Specifying and Reasoning About Shared-Variable Concurrency

Ian J. Hayes[1(✉)], Cliff B. Jones[2], and Larissa A. Meinicke[1]

[1] School of Electrical Engineering and Computer Science, The University of Queensland, Brisbane 4072, QLD, Australia
Ian.Hayes@uq.edu.au

[2] School of Computing, Newcastle University, 1, Science Square, Newcastle upon Tyne NE4 5TG, UK

Abstract. Specifications are a necessary reference point for correctness arguments. Top-down descriptions of concurrent programs require a way of recording information about the environment in which the component will be required to function. It was shown in the 1980s that adding rely and guarantee conditions to pre and post conditions could support formal specification and reasoning about a class of concurrent systems. More recent research has both widened the class of specifications to include progress requirements and facilitated mechanisation of proofs. This paper describes the algebraic underpinnings that have made this possible. Particular attention is paid to notions of atomicity.

Keywords: shared-variable concurrency · rely-guarantee approach · refinement calculus · program algebra · atomic specification commands

1 Introduction

At the heart of an effective software development method is the ability to specify a program component independently from its implementation. From the point of view of deployment, such an independent specification should allow the use of a component to depend solely on its specification (and not on the details of a particular implementation of the component). Considering the task of its developers, the correctness of an implementation of a component should depend solely on its specification (and not the context(s) in which it is used). The techniques required to achieve this for sequential programs are both well established[1] and used in practical development environments.

When compared with sequential programs, reasoning about concurrent programs introduces the additional complexities of inherent nondeterminism and interference between threads that gives rise to an explosion of the number of possible execution paths between the interacting threads. While pre/postcondition pairs are sufficient to specify sequential components, concurrency introduces additional complexities:

[1] See for example the excellent review in [1].

J. P. Bowen et al. (Eds.): *Theories of Programming and Formal Methods*, LNCS 14080, pp. 110–135, 2023.
https://doi.org/10.1007/978-3-031-40436-8_5

interference: a concurrent thread may modify variables accessed by a component at any point in its execution,

atomicity: programs might be written in high-level programming languages but evaluation of expressions and execution of (assignment) statements at the machine code level cannot be assumed to be atomic with respect to programming language concepts and

termination/progress: termination of operations may be affected by –or even rely on– interference from other threads, and operations may be required to wait for access to shared resources or locks.

Furthermore there are interactions between these issues: the granularity of atomicity affects the extent of the interference (e.g. a data structure controlled by a lock has a coarser granularity of atomicity); and handling progress properties requires an approach to interference that handles possibly non-terminating operations.

As for specifying sequential data structures/types, it is advantageous to make use of an abstract model of the (encapsulated) state of the data structure and make use of a data refinement that introduces a lower-level state for the implementation. For concurrent data structures the choice of representation can affect the manner in which the operations on the data structure may interfere with each other. Often the representation is chosen so that it distinguishes between data and control variables (e.g. locks), where the latter control access to the data structure and are usually of atomic types, whereas the former are typically not of atomic types and rely on the control variables being used to ensure mutual exclusion on the parts of the data being accessed by an operation.

The interference which is characteristic of shared-variable concurrency makes it difficult to achieve a compositional development method. Early concurrency research [3,4,51,52] provided approaches that were neither modular nor compositional – see [38] for more details on these early approaches.

This paper surveys an approach to the development of shared-variable concurrent programs. Specifically it looks back over 40 years of evolution of the rely-guarantee approach including recent research on showing how an algebraic reformulation of the basic idea can provide the key to effective mechanisation of concurrent program development. Section 2 examines the interaction of interference with program assertions, expression evaluation, and assignment commands. Section 3 focusses on abstracting interference by rely conditions and Sect. 4 brings in guarantee conditions to allow concurrent operations to be specified. Section 5 overviews an algebraic approach to specifying and refining concurrent programs. Section 6 looks at specifying atomic operations, which can be used to specify operations on shared data structures and to specify atomic machine operations, such as test-and-set. Section 7 examines termination of loops in the context of interference from parallel threads. Section 8 discusses the role of data abstraction and refinement in the context of concurrency. Section 9 examines operations that may have to wait for resources or locks; such operations may potentially wait forever.

Our overall goal is to provide a concurrent program refinement theory in Isabelle/HOL that supports the derivation and verification of concurrent programs, with all refinement laws used for deriving programs being proven valid within the theory. In particular, we avoid making assumptions about expression evaluation and assignment commands being atomic, and instead make use of laws that show they are effectively indivisible, given certain assumptions.

In this whole research arena, there are important conceptual distinctions and less critical differences in concrete syntax. It is important not to let the latter obfuscate the former and we have tried to tease apart these issues.

Connections with Prof. He Jifeng's Research

Jifeng's work with Tony Hoare on unifying theories of programming [31] has heavily influenced the approach taken to the trace semantics underlying our work, while the algebraic approach to programming derives from the earlier laws of programming [30]. His work on rely/guarantee concurrency [72] has also influenced our approach.

2 Atomicity

An operation executing within a thread is *atomic* if no parallel thread may observe an intermediate state of the operation and the operation cannot observe intermediate states of operations in parallel threads. Some programming language or machine architecture types can be considered atomic, i.e. for a read or write access of a variable of an atomic type, no concurrent thread can observe an intermediate state part way through an access. By contrast, for example, 64-bit integers on a 32-bit architecture do not form an atomic type. For the rest of this paper, we assume that scalar types (such as integers and booleans) are atomic; however, for structured types such as arrays and records, we make no atomicity assumptions about access of the whole structure but assume that access to their sub-components that are of atomic types is atomic. We do not assume that execution of programming language statements –nor evaluation of their conditions– is atomic.

2.1 Program Assertions

Assertions about the state of a program are essential for Floyd/Hoare-style reasoning about programs [21,29] but a program assertion may not be stable under interference from parallel components. An assertion, P, is a set of program states, and a relation, R, is a set of pairs of states, where a program state, σ, can be represented as a mapping from program variable names to their values, i.e. $\sigma\, x$, is the value of variable x in state σ.[2]

[2] The program state may also include a heap but no further discussion of heap store is included below.

Definition 1 (stable). *An assertion P on the program state space is* stable *under a relation R if and only if* $\forall(\sigma, \sigma') \in R \,.\, (\sigma \in P \Rightarrow \sigma' \in P)$.

In the examples we use characteristic predicates for assertions, so that $x > 0$ characterises the set of states $\{\sigma \,.\, \sigma\, x > 0\}$. Similarly, the predicate $x \leq x'$, in which x stands for the initial value of x and x' for the final value of x, characterises the relation $\{(\sigma, \sigma') \,.\, \sigma\, x \leq \sigma'\, x\}$. The assertion $l \geq r$ is not stable under interference that can decrease l or increase r, which is characterised by the predicate $l > l' \vee r < r'$ but $l \geq r$ is stable under interference that can neither decrease l nor increase r, that is, interference satisfying $l \leq l' \wedge r \geq r'$. Note that the interference under which an assertion and its negation are stable may be different. The definition of stable does not require that the program variables referenced within P are unmodified, for example, the assertion, *even* i, is stable under interference that increases i by 2.

2.2 Conditions

A program assertion, P, is used as a judgement about a single program state, σ; either σ is in P or it is not. On the other hand, conditions in if and while commands are evaluated in a context in which the program state may be modified (multiple times) by interference from the environment, and hence are evaluated over a sequence of potentially different states. Each reference to a variable within a condition may access its value in a different state, for example, the condition, $i = i$, may evaluate to false if its two accesses to i are in states with different values of i.

A common restriction [52] is that a condition contains at most one shared variable and at most one reference to that variable, thus ruling out a condition such as $i = i$. With this restriction, evaluating a condition over a sequence of states is equivalent to evaluating it in the single state in which the shared variable is accessed. If a condition satisfies this restriction, it is affected by interference in a similar manner to program assertions. For example, the condition, $i > 0$, in an if command may be true in the state in which i is accessed but it is not stable under the interference that may decrease i, and hence it may no longer be true when the start of the then branch of the if command is reached. However, if $i > 0$ evaluates to false, it will still be false when the start of the else branch is reached if the interference cannot increase i.

2.3 Expressions

As with conditions, evaluation of expressions can lead to anomalies, for example, for an integer variable i, the expression $i + i$ may evaluate to an odd value if i is modified between the two accesses to i, whereas $2 * i$ always evaluates to an even number because there is only a single access to i. Note that it is a valid refinement to replace $i + i$ by $2 * i$ but not vice versa.

We assume the syntax of an expression, e, consists of either a constant, k, a variable, v, a unary operator $\ominus$ applied to an expression, $\ominus e_1$, or a binary

operator $\oplus$ applied to two expressions, $e_1 \oplus e_2$.[3] Evaluation of an expression in a (single) state σ, written e_σ, is defined in the usual manner. If an expression, e, evaluates to the same value before and after interference satisfying a relation R, we say that e is invariant under R.

