wedge q_1 \Rrightarrow q_0$,*

$$\langle p_0, q_0\rangle \sqsubseteq \langle p_1, q_1\rangle \tag{13}$$
$$\left[p_0,\ q_0\right] \sqsubseteq \left[p_1,\ q_1\right] \tag{14}$$

Lemma 5 (Sequential) *For any predicates p_0 and p_1, and relations q, q_0 and q_1, such that $p_0 \wedge ((q_0 \wedge p_1') \mid q_1) \Rrightarrow q$,*

$$\left[p_0,\ q\right] \sqsubseteq \left[p_0,\ q_0 \wedge p_1'\right];\ \left[p_1,\ q_1\right].$$

Lemma 6 (Introduce-test) *For any boolean expression b, $\left[def(b),\ b \wedge \mathrm{id}\right] \sqsubseteq [[b]]$.*

Lemma 7 (Iteration-induction) *For any relation r and commands c, d and x,*

$$x \sqsubseteq_r d \sqcap c\,;\ x \;\Rightarrow\; x \sqsubseteq_r c^*\,;\ d \tag{15}$$
$$d \sqcap c\,;\ x \sqsubseteq_r x \;\Rightarrow\; c^\omega\,;\ d \sqsubseteq_r x \tag{16}$$

Lemma 8 (Conjunction-monotonic) *For any relation r and commands c_0, c_1, d_0 and d_1,*

$$(c_0 \sqsubseteq_r d_0) \wedge (c_1 \sqsubseteq_r d_1) \;\Rightarrow\; (c_0 \Cap c_1) \sqsubseteq_r (d_0 \Cap d_1)$$

Lemma 9 (Conjunction-strict) *For any predicate p and commands c and d,*

$$\{p\}(c \Cap d) = (\{p\}c) \Cap d = c \Cap (\{p\}d) = (\{p\}c) \Cap (\{p\}d).$$

Lemma 10 (Interchange-conjunction) *For any commands c_0, c_1, d_0 and d_1, the following hold.*

$$(c_0 \parallel c_1) \Cap (d_0 \parallel d_1) \sqsubseteq (c_0 \Cap d_0) \parallel (c_1 \Cap d_1) \tag{17}$$
$$(c_0\,;\ c_1) \Cap (d_0\,;\ d_1) \sqsubseteq (c_0 \Cap d_0)\,;\ (c_1 \Cap d_1) \tag{18}$$

Lemma 11 (Distribute-conjunction) *For any relation g and commands c and d,*

$$\langle g\rangle^{\omega} \Cap (c \parallel d) = (\langle g\rangle^{\omega} \Cap c) \parallel (\langle g\rangle^{\omega} \Cap d)$$

Lemma 12 (Refine-in-guarantee-context) *For relations g and r and commands c_0, c_1 and d, such that $c_0 \sqsubseteq_{g \vee r} c_1$,*

$$c_0 \parallel (\mathbf{guar}\ g \bullet d) \sqsubseteq_r c_1 \parallel (\mathbf{guar}\ g \bullet d).$$

Lemma 13 (Tests) *For any boolean expressions a and b the following hold.*

$$[[a \wedge b]] = [[a]] \parallel [[b]] \sqsubseteq \quad [[a]]\ ;\ [[b]] \quad = [[a\ \mathbf{cand}\ b]]$$
$$[[a \vee b]] = [[a]] \sqcap [[b]] \sqsubseteq [[a]] \sqcap [[\neg\, a]]\ ;\ [[b]] = [[a\ \mathbf{cor}\ b]]$$

and for variables x and y assuming atomic access to a single variable,

$$[[x < y]] = \bigsqcap \left\{v \in Val, w \in Val \mid v < w \bullet \langle x = v \wedge \mathrm{id}\rangle \parallel \langle y = w \wedge \mathrm{id}\rangle\right\}$$

and other relational operators are treated similarly.

Lemma 14 (Parallel-interference) *For any relations r_0 and r_1,* $\quad \langle r_0\rangle^* \parallel \langle r_1\rangle^* = \langle r_0 \vee r_1\rangle^*$.

Lemma 15 (Repeated-interference) *For any relation r,* $\quad \langle r\rangle^*\ ;\ \langle r\rangle^* = \langle r\rangle^*$.