Definition 2 (invariant-expression). *An expression, e, is* invariant *under a relation R if and only if* $\forall(\sigma, \sigma') \in R \,.\, e_\sigma = e_{\sigma'}$.

For example, the expression, i mod N, is invariant under interference that increments i by the constant N, (i.e. under the relation $\{(\sigma, \sigma') \mid \sigma'\, i = \sigma\, i + N\}$), similarly, $i - i$, is invariant under any interference because evaluating $i - i$ in any single state gives 0. However, evaluating $i - i$ over a sequence of states (as for a programming language expression) under interference that may change i, may not give 0 because $i - i$ has two references to i that may be evaluated in different states to give different values. Expressions that only have a single reference can be reasoned about more easily.

Definition 3 (single-reference). *An expression, e, is* single reference *under a relation R, if and only if e is,*

- *either a constant, k, or an atomic variable, v,*
- *of the form, $\ominus e_1$, and e_1 is single reference under R, or*
- *of the form, $e_1 \oplus e_2$, in which both e_1 and e_2 are single reference under R, and either e_1 or e_2 is invariant under R.*

For example, for integer atomic variables i and j, the expression $(i \text{ mod } 2) + j$, is single reference under interference, R, that may increment i by any multiple of 2 (including 0) and may modify j arbitrarily, because

- i mod 2, is single reference under R because
  - the atomic variable i is single reference under R, and
  - the constant 2 is both single reference and invariant under R,
- i mod 2 is invariant under R, and
- j is single reference under R.

That means the expression, i mod 2, evaluates to the same value, no matter in which state during its evaluation i is accessed, and hence any variance in the value of $(i \text{ mod } 2) + j$, is due to the various values that j can take during the evaluation. If an expression is both single-reference and invariant under R, its evaluation over a sequence of states will return the same value as its evaluation in any of the states.

Other approaches to handling expressions [11,70] assume that the expression has only a single variable, v, that may be modified by the environment and that v is referenced only once. This is a strictly stronger requirement than Definition 3, for example, as shown above $(i \text{ mod } 2)+j$ is single reference under interference that may increment i by a multiple of 2 and arbitrarily update j but it does not satisfy the stricter requirement that only one variable may be modified by interference because both i and j may be modified by the interference.

[3] Conditional "and" and "or" (&& and || in C, Java, etc.) are handled by conditional expressions, which we do not consider here.

2.4 Assignments

An assignment command, $x := e$, in a concurrent context may be subject to interference on variables referenced within e during its evaluation (as in Sect. 2.3) as well as interference that may modify x after it has been assigned. We assume accesses to x and variables within e are atomic.

Owicki [52], Xu et al. [72], Prensa Nieto [56], Stølen [65], Dingel [16], Schellhorn [58] and Sanan [57] treat a complete assignment as atomic, although they do allow interference before and after the assignment. Their reasoning makes use of preconditions and (single-state) postconditions that are stable under interference.

A common observation [7,52] is that, if an assignment only accesses at most a *single shared variable* (i.e. all other variables accessed are unchanged by interference) and there is only one access to that variable, the assignment can be thought of as being atomic—it can be viewed as happening atomically at the (single) point the shared variable is accessed. In an assignment, $x := e$, the single shared variable may be either x or some variable accessed within e (but not both). Hence these approaches commonly impose a syntactic restriction on programs that this property holds for all assignments. Any assignment for which the property does not hold needs to be broken down into a sequence of assignments that do satisfy the property, and which may require fresh local variables. This also introduces additional intermediate assertions and proof obligations.

The validity of the single shared variable approach cannot be proven in the above listed theories, due to their assumption that assignments are atomic. The approach we have taken does not assume assignments are atomic, which is in line with the fact that the (concurrent) semantics of programming language assignments is not atomic. That allows us to show, for example, that the above single shared variable approach is valid, but we can also generalise it to use the more general Definition 3 (single-reference), rather than single shared variable.

3 Interference and Rely Conditions

Pre and rely conditions should inform deployment decisions in that it should be established (preferably by proof) that the context in which an implementation will be deployed will satisfy these assumptions. However, considering the execution of an implementation, semantics must be given to situations where assertions (preconditions, intermediate assertions, loop invariants) of a component are violated by interference from concurrent threads in its environment. For example, consider the following assertion for a set of integers s and integer constant N.

$$s \subseteq \{0 \,..\, N\} \tag{1}$$

The assertion states that all elements of the set s must be in the subrange 0 through N, inclusive. In the context of interference from the environment, the assertion may be invalidated by the environment adding an element outside

$\{0 \,..\, N\}$ to s. However, if the interference only removes elements from s, (1) is *stable*, i.e. if it holds before the interference, it holds after. The interference can be represented by a *rely* condition, in this case the relation characterised by

$$s' \subseteq s \tag{2}$$

in which s is the value of the set before the interference and s' is its value after. By Definition 1 (stable), (1) is stable under the rely condition (2) because

$$s' \subseteq s \vDash (s \subseteq \{0 \,..\, N\} \Rightarrow s' \subseteq \{0 \,..\, N\}).$$

In practice, there may be zero or more steps of interference from the environment, including steps that do not modify s. Such a sequence of environment steps satisfies the reflexive, transitive closure of R, R^*. For example, if each step satisfies $s' \subset s$, any sequence of zero or more steps satisfies $s' \subseteq s$, which is a reflexive and transitive relation. For this reason we consistently use relations that are both reflexive and transitive when abstracting interference as rely condition.

Lemma 1 (stable-many-steps). *If an assertion P is stable under a step of inference that satisfies R, it is stable under interference that satisfies R^*, that is, it is stable under zero or more steps of interference satisfying R.*

4 Rely/Guarantee Thinking

This paper focuses on shared-variable concurrency: reasoning about threads that experience and inflict interference on each other cannot be adequately specified with just pre and post conditions. The idea to add explicit rely and guarantee conditions should again be understood independently of concerns about a concrete syntax for recording specifications.

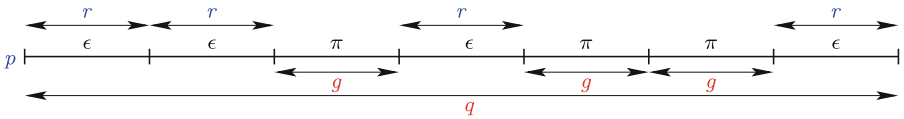

Fig. 1. Execution sequence of program (π) and environment (ϵ) steps, with precondition p, postcondition q, rely condition r and guarantee condition g. If the initial state satisfies p and all environment steps satisfy r, then all program steps must satisfy g and the postcondition q must be satisfied between the initial and final states.

The concept presented by Jones [34–36] can be understood by examining Fig. 1. His approach recovers the crucial property of compositionality by explicitly recording (and offering inference rules for reasoning about) interference. A rely condition r is a binary relation on program states that represents an assumption that any interference steps from the environment of the thread satisfy r between their before and after program states. To complement that, a

thread also has a guarantee condition g, also a relation, that all its program steps must satisfy. A guarantee for a thread must imply the rely conditions of all the threads in its environment.

Consider the example of calculating the prime numbers up to some limit N using a parallel version of the sieve of Eratosthenes. It begins with a set s containing all natural numbers between 2 and N, and uses a set of parallel threads: the first removes all the multiples of 2, the second removes all the multiples of 3, and so on. A basic operation used by all the threads is removing a single element i from s under interference that cannot add elements to s but may remove elements, including i. The standard sequential pre/post specification of *remove*,

$$\mathsf{pre}\ s \subseteq \{0\,..\,N\} \wedge i \in \{0\,..\,N\} \tag{3}$$

$$\mathsf{post}\ s' = s - \{i\} \wedge i' = i \tag{4}$$

is inadequate: while the precondition is stable under interference satisfying the rely condition $s' \subseteq s \wedge i' = i$, the postcondition can be invalidated by interference that removes elements other than i from s. The alternative postcondition, $i \notin s'$, is stable under this interference but it is too weak on its own because it does not preclude the operation adding or removing elements other than i. This can be rectified by including a guarantee condition, $s - s' \subseteq \{i\} \wedge s' \subseteq s \wedge i' = i$, that must be satisfied by every program step made by the implementation of the operation. The specification of the concurrent *remove* operation becomes the following.

$$\mathsf{pre}\ s \subseteq \{0\,..\,N\} \wedge i \in \{0\,..\,N\} \tag{5}$$

$$\mathsf{rely}\ s' \subseteq s \wedge i' = i \tag{6}$$

$$\mathsf{guar}\ s - s' \subseteq \{i\} \wedge s' \subseteq s \wedge i' = i \tag{7}$$

$$\mathsf{post}\ i \notin s' \tag{8}$$

An important property of a postcondition is that it *tolerates* interference satisfying the rely condition, so that the postcondition will not be invalidated by such interference.

Definition 4 (tolerates). *A postcondition relation Q* tolerates *a rely condition R from precondition P, if*

$$\forall \sigma, \sigma' \,.\, \sigma \in P \wedge (\sigma, \sigma') \in (R^* \fatsemi Q \fatsemi R^*) \Rightarrow (\sigma, \sigma') \in Q \tag{9}$$

where $\fatsemi$ is relational composition.

Commonly R is reflexive and transitive, i.e. $R = R^\star$. For the remove operation the rely condition $s' \subseteq s \wedge i' = i$ is both reflexive and transitive, so it is equal to its reflexive transitive closure, and hence its postcondition tolerates its rely condition from its precondition as follows.

$$\begin{array}{l} s \subseteq \{0\,..\,N\} \wedge i \in \{0\,..\,N\} \wedge \\ (\exists s_1, i_1, s_2, , i_2 \,.\, (s_1 \subseteq s \wedge i_1 = i) \wedge i_1 \notin s_2 \wedge (s' \subseteq s_2 \wedge i' = i_2)) \end{array} \vDash i \notin s'$$

A postcondition whose truth can be subverted by an interference step that satisfies the rely condition is problematic because the only valid refinement of a postcondition is to strengthen it (under the assumption of the precondition and $(R \vee G)^\star$ as per the consequence rule 11 below). In the worst case there may be no feasible strengthening.

A postcondition tolerating interference R from a precondition P means that interference before and after the execution of its implementation preserves the postcondition. Interference during the execution is handled during the refinement of the specification.

4.1 Inference Rules

A Jones-style assertion represented in the form,

$$C \ \mathsf{sat}\ (P, R, G, Q) \tag{10}$$

is satisfied if every execution of the command C terminates and satisfies the relation Q end-to-end between its initial and final states provided the initial state satisfies P and every step of the environment of the thread satisfies R; in addition, every atomic program step of C satisfies the relation G between its before and after states provided all the environment steps up until that point have satisfied R and the initial state satisfied P.

There are various presentations in the literature of inference rules for such judgements. The earliest ones in [34] cope with inheriting the rely and guarantee conditions from the context by offering a consequence rule:

$$\boxed{\text{consequence}}\ \frac{\begin{array}{c}(P_2 \vDash P_1) \wedge (R_2 \vDash R_1) \wedge (G_1 \vDash G_2) \\ P_2 \wedge Q_1 \wedge (R_2 \vee G_1)^\star \vDash Q_2 \\ C \ \mathsf{sat}\ (P_1, R_1, G_1, Q_1)\end{array}}{C \ \mathsf{sat}\ (P_2, R_2, G_2, Q_2)} \tag{11}$$

and an inference rule that checks that the rely and guarantee conditions are consistent between sibling threads (shown for clarity with only two threads):

$$\boxed{\text{parallel}}\ \frac{\begin{array}{c}(G_1 \vDash R_2) \wedge (G_2 \vDash R_1) \\ C_1 \ \mathsf{sat}\ (P_1, R_1, G_1, Q_1) \\ C_2 \ \mathsf{sat}\ (P_2, R_2, G_2, Q_2)\end{array}}{C_1 \parallel C_2 \ \mathsf{sat}\ (P_1 \wedge P_2, R_1 \wedge R_2, G_1 \vee G_2, Q_1 \wedge Q_2)} \tag{12}$$

Two points should be noted here:

- The parallel rule handles asymmetric threads; a simpler rule can be given where the threads have specifications that differ only in a parameter.
- The above rule shares with Hoare-like rules for sequential programming constructs the property that they can be read from hypotheses to conclusion to justify a step of decomposition; reading the rule from conclusion to hypotheses facilitates its use to justify top-down compositions.

The soundness of these rules for partial correctness has been shown with respect to an operational semantics of a programming language [11] which allows environment transitions satisfying the rely condition as well as program transitions [10].

The example used in the current paper is a parallel version of the Sieve of Eratosthenes for finding prime numbers. That example uses concurrent instances of threads that differ only by the value of a parameter; more interesting examples of the effectiveness of rely and guarantee conditions come from applications where the concurrent threads differ, which is the case with "Asynchronous Communication Mechanisms" (see [39]) or "On-the-fly Garbage collection" (see [41]).

5 Concurrent Refinement Algebra

The insights to be gained by studying the algebraic properties of programs in general –and concurrent constructs in particular– have been studied in [30,32]; this section provides an algebraic presentation of rely and guarantee conditions. This view also points the way to mechanisation of developments using the rely-guarantee approach.

For the satisfaction relation (10), the 4-tuple (p, r, g, q) can be viewed as a specification for the command c. Dingel's refinement calculus for rely/guarantee concurrency [16] viewed this 4-tuple as a specification command, in a manner similar to Morgan's specification command, $[p, q]$, for the sequential refinement calculus [46,47]. In the sequential refinement calculus it has been recognised that such a specification command can be split into an assertion command $\{p\}$ and a postcondition command $[q]$, so that $[p, q] = \{p\} ; [q]$. As a 4-tuple specification can become cumbersome, especially when some of the components are not relevant, the approach we have taken is to define four commands, $\mathsf{pre}\, p$, $\mathsf{rely}\, r$, $\mathsf{guar}\, g$, and $\mathsf{post}\, q$ for the four components. The command $\mathsf{pre}\, p$ aborts if p does not hold initially, otherwise it allows any non-aborting behaviour. The command $\mathsf{rely}\, r$ aborts if its environment performs a step not satisfying r, otherwise it allows any non-aborting behaviour. The command $\mathsf{guar}\, g$ ensures all program steps satisfy the relation g between their initial and final states. The command $\mathsf{post}\, q$ ensures its initial and final states satisfy q end-to-end; it also terminates (see Sect. 7). The four commands can be combined to form a full specification equivalent to the 4-tuple using the weak conjunction operator $\Cap$, so that the satisfaction relation (10) can be written using the refinement relation $\succcurlyeq$, where for commands c and d, $c \succcurlyeq d$ means c is refined (or implemented) by d,[4]

$$\mathsf{pre}\, p \Cap \mathsf{rely}\, r \Cap \mathsf{guar}\, g \Cap \mathsf{post}\, q \succcurlyeq c \tag{13}$$

[4] In our earlier papers we followed the sequential refinement calculus more closely and used, $\mathsf{rely}\, r \Cap \mathsf{guar}\, g \Cap \{p\} ; \mathsf{post}\, q \succcurlyeq c$, but manipulating the sequential composition of the assertion $\{p\}$ is more complicated than using the conjoined form in (13) because $\Cap$ is an associative, commutative and idempotent operator. Given that $\{p\} ; \mathsf{post}\, q = \mathsf{pre}\, p \Cap \mathsf{post}\, q$, one can switch between the two as necessary. Note that $\{p\}$ terminates immediately if p holds initially, whereas $\mathsf{pre}\, p$ allows any non-aborting behaviour if p holds initially. Both abort if p does not hold initially.

where the weak conjunction of two commands, $c_1 \Cap c_2$, behaves as both c_1 and c_2 up to the point at which either c_1 or c_2 aborts at which point it aborts. For example, weak conjunction satisfies the following.

$$\boxed{\text{Weak conjunction}} \quad \frac{\begin{array}{c} c_1 \ \mathsf{sat}\ (p_1, r_1, g_1, q_1) \\ c_2 \ \mathsf{sat}\ (p_2, r_2, g_2, q_2) \end{array}}{(c_1 \Cap c_2) \ \mathsf{sat}\ (p_1 \cap p_2, r_1 \cap r_2, g_1 \cap g_2, q_1 \cap q_2)} \tag{14}$$

Weak conjunction is monotone in its arguments, i.e. if $c_1 \succcurlyeq d_1$ and $c_2 \succcurlyeq d_2$, then $c_1 \Cap c_2 \succcurlyeq d_1 \Cap d_2$. That allows one to provide a set of simpler refinement laws below that can be combined to give the equivalent of the consequence rule (11), in which the notation $p \lhd r$ stands for the relation r with its domain restricted to the set p.

$$\mathsf{pre}\ p_1 \succcurlyeq \mathsf{pre}\ p_2 \qquad \text{if } p_1 \subseteq p_2 \tag{15}$$

$$\mathsf{rely}\ r_1 \succcurlyeq \mathsf{rely}\ r_2 \qquad \text{if } r_1 \subseteq r_2 \tag{16}$$

$$\mathsf{guar}\ g_1 \succcurlyeq \mathsf{guar}\ g_2 \qquad \text{if } g_2 \subseteq g_1 \tag{17}$$

$$\mathsf{pre}\ p \Cap \mathsf{post}\ q_1 \succcurlyeq \mathsf{pre}\ p \Cap \mathsf{post}\ q_2 \qquad \text{if } p \lhd q_2 \subseteq q_1 \tag{18}$$

$$\mathsf{rely}\ r \Cap \mathsf{guar}\ g \Cap \mathsf{post}\ q_1 \succcurlyeq \mathsf{rely}\ r \Cap \mathsf{guar}\ g \Cap \mathsf{post}\ q_2 \qquad \text{if } (r \cup g)^* \cap q_2 \subseteq q_1 \tag{19}$$

In addition, many refinement rules focus on refining a command c_1 to c_2 in the context of a rely r:

$$\mathsf{rely}\ r \Cap c_1 \succcurlyeq \mathsf{rely}\ r \Cap c_2, \tag{20}$$

where the precondition and guarantee are not relevant.