Lemma 16 (Interference-atomic) *For any predicate p and relations q and r,*

$$\langle p, q\rangle \parallel \langle r\rangle^* = \langle r\rangle^*\ ;\ \langle p, q\rangle\ ;\ \langle r\rangle^*.$$

Lemma 17 (Distribute-parallel) *For any commands c_0, c_1 and d, non-empty set of commands C, and relation r,*

$$\left(\bigsqcap C\right) \parallel d = \bigsqcap \{c \in C \bullet (c \parallel d)\} \tag{19}$$

$$(c_0\ ;\ c_1) \parallel \langle r\rangle^* = (c_0 \parallel \langle r\rangle^*)\ ;\ (c_1 \parallel \langle r\rangle^*) \tag{20}$$

$$(c_0 \parallel c_1) \parallel \langle r\rangle^* = (c_0 \parallel \langle r\rangle^*) \parallel (c_1 \parallel \langle r\rangle^*) \tag{21}$$

A.3. Derived laws

This section presents a set of higher-level refinement laws which can be derived from the lemmas. The laws presented below are the subset of laws used in the development of the *Sieve*. A much larger set of laws and an additional example of their application is given in [HJC14]. Proofs of several of the laws are included to show the incremental and algebraic style of proof that may be adopted; all of the laws are proved in full in [HJC14].

Law 18 (*Refinement-monotonic*) For any commands c and d and relations r_0 and r_1, if $r_0 \Rrightarrow r_1 \vee \mathrm{id}$ and $c \sqsubseteq_{r_1} d$, then $c \sqsubseteq_{r_0} d$.

Law 19 (*Refine-iterated-relation*) For any relation g, $\quad \left[g^*\right] \sqsubseteq \langle g\rangle^*$.

A.3.1. Laws for program conjunction

Law 20 (*Refine-conjunction*) For any commands c_0, c_1 and d, if $c_0 \sqsubseteq_r d$ and $c_1 \sqsubseteq_r d$, then $c_0 \Cap c_1 \sqsubseteq_r d$.

Proof Any trace of d in environment r must also be a trace of both c_0 and c_1, and hence it is a trace of $c_0 \Cap c_1$, noting that if either c_0 or c_1 can abort their conjunction also can. □

Law 21 (*Simplify-conjunction*) For commands c and d, if $c \sqsubseteq_r d$, $\quad c \Cap d \sqsubseteq_r d$.

Proof The proof follows from Law 20 as $c \sqsubseteq_r d$ and $d \sqsubseteq_r d$. □

Law 22 (*Conjoined-specifications*) For any relations q_0 and q_1,

$$\left[q_0 \wedge q_1\right] = \left[q_0\right] \Cap \left[q_1\right].$$

Law 23 (*Conjunction-with-terminating*) For any predicate p, relations r and g, and command c, such that $p \Rrightarrow stops(c, r)$,

$$\{p\}\langle g\rangle^* \sqsubseteq_r \langle g\rangle^{\omega} \Cap (\{p\}c).$$

Law 24 (*Conjunction-atomic-iterated*) For any relations g_0 and g_1 both the following hold.

$$\langle g_0\rangle^{\omega} \Cap \langle g_1\rangle^{\omega} \;=\; \langle g_0 \wedge g_1\rangle^{\omega} \tag{22}$$
$$\langle g_0\rangle^{*} \Cap \langle g_1\rangle^{*} \;=\; \langle g_0 \wedge g_1\rangle^{*} \tag{23}$$

A.3.2. Laws for the guarantee command

Law 25 (*Guarantee-monotonic*) For any commands c and d, and relations g and r,

$$c \sqsubseteq_r d \quad\Rightarrow\quad (\mathbf{guar}\ g \bullet c) \sqsubseteq_r (\mathbf{guar}\ g \bullet d).$$

Proof The proof relies on Lemma 8. If $c \sqsubseteq_r d$,
$(\mathbf{guar}\ g \bullet c) = \langle g \vee \mathrm{id}\rangle^{\omega} \Cap c \sqsubseteq_r \langle g \vee \mathrm{id}\rangle^{\omega} \Cap d = (\mathbf{guar}\ g \bullet d)$. □

Law 26 (*Guarantee-true*) For any command c, $(\mathbf{guar\ true} \bullet c) = c$.