Some History. It is worth reviewing the path to our current approach. Our earlier approach to handling guarantee commands made use of a command, $(\mathsf{Guar}\ g\ .\ c)$, that restricted the behaviour of the command c so that all program steps satisfy the guarantee relation g. Initially this command was defined directly in terms of an operational semantics [24]. However, in order to define its trace semantics, [25] introduced a *weak conjunction* operator[5] $\Cap$ and defined $(\mathsf{Guar}\ g\ .\ c)$ via a weak conjunction of c with a construct that contained all possible non-aborting traces whose program steps satisfy g. We later realised that it was simpler to just define $\mathsf{guar}\ g$ as a command in its own right and write $\mathsf{guar}\ g \Cap c$ in place of $(\mathsf{Guar}\ g\ .\ c)$. That allowed us to discuss properties of guarantees, (e.g. strengthening and merging guarantees), in isolation, and avoided issues with nesting of guarantee and rely commands, thus leading to a simpler theory.

Our earlier approach also made use of a command, $(\mathsf{Rely}\ r\ .\ c)$, that, if run in parallel with interference satisfying r, implemented c [25]. The effect of this rely command is to strengthen specification c to handle interference satisfying r. The theory using this earlier rely command was complicated because we needed to introduce a refinement relation, $\succcurlyeq_r$, parametrised by a rely condition r, further

[5] Called *strict conjunction* there because it is abort strict, i.e. $c \Cap \mathsf{abort} = \mathsf{abort}$.

parameterise the rely command with the rely condition, z, implicit within c, written ($\mathsf{Rely}\ r\ .\ c_z$), and introduce a predicate, $stops(c, r)$, characterising the set of states from which c terminates under interference satisfying r. Further complications were introduced when nesting guarantees within relies. This earlier approach used a *weak* specification command that only guaranteed to satisfy its postcondition for interference that satisfied the identity relation between states, whereas our newer approach uses a *strong* specification command that achieves its postcondition under any interference and is weakened by weakly conjoining it with the newer, $\mathsf{rely}\, r$, command, in a similar way to a specification being weakened by adding a precondition. Note that in the new theory both the failure of a precondition and the failure of a rely condition are modelled by abort, that means the theory takes a consistent approach to handling assumptions.

In the newer approach, both guarantee and rely commands are defined in terms of other language primitives, and combined with other constructs using weak conjunction, an associative, commutative and idempotent operator. This results in a theory in which it is much easier to manipulate rely/guarantee specifications algebraically and completely avoids the issues with nesting of relies and guarantees with the earlier approach.

6 Specifying Atomic Operations

For a component of a concurrent program, one can distinguish whether a postcondition is to be met by a sequence of state transitions between the initial and final states of the component execution (as in [34]), or whether it needs to be met by what appears to other threads to be an atomic transition, albeit with the possibility of finite stuttering program steps (i.e. steps that do not change the observable program state) before and after (as used by Dingel in his refinement calculus [16]). We use the specification command $\langle q \rangle$ for a command that achieves the postcondition q atomically [28]. The command is allowed to perform a finite number of stuttering program steps before or after the atomic step establishing q and, of course, environment steps may be arbitrarily interleaved between its program steps. The stuttering steps represent program steps that do not modify the observable state (e.g. updates to hidden registers or branch instructions).

Low-level concurrent algorithms, such as those used to implement primitives like locks or message queues, often make use of machine instructions, such as compare-and-swap (CAS) and fetch-and-add (FAA), that are guaranteed to be atomic. Morgan [46,47] defined a specification command, $X :[p, q]$, with X giving the set of variables that may be modified by the command. X is referred to as the *frame* of the command. Here we extend his framing notation to apply to any command, $X : c$, so that execution of c may only modify variables within X.[6] For example, a compare-and-swap (CAS) instruction (21) takes s and (typically local) *old*, *new* and *done* as parameters, it has a frame of s and *done*, and if s equals *old* it succeeds and updates s to *new* otherwise it fails and leaves s unchanged.

[6] The frame is a special form of guarantee that no variables outside X are modified.

The returned boolean value *done* indicates whether the CAS succeeded. An FAA instruction (22) has a frame of the variables x and y, it takes a value k and a variable x whose initial value is fetched and stored in the (local) variable y and then x is updated to $x + k$, all atomically.

$$done \leftarrow \mathsf{CAS}(\mathsf{ref}\ s, old, new) \mathrel{\widehat{=}} s, done : \langle (s = old \Rightarrow s' = new \wedge done') \wedge (s \neq old \Rightarrow s' = s \wedge \neg done') \rangle \quad (21)$$

$$y \leftarrow \mathsf{FAA}(\mathsf{ref}\ x, k) \mathrel{\widehat{=}} x, y : \langle y' = x \wedge x' = x + k \rangle \quad (22)$$

While the instructions take place atomically, they may be preceded and followed by steps taken by their environment that may invalidate the relation.

The other situation in which atomic specifications are useful is for operations on concurrent data structures that are to be implemented using non-blocking algorithms (perhaps utilising atomic instructions such as CAS or FAA). The *remove* operation discussed earlier can also be specified using an atomic specification command.

$$\mathsf{pre}(s \subseteq \{0 \mathinner{..} N\} \wedge i \in \{0 \mathinner{..} N\}) \Cap \mathsf{rely}(s' \subseteq s \wedge i' = i) \Cap s : \langle s' = s - \{i\} \rangle \quad (23)$$

Note that the guarantee (7), $s - s' \subseteq \{i\} \wedge s' \subseteq s \wedge i' = i$, of the earlier specification is implicitly satisfied by the atomic specification: the postcondition ensures both $s - s' \subseteq \{i\}$ and $s' \subseteq s$ and the frame of s ensures $i' = i$.

As another example, a communication channel between two threads may be specified via a queue (qu) of messages sent by one thread but not yet received by the other. The operation to *receive* a message has a precondition that the queue is non-empty and the operation to *send* a message requires that the length of the queue is not at its upper bound N,

$$send(x) \mathrel{\widehat{=}} \mathsf{pre}(\#qu < N) \Cap \mathsf{rely}(qu' \ \mathsf{suffixof}\ qu \wedge x' = x) \Cap qu : \langle qu' = qu \frown [x] \rangle \quad (24)$$

$$x \leftarrow receive \mathrel{\widehat{=}} \mathsf{pre}(qu \neq [\,]) \Cap \mathsf{rely}(qu\ \mathsf{prefixof}\ qu' \wedge x' = x) \Cap qu, x : \langle qu = [x'] \frown qu' \rangle \quad (25)$$

where qu is a sequence of elements, $\#s$ gives the number of items in the sequence s, the operator $\frown$ is sequence concatenation, $[x]$ is the singleton sequence containing x, and $[\,]$ is the empty sequence. This version of send/receive assumes a single sender and a single receiver: multiple senders or receivers would invalidate the rely conditions of send and receive, respectively.

7 Termination

If a thread, T, running concurrently with other threads is never scheduled to execute, the operation running in thread T will never terminate. For example, if the assignment $b := false$ in the parallel composition in (26) is never scheduled, the while loop (and hence the program) will never terminate [53].

$$b := true \,;\, y := 0 \,;\, (b := false \parallel \mathsf{while}\ b\ \mathsf{do}\ y := y + 1) \quad (26)$$

If $b := \mathit{false}$ is scheduled and hence terminates, the whole program completes with the value of y being some (arbitrary) natural number. Hence a basic requirement for termination is that every thread is scheduled with minimal fairness [22].

Rather than building fairness into our primitive parallel operator, we make use of a command, fair, that rules out preemption by its environment forever (i.e. performing an infinite sequence of environment steps) [26]. When fair is conjoined with a command c, their combination,$c \Cap \mathsf{fair}$, represents fair execution of c.