Law 27 (*Strengthen-guarantee*) For any command c and relations g_0 and g_1,

$$(g_0 \Rrightarrow g_1 \vee \mathrm{id}) \;\Rightarrow\; (\mathbf{guar}\ g_1 \bullet c) \sqsubseteq (\mathbf{guar}\ g_0 \bullet c).$$

Law 28 (*Intro-g*) For any command c and relation g, $\;c \sqsubseteq (\mathbf{guar}\ g \bullet c)$.

Proof The proof follows from Laws 26 and 27:
$c = (\mathbf{guar\ true} \bullet c) \sqsubseteq (\mathbf{guar}\ g \bullet c)$ □

Law 29 (*Nested-g*) For a command c and relations g_0 and g_1,

$$(\mathbf{guar}\ g_0 \bullet (\mathbf{guar}\ g_1 \bullet c)) = (\mathbf{guar}\ g_0 \wedge g_1 \bullet c)\ .$$

Proof The proof relies on the property of relations: $(g_0 \vee \mathrm{id}) \wedge (g_1 \vee \mathrm{id}) = (g_0 \wedge g_1) \vee \mathrm{id}$.

$(\mathbf{guar}\ g_0 \bullet (\mathbf{guar}\ g_1 \bullet c))$
$=$ by (9) twice, "$\Cap$" is associative
$\langle g_0 \vee \mathrm{id}\rangle^{\omega} \Cap \langle g_1 \vee \mathrm{id}\rangle^{\omega} \Cap c$
$=$ by Law 24 part (22) using the above property of relations
$\langle (g_0 \wedge g_1) \vee \mathrm{id}\rangle^{\omega} \Cap c$
$= (\mathbf{guar}\ g_0 \wedge g_1 \bullet c)$

□

Law 30 (*Distribute-g-parallel*) For any relation g and commands c and d,

$$\mathbf{guar}\ g \bullet (c \parallel d) \;=\; (\mathbf{guar}\ g \bullet c) \parallel (\mathbf{guar}\ g \bullet d)$$

Proof Follows directly from the definition of guarantee (9) and Lemma 11. □

Law 31 (*Trading-g-q*) For any relations g and q,

$$(\mathbf{guar}\ g \bullet \lceil g^* \wedge q \rceil) = (\mathbf{guar}\ g \bullet \lceil q \rceil).$$

Proof By Lemma 4 $\lceil q \rceil \sqsubseteq \lceil g^* \wedge q \rceil$ and hence by Law 25 the refinement holds from right to left. The refinement from left to right below uses the fact that $\langle g \vee \mathrm{id}\rangle^{*} \sqsubseteq_{\mathrm{id}} \langle g \vee \mathrm{id}\rangle^{\omega} \Cap \lceil q \rceil$ by Law 23.

$\mathbf{guar}\ g \bullet \lceil g^* \wedge q \rceil$
$=$ by (9), Law 22
$\langle g \vee \mathrm{id}\rangle^{\omega} \Cap \lceil g^* \rceil \Cap \lceil q \rceil$
$=$ by Law 19 and $(g \vee \mathrm{id})^* = g^*$
$\langle g \vee \mathrm{id}\rangle^{\omega} \Cap \langle g \vee \mathrm{id}\rangle^{*} \Cap \lceil q \rceil$
$\sqsubseteq_{\mathrm{id}}$ by Law 21 as $\langle g \vee \mathrm{id}\rangle^{*} \sqsubseteq_{\mathrm{id}} \langle g \vee \mathrm{id}\rangle^{\omega} \Cap \lceil q \rceil$
$\langle g \vee \mathrm{id}\rangle^{\omega} \Cap \lceil q \rceil$
$=$ by (9)
$\mathbf{guar}\ g \bullet \lceil q \rceil$

□

Note that Laws 28, 29, 30, and 31 correspond to laws given in Sect. 2.3.

A.3.3. Introducing concurrency

The following law decomposes a specification to two parallel processes, each with a stronger rely condition and weaker guarantee, with the trade-off that each need fulfill a weaker specification. It corresponds to the parallel introduction law of [Jon83b], and is a binary version of the law *Intro-multi-* $\parallel$ used in Sect. 2.3.