As fairness relates to scheduling threads, we do not build fairness into our encoding of primitive executable code commands, such as an assignment command, that is, such commands do not preclude the environment preempting them forever. Hence when showing "termination" of a command, we show that it takes a finite number of steps, unless it is preempted by its environment forever. We use the command, term, that performs a finite number of (program or environment) steps but may be preempted by its environment forever. If term is conjoined with fair, their combination, $\mathsf{term} \Cap \mathsf{fair}$, can perform only a finite number of steps. To show a command c terminates, we show $\mathsf{term} \succcurlyeq c$, which implies $\mathsf{term} \Cap \mathsf{fair} \succcurlyeq c \Cap \mathsf{fair}$, that is, fair execution of c performs only a finite number of steps. Although our primitive parallel operator ($\|$) does not impose fairness, one can define a fair parallel operator in terms of the primitive parallel and fair [26,27].

A more complex example is (27) in which we assume the decrement and increment of i are atomic. If the two threads alternately decrement and increment i, neither loop terminates but if i manages to get to 0 for the test in the left loop that loop will terminate and hence the other loop will terminate with i being 10 (and similarly if i gets to 10 for the test in the right loop).

$$i := 5 \,;\, (\mathsf{while}\, 0 < i \,\mathsf{do}\, i := i - 1) \parallel (\mathsf{while}\, i < 10 \,\mathsf{do}\, i := i + 1) \tag{27}$$

An important observation coming from this example is that one cannot abstract the interference imposed on the left loop by a *finite* number of increments of i because for any finite number, n, of increments of i, the left loop can iterate $n + 5$ times and terminate.

For partial correctness, one can use the following intuitive algebraic equivalence, similar to that used in Concurrent Kleene Algebra [33] and our earlier approach [23],

$$(\mathsf{Rely}\ r\ .\ \mathsf{post}\ q) \succcurlyeq_p d \iff \mathsf{post}\ q \succcurlyeq_p (d \parallel \langle r \rangle^\star) \tag{28}$$

where $\succcurlyeq_p$ represents partial correctness refinement, $\langle r \rangle^\star$ represents a finite number of iterations of an atomic program step that satisfies r between its before and after states. Hence $d \parallel \langle r \rangle^\star$ represents executing d with a finite number of interference steps satisfying r. If that is partially correct with respect to specification post q, d is partially correct with respect to the specification (Rely r . post q).

Unfortunately, this approach does not extend easily to handling total correctness because, as in the example program (27), the interference from the environment is not guaranteed to be finite. Note that replacing $\langle r \rangle^\star$ with $\langle r \rangle^\omega$, which allows either finite or infinite iteration, means that $d \parallel \langle r \rangle^\omega$ has infinite

behaviour (because $\langle r \rangle^\omega$ does) and hence will not refine $\mathsf{post}\, q$ because $\mathsf{post}\, q$ requires termination.[7]

Interference may affect the stability of a loop's guard or its negation. For example, the following code is an implementation of the remove operation specified by the weak conjunction of (5), (6), (7) and (8), assuming access to s is atomic.

$$\mathsf{while}\, i \in s\, \mathsf{do}(so := s; sn := so - \{i\}; done \leftarrow \mathsf{CAS}(s, so, sn)) \tag{29}$$

The condition $i \in s$ is not stable under interference that can remove elements (including i) from s and hence the body of the loop cannot assume $i \in s$ as its precondition (as it would for a loop in a sequential program). However, the negation of the condition, $i \notin s$, is stable under interference that may remove elements from s and not change i, and hence when the loop terminates it does stably establish $i \notin s$. Note that the flag *done* from the CAS is not used because it may be a concurrent thread that removes i, rather than the CAS, possibly after the CAS but before the test of i in the loop guard.

For a while loop running within a thread, the conventional approach of using a loop variant to show termination may be invalidated if interference can increase the loop variant. However, if the interference never increases the variant, its use to show termination is still valid. For example, the while loop in the code of the remove operation (29) terminates under interference that can only remove elements from s; one can use the set s as the variant expression under the well-founded order of strict finite set inclusion $s \supset s'$ to show termination of the loop because either the CAS succeeds and establishes $i \notin s$ stably or the CAS fails because interference removes some element (possibly i) from s, thus decreasing the loop variant.

8 Data Abstraction and Interference

With sequential programs, employing abstract data types proves to be extremely effective in creating understandable specifications and design histories. Interestingly, most examples of developments using rely-guarantee conditions employ data abstraction and reification (see [37]). With concurrency, there can be the additional bonus that avoidance of data races can be thought out on abstract objects rather than on detailed representations.

The choice of data representation for the implementation can be crucial for reducing the contention/interference between threads. The interference on the abstraction can be completely different to that on its representation. For the sieve example an obvious choice of representation for the set s is as a bitmap. Because N is expected to be larger than the number of bits in a word (say, 64), an array of $\lceil (N+1)/64 \rceil$ words is required, each word representing 64 elements of the set. While interference will still occur when threads are accessing the

[7] To handle termination one may combine (28) with an extra condition to handle termination [25] but that approach becomes quite complicated.

same word in the array, threads accessing separate words will not interfere, in particular, it is not necessary to lock the complete data structure, and updates to the individual words can be done via a compare-and-swap (CAS) instruction, and hence do not require locks.

Simpson's algorithm for achieving an Asynchronous Communication Mechanism [62,63] allows one thread to read the most up to date value of a buffer, d, written by a second thread. Abstractly, the mechanism provides atomic *read* and *write* operations, where d may contain multiple words (i.e. access to d is not atomic (at the hardware level)).

$$r \leftarrow read \mathrel{\widehat{=}} \mathsf{rely}(r' = r) \Cap r : \langle r' = d \rangle \tag{30}$$

$$write(v) \mathrel{\widehat{=}} \mathsf{rely}(v' = v) \Cap d : \langle d' = v \rangle \tag{31}$$

To avoid locking, Simpson's algorithm uses a 2×2 matrix of buffers and carefully arranges their access so that if both threads are active, the slot used by the reading thread differs from that of the writing thread. The slots used are determined by a number of control bits, each of which can be atomically accessed, unlike the buffer itself. Such representations are common for non-blocking algorithms. In the development presented in [39], the number of slots and their arrangement is not determined in the first representation. Not only does this open up a space of alternative representations, it also pinpoints the issues of data races on the slots that are carefully avoided by Hugo Simpson's clever 2×2 organisation. Sorting out race freedom on the intermediate abstraction provides a clear understanding and record of the design.

The message queue from Sect. 6 can be represented by a cyclic buffer of $N+1$ elements (i.e. $buf \in \{0\,..\,N\} \to Value$) plus an index r of the next element to be read and an index w of the next element to be written. If $r = w$ the queue is empty and the queue is full if $(w + 1 = r) \mod (N + 1)$. Note that the queue only stores at most N elements but the size of the buffer is $N + 1$. The extra element allows empty and full queues to be distinguished. The coupling invariant between the queue and its representation can be defined as follows.

$$\begin{aligned} qu = extract(buf, r, w) \ \mathsf{where} \\ extract(buf, r, w) \mathrel{\widehat{=}} \mathsf{if}\ r = w\ \mathsf{then}\ [\,] \\ \mathsf{else}\ (buf\ r) \frown extract(buf, (r + 1) \bmod (N + 1), w) \end{aligned}$$

While access to the elements of buf is not assumed to be atomic, the index variables r and w are assumed to be atomic. The implementations of *send* (24) and *receive* (25) on the representation become the following.

$$\begin{aligned} send(x) &\mathrel{\widehat{=}} buf[w] := x \,;\, w := (w + 1) \bmod (N + 1) \\ x \leftarrow receive &\mathrel{\widehat{=}} x := buf[r]; r := (r + 1) \bmod (N + 1) \end{aligned}$$

In *send*, the assignment to $buf[w]$ does not have to be atomic; it does not become part of the queue until w is updated. While the assignment command updating w is not assumed to be atomic, its store into w is atomic. Similarly, the assignment,

$x := buf[r]$, in *receive* does not have to be atomic because both r and $buf[r]$ are not modified by interference from the sending thread. The update of r removes the first element from the queue. Again the assignment command updating r is not assumed to be atomic but its store into r is atomic. Note that only the sending thread updates w and only the receiving thread updates r. This separation of the control variables that are written by each thread is similar to the way the control variables are used for Simpson's algorithm.

9 Progress

9.1 Waiting for Resources

Termination of an operation that is accessing a shared resource (or lock) requires cooperation from the other threads accessing the resource. Using a variant within a single thread is not applicable to showing that the waiting thread eventually gains control of the resource (or lock). In the simple case the thread with the resource may eventually signal it has finished using the resource and if no other threads are contending for the resource, the waiting thread can acquire the resource and continue, but if the resource is never released, the waiting thread will never make progress. Hence when specifying operations like acquiring a resource or lock, one needs to accommodate the possibility of the operation never terminating if the resource is never released by the thread holding it.