Law 32 (*Introduce-parallel-spec*) For any predicates p, p_0 and p_1, and relations q, q_0, q_1, g_0 and g_1, such that $p \Rrightarrow p_0 \wedge p_1$ and $p \wedge q_0 \wedge q_1 \Rrightarrow q$,

$$[p,\ q] \sqsubseteq (\textbf{guar}\ g_0 \bullet (\textbf{rely}\ g_1 \bullet [p_0,\ q_0])) \parallel (\textbf{guar}\ g_1 \bullet (\textbf{rely}\ g_0 \bullet [p_1,\ q_1])).$$

References

[Abr96] Abrial JR (1996) The B-book: assigning programs to meanings. Cambridge University Press, Cambridge

[Abr10] Abrial JR (2010) The event-B book. Cambridge University Press, Cambridge

[AM71] Ashcroft EA, Manna Z (1971) Formalization of properties of parallel programs. In: Meltzer B, Michie D (eds) Machine intelligence, vol 6. Edinburgh University Press, Edinburgh, pp 17–41

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

[BA10] Bornat R, Amjad H (2010) Inter-process buffers in separation logic with rely-guarantee. Form Asp Comput 22(6):735–772

[BA13] Bornat R, Amjad H (2013) Explanation of two non-blocking shared-variable communication algorithms. Form Asp Comput 25(6):893–931

[BGMY12] Burckhardt S, Gotsman A, Musuvathi M, Yang H (2012) Concurrent library correctness on the TSO memory model. In: ESOP

[Boy03] Boyland J (2003) Checking interference with fractional permissions. In: Cousot R (ed) Static analysis, vol 2694 of LNCS. Springer, New York, pp 55–72

[BS91] Back RJ, Sere K (1991) Stepwise refinement of action systems. Struct Program 12:17–30

[BvW98] Back R-JR, von Wright J (1998) Refinement calculus: a systematic introduction. Springer, New York

[CJ00] Collette P, Jones CB (2000) Enhancing the tractability of rely/guarantee specifications in the development of interfering operations. In: Plotkin G, Stirling C, Tofte M (eds) Proof, language and interaction, vol 10. MIT Press, USA, pp 277–307

[CJ07] Coleman JW, Jones CB.(2007) A structural proof of the soundness of rely/guarantee rules. J Log Comput 17(4):807–841

[Col08] Coleman JW (2008) Constructing a tractable reasoning framework upon a fine-grained structural operational semantics. PhD thesis, Newcastle University

[Cou08] Cousot P (2008) The verification grand challenge and abstract interpretation. In: Meyer B, Woodcock J (eds) Verified software: theories, tools, experiments, vol 4171. Lecture notes in computer science. Springer, Berlin/Heidelberg, pp 189–201. doi:10.1007/978-3-540-69149-5_21

[DDH72] Dahl OJ, Dijkstra EW, Hoare CAR (1972) Structured programming. Academic Press, Dublin

[DFPV09] Dodds M, Feng X, Parkinson M, Vafeiadis V (2009) Deny-guarantee reasoning. In: Castagna G (ed) Programming languages and systems, vol 5502. Lecture notes in computer science. Springer, Berlin/Heidelberg, pp 363–377

[Dij68] Dijkstra EW (1968) Letters to the editor: go to statement considered harmful. Commun ACM 11(3):147–148

[Din00] Dingel J (2000) Systematic parallel programming. PhD thesis, Carnegie Mellon University, CMU-CS-99-172

[Din02] Dingel J (2002) A refinement calculus for shared-variable parallel and distributed programming. Form Asp Comput 14(2):123–197

[dR01] de Roever WP (2001) Concurrency verification: introduction to compositional and noncompositional methods. Cambridge University Press, Cambridge

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

[DYBG+13] Dinsdale-Young T, Birkedal L, Gardner P, Parkinson M, Yang H (2013) Views: compositional reasoning for concurrent programs. In: Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages, pp 287–300. ACM

[DYDG+10] Dinsdale-Young T, Dodds M, Gardner P, Parkinson MJ, Vafeiadis V (2010) Concurrent abstract predicates. In: Proceedings of the 24th European conference on object-oriented programming. Springer, Berlin, pp 504–528