Precondition Versus Termination. For sequential programs, total correctness requires that if the precondition holds initially, the operation terminates. However, for concurrent programs, the precondition holding initially does not ensure an operation will terminate because it may need to wait for a resource and hence its termination is conditional on the behaviour of concurrent threads. For this reason, the identification of the set of initial states from which an operation is guaranteed to terminate with the precondition, as is standard for sequential programs, does not apply to concurrent programs. In fact, it is not in general possible to define a set of initial states that guarantee termination because termination may also be dependent on the interference from concurrent threads.

Deadlock. Waiting for a resource introduces the possibility of a set of threads deadlocking so that all threads are simultaneously waiting, e.g. if thread T_1 requests access to resource A and then resource B while thread T_2 requests access to resource B then resource A, the two threads may deadlock if thread T_1 gets A and then thread T_2 gets B because both threads are then waiting for access to the resource held by the other. As Dijkstra [15] recognised early on, one needs to consider how threads *cooperate* in order to reason about a set of parallel threads.

Starvation. In the more complex case where there is contention on a resource, although there may be no deadlock (i.e. some thread is making progress) it is possible for a thread to be starved if it always loses out to some other thread

every time it tries to acquire the resource. To avoid this issue, access to a resource may be queued (e.g. using a ticket lock rather than a test-and-set lock) so that, provided every thread that acquires the resource eventually releases it, all threads will eventually get access to the resource.

9.2 Conditional Termination

The termination of operations that need to wait for resources is conditional on the behaviour of other threads. For example, the operation to acquire a test-and-set lock may repeatedly attempt to acquire the lock but it may repeatedly miss out because other threads are allocated the lock in preference to it. However, if there are no competing threads trying to acquire the lock, the acquire lock operation is guaranteed to succeed and terminate.

To express the conditional termination of an operation we use an extension of Pnueli's Linear Temporal Logic (LTL) [54]. Each LTL formula f is encoded as a command $\langle\!\langle f \rangle\!\rangle$, whose behaviours are exactly those that satisfy f. For the LTL formula that states that p holds in the initial state, we define an explicit $\iota\, p$ operator, but elide the "ι" within examples so that the notation better matches that of Pnueli.[8] To allow LTL formulae to distinguish program and environment steps, we introduce two new primitives: $\Pi\, r$, that must start with a program step satisfying the relation r, and $\mathcal{E}\, r$, that must start with an environment step satisfying the relation r.

LTL formula f	$\langle\!\langle f \rangle\!\rangle$	$\neg f$
$\iota\, p$	$\tau\, p\, ;\, \boldsymbol{\alpha}^\infty$	$\iota\, \overline{p}$
$\Pi\, r$	$\pi\, r\, ;\, \boldsymbol{\alpha}^\infty$	$\Pi\, \overline{r} \vee \mathcal{E}\, \mathsf{univ}$
$\mathcal{E}\, r$	$\epsilon\, r\, ;\, \boldsymbol{\alpha}^\infty$	$\mathcal{E}\, \overline{r} \vee \Pi\, \mathsf{univ}$
$\Diamond\, f$	$\boldsymbol{\alpha}^\star\, ;\, \langle\!\langle f \rangle\!\rangle$	$\Box\, \neg f$
$\Box\, f$	$\nu x\, .\, (\langle\!\langle f \rangle\!\rangle \wedge (\boldsymbol{\alpha}\, ;\, x))$	$\Diamond\, \neg f$
$f_1 \wedge f_2$	$\langle\!\langle f_1 \rangle\!\rangle \wedge \langle\!\langle f_2 \rangle\!\rangle$	$\neg f_1 \vee \neg f_2$
$f_1 \vee f_2$	$\langle\!\langle f_1 \rangle\!\rangle \vee \langle\!\langle f_2 \rangle\!\rangle$	$\neg f_1 \wedge \neg f_2$

Fig. 2. Encoding LTL formulae as commands

The encodings of the LTL operators are defined in Figure 2,[9] where, $\tau\, p$, represents an instantaneous test that the current state is in the set of states p; $\boldsymbol{\alpha}$ allows any single step, either program or environment and hence $\boldsymbol{\alpha}^\star$ allows any finite sequence of steps and $\boldsymbol{\alpha}^\infty$ allows any infinite sequence of steps; and

[8] Note that p is a predicate on a single state, whereas $\iota\, p$ is an LTL predicate on a trace that holds if and only if p holds in the initial state of the trace. Formalisation in Isabelle/HOL requires an explicit operator, rather than using the type of p as done by Pnueli.

[9] The encoding used here is similar to that used in [12] for a trace semantics but here we encode *true* as $\boldsymbol{\alpha}^\infty$ (rather than abort).

$\nu x \,.\, f(x)$ represents the greatest fixed point of the equation $x = f(x)$. Within temporal logic formulae $\wedge$ and $\vee$ are temporal logic operators, whereas in the encodings of commands they are lattice meet (strong conjunction) and join (non-deterministic choice) of commands, respectively.

The eventually operator, $\Diamond f$, can be thought of as being fair because f is established after a finite number of steps, and hence it disallows preemption by the environment forever. In line with our approach to handling termination, we also define an "unfair" eventually operator $\blacklozenge f$. Similarly, the always operator, $\Box f$, can be thought of as being unfair because it allows preemption by the environment forever, and hence we define a "fair" always operator $\blacksquare f$. The temporal logic formula fair_{LTL} requires that a program step is always eventually taken.[10] Its negation allows preemption by the environment forever.

$$\mathsf{fair}_{LTL} \mathrel{\widehat{=}} \Box\Diamond(\Pi\,\mathsf{univ}) \tag{32}$$

$$\neg\mathsf{fair}_{LTL} = \Diamond\Box(\mathcal{E}\,\mathsf{univ}) \tag{33}$$

$$\blacklozenge f \mathrel{\widehat{=}} \neg\mathsf{fair}_{LTL} \vee \Diamond f \tag{34}$$

$$\blacksquare f \mathrel{\widehat{=}} \mathsf{fair}_{LTL} \wedge \Box f \tag{35}$$

The operators $\blacklozenge$ and $\blacksquare$ are deMorgan duals, that is, $\neg \blacklozenge f = \blacksquare \neg f$.

An operation for a thread with unique identifier, *tid*, to acquire a test-and-set lock can be specified to allow the operation to fail to terminate if the lock is always eventually not free or the operation is preempted forever. The non-terminating alternative behaviour is specified by $\langle\!\langle \Box\blacklozenge(lock \neq free)\rangle\!\rangle$, where $\blacklozenge$ is used rather than $\Diamond$ to allow preemption by the environment forever to lead to non-termination. The non-terminating behaviour does not change the program state and hence it has an empty frame, $\emptyset$.

$$\begin{aligned} &acquire(tid) \mathrel{\widehat{=}} \\ &\quad \mathsf{rely}(lock = tid \Rightarrow lock' = tid) \wedge (lock \neq tid \Rightarrow lock' \neq tid) \wedge tid' = tid \Cap \\ &\quad \{lock \neq tid\} \,;\, (lock : \langle lock = free \wedge lock' = tid\rangle \vee \emptyset : \langle\!\langle \Box\blacklozenge(lock \neq free)\rangle\!\rangle) \end{aligned} \tag{36}$$

The condition under which it is guaranteed to terminate is that eventually the lock is always free, which can be expressed as the linear temporal logic formula $\Diamond\blacksquare(lock = free)$. Note that the operation may terminate even when this condition does not hold because it may successfully acquire the lock (the first alternative of the non-deterministic choice). The specification allows the environment to preempt it forever (because the atomic specification command allows that); fair execution of the operation would rule out that possibility.