[Fen09] Feng X (2009) Local rely-guarantee reasoning. In: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL'09. ACM, New York, pp 315–327

[FFS07] Feng X, Ferreira R, Shao Z (2007) On the relationship between concurrent separation logic and assume-guarantee reasoning. In: ESOP: programming languages and systems. Springer, New York, pp 173–188

[Flo67] Floyd RW (1967) Assigning meaning to programs. Math Asp Comput Sci 19:19–32

[GCPV09] Gotsman A, Cook B, Parkinson M, Vafeiadis V (2009) 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. ACM, New York, pp 16–28

[GY11] Gotsman A, Yang H (2011) Liveness-preserving atomicity abstraction. In: ICALP

[HA10] Hoang TS, Abrial JR (2010) Event-B decomposition for parallel programs. In: Frappier M, Glaesser U, Sarfraz K, Laleau R, Reeves S (eds) ABZ, vol 5977 of LNCS. Springer, New York, pp 319–333

[HBDJ13] Hayes IJ, Burns A, Dongol B, Jones CB (2013) Comparing degrees of non-deterministic in expression evaluation. Comput J 56(6):741–755

[HJC14] Hayes IJ, Jones CB, Colvin RJ (2014) Laws and semantics for rely-guarantee refinement. Technical report CS-TR-1425, Newcastle University

[HMSW11] Hoare T, Möller B, Struth G, Wehrman I (2011) Concurrent Kleene algebra and its foundations. J Log Algebra Program 80(6):266–296

[Hoa69] Hoare CAR (1969) An axiomatic basis for computer programming. Commun ACM 12(10):576–580, 583

[Hoa72] Hoare CAR (1972) Towards a theory of parallel programming. In: Operating system techniques. Academic Press, Dublin, pp 61–71

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

[Jon80] Jones CB (1980) Software development: a rigorous approach. Prentice Hall International, Englewood Cliffs

[Jon81] Jones CB (1981) Development methods for computer programs including a notion of interference. PhD thesis, Oxford University. Printed as: Programming Research Group, Technical Monograph 25.

[Jon83a] Jones CB (1983) Specification and design of (parallel) programs. In: Proceedings of IFIP'83, North-Holland, pp 321–332

[Jon83b] Jones CB (1983) Tentative steps toward a development method for interfering programs. Trans Program Lang Syst 5(4):596–619

[Jon90] Jones CB (1990) Systematic software development using VDM, 2nd edn. Prentice Hall International, USA

[Jon96] Jones CB (1996) Accommodating interference in the formal design of concurrent object-based programs. Form Methods Syst Des 8(2):105–122

[Jon03] Jones CB (2003) Wanted: a compositional approach to concurrency. In: McIver A, Morgan C (eds) Programming methodology. Springer, New York, pp 5–15

[Jon07] Jones CB (2007) Splitting atoms safely. Theor Comput Sci 375(1–3):109–119

[Jon10] Jones CB (2010) The role of auxiliary variables in the formal development of concurrent programs. In: Jones CB, Roscoe AW, Wood K (eds) Reflections on the work of C.A.R. Hoare, vol 8. Springer, New York, pp 167–188

[Jon12a] Jones CB (2012) Abstraction as a unifying link for formal approaches to concurrency. In: Eleftherakis G, Hinchey M, Holcombe M (eds) Software engineering and formal methods, vol 7504. Lecture notes in computer science, pp 1–15

[Jon12b] Jones CB (2012) A specification for ACMs. Technical report CS-TR-1360, Newcastle University

[JP08] Jones CB, Pierce KG (2008) Splitting atoms with rely/guarantee conditions coupled with data reification. In: ABZ2008, number 5238 in lecture notes in computer science. Springer, New York, pp 360–377

[JP11] Jones CB, Pierce KG (2011) Elucidating concurrent algorithms via layers of abstraction and reification. Form Asp Comput 23(3):289–306

[Lan66] Landin PJ (1966) The next 700 programming languages. Commun. ACM 9(3):157–166

[Lia14] Liang H (2014) Refinement verification of concurrent programs and its applications. PhD thesis, USTC, China