A ticket lock has better termination properties than a test-and-set lock because it orders access to the lock. Each thread attempts to acquire a lock by initially taking a ticket (like in a bakery or delicatessen). The tickets are allocated in order of request. The ticket lock tracks the ticket number, *lock*, of the current holder of the lock and when that thread releases the lock, that number

[10] Here, fair_{LTL} equals command fair (from Sect. 7) conjoined with $\boldsymbol{\alpha}^{\infty}$ (our encoding of *true* in these LTL formula).

is incremented so that the next thread in sequence acquires the lock. An acquire operation on a ticket lock is guaranteed to terminate provided every thread that acquires the lock, eventually releases it. The acquire operation consists of two phases: one to take a ticket and the other to wait until its ticket is the one currently being served, i.e. equal to *lock*. Taking a ticket can be specified as an atomic operation that sets the local variable tk to the current (global) counter value ct and increments ct, all atomically,

$$tk \leftarrow take\_ticket \mathrel{\widehat{=}} tk, ct : \langle tk' = ct \wedge ct' = ct + 1 \rangle \tag{37}$$

which can be implemented by a fetch-and-add instruction (22), $tk \leftarrow \mathsf{FAA}(ct, 1)$. The second phase of acquire operation terminates when the current value of *lock* is the thread's ticket, tk, but it may fail to terminate if the lock never corresponds to the thread's ticket, i.e. $\Box(lock \neq tk)$.

$$\begin{aligned} &acquire\_lock(tk) \mathrel{\widehat{=}} \\ &\quad \mathsf{rely}(lock \leq lock' \leq tk \wedge ct \leq ct' \wedge tk' = tk) \Cap \\ &\quad \emptyset : (\lceil lock' = tk \rceil \vee \langle\!\langle \Box(lock \neq tk) \rangle\!\rangle) \end{aligned} \tag{38}$$

Another example is message channel send/receive operations with blocking when the buffer is full/empty, respectively. Note that the preconditions of the previous versions, (24) and (25), become wait conditions.

$$\begin{aligned} send(x) \mathrel{\widehat{=}}\ & \mathsf{rely}(qu' \ \mathsf{suffixof}\ qu \wedge x' = x) \Cap \\ & (qu : \langle \#qu < N \wedge qu' = qu \frown [x] \rangle \vee \emptyset : \langle\!\langle \Box(\#qu = N) \rangle\!\rangle) \end{aligned} \tag{39}$$

$$\begin{aligned} x \leftarrow receive \mathrel{\widehat{=}}\ & \mathsf{rely}(qu \ \mathsf{prefixof}\ qu' \wedge x' = x) \Cap \\ & (x, qu : \langle qu \neq [\,] \wedge qu = [x] \frown qu' \rangle \vee \emptyset : \langle\!\langle \Box(qu = [\,]) \rangle\!\rangle) \end{aligned} \tag{40}$$

Related Work. To handle termination for a blocking await command of the form $\mathsf{await}\ b\ \mathsf{do}\ c$, that waits until condition b holds and atomically with the condition b succeeding executes c, Stølen [64,65] developed a rely/guarantee theory that augments Jones' quintuple with an additional wait condition, w, that characterised the set of states in which a thread can block.[11] The operation to acquire a test-and-set lock (36) is implemented by the command,

$$\mathsf{await}\ lock = free\ \mathsf{do}\ lock := tid.$$

Stølen developed a rule for showing parallel threads do not deadlock: if thread t_1 has a wait condition w_1 and postcondition q_1 and thread t_2 has a wait condition w_2 and postcondition q_2, if $\neg(w_1' \wedge w_2') \wedge \neg(w_1' \wedge q_2) \wedge \neg(w_2' \wedge q_1)$, deadlock is avoided, i.e. both t_1 and t_2 cannot be blocked at the same time, and if t_2 has terminated t_1 cannot be blocked, and if t_1 has terminated t_2 cannot be blocked. For the test-and-set lock example, one can use a wait conditions of $w_1 \mathrel{\widehat{=}} lock \neq t_2$ and $w_2 \mathrel{\widehat{=}} lock \neq t_1$ and noting $\neg(w_1' \wedge w_2') = (lock' = t_2 \vee lock' = t_1)$ meaning that one of the threads has acquired the lock.

[11] Xu Qiwen tackled the same issues in [71] in a similar way.

In the theory we have developed, we can define an await command with a body that is a postcondition relation q to be achieved atomically when b holds; it is guaranteed to terminate if the condition b eventually always holds, or to put it another way, infinite behaviour is allowed if it is always the case that eventually b is false.

$$\mathsf{await}\ b\ \mathsf{do}\ q \mathrel{\widehat{=}} \langle b \lhd q \rangle \vee \emptyset : \langle\!\langle \Box \blacklozenge \neg b \rangle\!\rangle \tag{41}$$

In later work, Stølen [66] gives two semantic interpretations —weak and strong fairness— of an await command. The weak fairness version corresponds to (41) and the strong fairness to the following.

$$\mathsf{await}_{strong}\ b\ \mathsf{do}\ q \mathrel{\widehat{=}} \langle b \lhd q \rangle \vee \emptyset : \langle\!\langle (\blacklozenge \Box \neg b) \rangle\!\rangle \tag{42}$$

Note the swap in the order of $\Box \blacklozenge$ to $\blacklozenge \Box$, meaning that the strong fairness await can only block forever if it is permanently disabled, whereas the weak fairness await can also block forever if b perpetually alternates between true and false.

10 Conclusions

10.1 Summary

The main role of this paper has been to evaluate approaches to specifying concurrent operations or programs in a rely/guarantee style. Because machines include instructions that behave atomically, it makes sense to include a form of specification that represents an atomic operation so that one can specify these instructions within the concurrency theory. Further, atomic specifications in conjunction with preconditions and rely conditions can be used to specify atomic operations on concurrent data structures [16].

The algebraic approaches of Hoare et al. [33], Armstrong et al. [2] and Hayes [18,23] make use of intuitive algebraic properties to handle rely conditions but suffer when it comes to handling termination, nontermination and nesting of constructs. Our initial theory that used a weak specification command forms part of a theory similar to the Concurrent Kleene Algebra of Hoare et al. [33] and the algebraic approaches of Armstrong et al. [2] and Hayes [23]. While this approach works well for handling partial correctness, when the approach is extended to handle total correctness, it becomes considerably more complex.

The combination of the strong specification command weakly conjoined with the (newer) rely command, $\mathsf{rely}\ r$, that weakens the specification to only need achieve its postcondition in contexts where the environment satisfies the rely condition r, supports the specification of always terminating, conditionally terminating and non-terminating threads. That leads to a simpler, more expressive theory than our earlier approach that combined weak specification commands and the rely command, $(\mathsf{Rely}\ r\ .\ c)$, that strengthened the c to handle interference that satisfies r. Because the $\mathsf{rely}\ r$ and $\mathsf{guar}\ g$ commands in the newer theory are weakly conjoined, issues with nesting of older $(\mathsf{Rely}\ r\ .\ c)$ and $(\mathsf{Guar}\ g\ .\ c)$ commands are avoided.

The concurrent refinement algebra approach reported in this paper has been mechanised as a set of Isabelle/HOL theories. The early work on the general algebra is available within the Archive of Formal Proofs [18]. To support data refinement theories for handling localisation [45] and (coupling) invariants have been developed.

10.2 Related Work

There are many developments relating to rely-guarantee ideas that are not covered in the body of this paper; they include:

- local rely/guarantee reasoning is presented in [19];
- in [17] it is argued that deny and guarantee conditions are required to handle fork/join-like concurrency;
- explicit combinations of rely-guarantee thinking with concurrent separation logic (e.g. [50]) are presented in [20,68,69];
- more implicit combinations are used in [8,11,40];
- Barringer, Kuiper and Pnueli [5,6] show the relevance of rely-guarantee ideas to temporal logics;
- Moszkowski's Interval Temporal Logic (ITL) dates from [48,49]; a combination of ITL with rely-guarantee ideas (RGITL) is covered in [58–60,67] and progress aspects are discussed in [61];
- a notion of "Simulation" is developed in [44]; and
- progress and fairness issues are covered in [42,43].

Prenso Nieto [55,56] has developed Isabelle/HOL theories for rely/guarantee concurrency. Her approach assumes condition evaluation and assignment commands are atomic and allows a multiway parallel at the top level but no nested parallel.

A related topic is relaxed memory models used by machine architectures and compilers. For our work, we have assumed that the implementation can be augmented with appropriate fencing to ensure it respects the specification. Coughlin et al. [14] explicitly handle rely and guarantee conditions in the presence of weak memory models.

Acknowledgements. Thanks are due to Joakim von Wright for introducing us to program algebra, and Callum Bannister, Emily Bennett, Robert Colvin, Diego Machado Dias, Chelsea Edmonds, Julian Fell, Matthys Grobbelaar, Oliver Jeaffreson, Patrick Meiring, Tom Manderson, Joshua Morris, Dan Nathan, Katie Deakin-Sharpe, Kim Solin, Andrius Velykis, Kirsten Winter, and our anonymous reviewers for feedback on ideas presented in this paper and/or contributions to the supporting Isabelle/HOL theories. This work is supported by the Australian Research Council https://www.arc.gov.au under their Discovery Program Grant No. DP190102142 and a grant (RPG-2019-020) from the Leverhulme Trust.