[Mid93] Middelburg CA (1993) Logic and specification: extending VDM-SL for advanced formal specification. Chapman and Hall, London

[Mor87] Morris JM (1987) A theoretical basis for stepwise refinement and the programming calculus. Sci Comput Program 9(3):287–306

[Mor94] Morgan CC (1994) Programming from specifications, 2nd edn. Prentice Hall, USA

[Mos85] Moszkowski B (1985) Executing temporal logic programs. In: Brookes SD, Roscoe AW, Winskel G (eds) Seminar on concurrency, vol 197 of LNCS. Springer, Berlin, pp 111–130

[OG76] Owicki SS, Gries D (1976) An axiomatic proof technique for parallel programs I. Acta Inform 6:319–340

[O'H07] O'Hearn PW (2007) Resources, concurrency and local reasoning. Theor Comput Sci 375(1–3):271–307

[Owi75] Owicki S (1975) Axiomatic proof techniques for parallel programs. PhD thesis, Department of Computer Science, Cornell University

[Par10] Parkinson M (2010) The next 700 separation logics. In: Leavens G, O'Hearn P, Rajamani S (eds) Verified software: theories, tools, experiments, vol 6217. Lecture notes in computer science. Springer, Berlin/Heidelberg, pp 169–182

[PBC06] Parkinson M, Bornat R, Calcagno C (2006) Variables as resource in Hoare logics. In: Proceedings of the 21st annual IEEE symposium on logic in computer science, pp 137–146

[Pre01] Prensa Nieto L (2001) Verification of parallel programs with the owicki-gries and rely-guarantee methods in Isabelle/HOL. PhD thesis, Institut für Informatic der Technischen Universitaet München

[Pre03] Prensa Nieto L (2003) The rely-guarantee method in Isabelle/HOL. In: Proceedings of ESOP 2003, vol 2618 of LNCS. Springer, New York

[Rey00] Reynolds JC (2000) Intuitionistic reasoning about shared mutable data structure. In: Davies J, Roscoe B, Woodcock J (eds) Millennial perspectives in computer science. Houndsmill, Hampshire, Palgrave, pp 303–321

[Rey02] Reynolds JC (2002) Separation logic: a logic for shared mutable data structures. In: Proceedings of 17th LICS. IEEE, pp 55–74

[Rod08] Rodin (2008) Event-B and the Rodin platform. http://www.event-b.org.

[San99] Sangiorgi D (1999) Typed π-calculus at work: a correctness proof of Jones's parallelisation transformation on concurrent objects. Theory Pract Obj Syst 5(1):25–34

[SB14] Svendsen K, Birkedal L (2014) Impredicative concurrent abstract predicates. In: Programming languages and systems. Springer, New York, pp 149–168

[Sim90] Simpson HR (1990) Four-slot fully asynchronous communication mechanism. Comput Dig Tech IEE Proc E 137(1):17–30

[STER11] Schellhorn G, Tofan B, Ernst G, Reif W (2011) Interleaved programs and rely-guarantee reasoning with ITL. In: Proceedings opf the eighteenth international symposium on temporal representation and reasoning (TIME), pp 99–106

[Stø90] Stølen K (1990) Development of parallel programs on shared data-structures. PhD thesis, Manchester University. Available as UMCS-91-1-1.

[TDB13] Turon A, Dreyer D, Birkedal L (2013) Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In: Proceedings of the 18th ACM SIGPLAN international conference on functional programming, ICFP'13. ACM, pp 377–390

[Vaf07] Vafeiadis V (2007) Modular fine-grained concurrency verification. PhD thesis, University of Cambridge

[VP07] Vafeiadis V, Parkinson M (2007) A marriage of rely/guarantee and separation logic. In: Caires L, Vasconcelos V (eds) CONCUR 2007— concurrency theory, vol 4703 of LNCS. Springer, NEw York, pp 256–271

[vW04] von Wright J (2004) Towards a refinement algebra. Sci Comput Program 51:23–45

[Wic13] Wickerson J (2013) Concurrent verification for sequential programs. PhD thesis, Cambridge

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

Received 19 August 2013
Revised 29 June 2014
Accepted 8 July 2014 by George Eleftherakis, Mike Hinchey, Michael Butler and Jim Woodcock
Published online 29 August 2014