References

1. Apt, K.R., Olderog, E.R.: Fifty years of Hoare's logic. Formal Aspects Comput. **31**(6), 751–807 (2019)
2. Armstrong, A., Gomes, V.B.F., Struth, G.: Algebras for program correctness in Isabelle/HOL. In: Höfner, P., Jipsen, P., Kahl, W., Müller, M.E. (eds.) RAMICS 2014. LNCS, vol. 8428, pp. 49–64. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06251-8_4
3. Ashcroft, E.A., Manna, Z.: Formalization of properties of parallel programs. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence, vol. 6, pp. 17–41. Edinburgh University Press (1971)
4. Ashcroft, E.A.: Proving assertions about parallel programs. J. Comput. Syst. Sci. **10**(1), 110–135 (1975)
5. Barringer, H., Kuiper, R.: Hierarchical development of concurrent systems in a temporal logic framework. In: Brookes, S.D., Roscoe, A.W., Winskel, G. (eds.) CONCURRENCY 1984. LNCS, vol. 197, pp. 35–61. Springer, Heidelberg (1985). https://doi.org/10.1007/3-540-15670-4_2
6. Barringer, H., Kuiper, R., Pnueli, A.: Now you may compose temporal logic specifications. In: Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing, STOC 1984, pp. 51–63. Association for Computing Machinery, New York (1984). https://doi.org/10.1145/800057.808665
7. Bernstein, A.J.: Analysis of programs for parallel processing. IEEE Trans. Electron. Comput. **EC-15**(5), 757–763 (1966). https://doi.org/10.1109/PGEC.1966.264565
8. Bornat, R., Amjad, H.: Explanation of two non-blocking shared-variable communication algorithms. Formal Aspects Comput. **25**(6), 893–931 (2013). https://doi.org/10.1007/s00165-011-0213-4
9. Bornat, R., Amjad, H.: Inter-process buffers in separation logic with rely-guarantee. Formal Aspects Comput. **22**(6), 735–772 (2010)
10. Aczel, P.H.G.: On an inference rule for parallel composition (1983). Private communication to Cliff Jones http://homepages.cs.ncl.ac.uk/cliff.jones/publications/MSs/PHGA-traces.pdf
11. Coleman, J.W., Jones, C.B.: A structural proof of the soundness of rely/guarantee rules. J. Log. Comput. **17**(4), 807–841 (2007). https://doi.org/10.1093/logcom/exm030
12. Colvin, R.J., Hayes, I.J., Meinicke, L.A.: Designing a semantic model for a wide-spectrum language with concurrency. Formal Aspects Comput. **29**(5), 853–875 (2017). https://doi.org/10.1007/s00165-017-0416-4
13. Combi, C., Leucker, M., Wolter, F. (eds.): Eighteenth International Symposium on Temporal Representation and Reasoning, TIME 2011, Lübeck, Germany, 12–14 September 2011. IEEE (2011)
14. Coughlin, N., Winter, K., Smith, G.: Rely/guarantee reasoning for multicopy atomic weak memory models. In: Huisman, M., Păsăreanu, C., Zhan, N. (eds.) FM 2021. LNCS, vol. 13047, pp. 292–310. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-90870-6_16
15. Dijkstra, E.: Cooperating sequential processes. In: Genuys, F. (ed.) Programming Languages, pp. 43–112. Academic Press (1968)
16. Dingel, J.: A refinement calculus for shared-variable parallel and distributed programming. Formal Aspects Comput. **14**(2), 123–197 (2002). https://doi.org/10.1007/s001650200032
17. Dodds, M., Feng, X., Parkinson, M., Vafeiadis, V.: Deny-guarantee reasoning. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 363–377. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00590-9_26

18. Fell, J., Hayes, I.J., Velykis, A.: Concurrent refinement algebra and rely quotients. Archive of Formal Proofs (2016). http://isa-afp.org/entries/Concurrent_Ref_Alg.shtml. Formal proof development
19. Feng, X.: Local rely-guarantee reasoning. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, pp. 315–327. ACM, New York (2009). https://doi.org/10.1145/1480881.1480922
20. Feng, X., Ferreira, R., Shao, Z.: On the relationship between concurrent separation logic and assume-guarantee reasoning. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 173–188. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71316-6_13
21. Floyd, R.W.: Assigning meanings to programs. In: Proceedings of Symposia in Applied Mathematics: Mathematics Aspects of Computer Science, vol. 19, pp. 19–32 (1967). https://doi.org/10.1090/psapm/019/0235771
22. van Glabbeek, R., Höfner, P.: Progress, justness, and fairness. ACM Comput. Surv. **52**(4), 1–38 (2019). https://doi.org/10.1145/3329125
23. Hayes, I.J.: Generalised rely-guarantee concurrency: an algebraic foundation. Formal Aspects Comput. **28**(6), 1057–1078 (2016). https://doi.org/10.1007/s00165-016-0384-0
24. Hayes, I.J., Jones, C.B., Colvin, R.J.: Refining rely-guarantee thinking. Technical report CS-TR-1334, Newcastle University (2012)
25. Hayes, I.J., Jones, C.B., Colvin, R.J.: Laws and semantics for rely-guarantee refinement. Technical report CS-TR-1425, Newcastle University (2014)
26. Hayes, I.J., Meinicke, L.A.: Encoding fairness in a synchronous concurrent program algebra. In: Havelund, K., Peleska, J., Roscoe, B., de Vink, E. (eds.) FM 2018. LNCS, vol. 10951, pp. 222–239. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-95582-7_13
27. Hayes, I.J., Meinicke, L.A.: Encoding fairness in a synchronous concurrent program algebra: extended version with proofs (2018). arXiv:1805.01681 [cs.LO]
28. Hayes, I.J., Meinicke, L.A., Meiring, P.A.: Deriving laws for developing concurrent programs in a rely-guarantee style (2021). https://doi.org/10.48550/ARXIV.2103.15292, https://arxiv.org/abs/2103.15292
29. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–580, 583 (1969). https://doi.org/10.1145/363235.363259
30. Hoare, C.A.R., et al.: Laws of programming. Commun. ACM **30**(8), 672–686 (1987). Corrigenda: CACM 30(9):770
31. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Series in Computer Science, Prentice Hall, London (1998)
32. Hoare, C.A.R., Möller, B., Struth, G., Wehrman, I.: Concurrent Kleene algebra. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 399–414. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04081-8_27
33. Hoare, C.A.R., Möller, B., Struth, G., Wehrman, I.: Concurrent Kleene algebra and its foundations. J. Log. Algebr. Program. **80**(6), 266–296 (2011). https://doi.org/10.1016/j.jlap.2011.04.005
34. Jones, C.B.: Development methods for computer programs including a notion of interference. Ph.D. thesis, Oxford University (1981). Available as: Oxford University Computing Laboratory (now Computer Science) Technical Monograph PRG-25
35. Jones, C.B.: Specification and design of (parallel) programs. In: Proceedings of IFIP 1983, pp. 321–332. North-Holland (1983)

36. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM ToPLaS **5**(4), 596–619 (1983). https://doi.org/10.1145/69575.69577
37. Jones, C.B.: Splitting atoms safely. Theoret. Comput. Sci. **375**(1–3), 109–119 (2007). https://doi.org/10.1016/j.tcs.2006.12.029
38. Jones, C.B.: Three early formal approaches to the verification of concurrent programs. Mind. Mach. (2023). https://doi.org/10.1007/s11023-023-09621-5
39. Jones, C.B., Hayes, I.J.: Possible values: exploring a concept for concurrency. J. Log. Algebraic Methods Program. **85**(5, Part 2), 972–984 (2016). https://doi.org/10.1016/j.jlamp.2016.01.002
40. Jones, C.B., Yatapanage, N.: Reasoning about separation using abstraction and reification. In: Calinescu, R., Rumpe, B. (eds.) SEFM 2015. LNCS, vol. 9276, pp. 3–19. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22969-0_1
41. Jones, C.B., Yatapanage, N.: Investigating the limits of rely/guarantee relations based on a concurrent garbage collector example. Formal Aspects Comput. **31**(3), 353–374 (2019). https://doi.org/10.1007/s00165-019-00482-3
42. Liang, H., Feng, X.: A program logic for concurrent objects under fair scheduling. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, pp. 385–399. ACM, New York (2016). https://doi.org/10.1145/2837614.2837635
43. Liang, H., Feng, X.: Progress of concurrent objects with partial methods. Proc. ACM Program. Lang. **2**(POPL), 20:1–20:31 (2018). https://doi.org/10.1145/3158108
44. Liang, H., Feng, X., Fu, M.: Rely-guarantee-based simulation for compositional verification of concurrent program transformations. ACM Trans. Program. Lang. Syst. **36**(1), 3:1–3:55 (2014)
45. Meinicke, L.A., Hayes, I.J.: Using cylindric algebra to support local variables in rely/guarantee concurrency. In: 2023 IEEE/ACM 11th International Conference on Formal Methods in Software Engineering (FormaliSE), 108–119 (2023). IEEE
46. Morgan, C.C.: The specification statement. ACM Trans. Prog. Lang. Syst. **10**(3), 403–419 (1988). https://doi.org/10.1145/44501.44503
47. Morgan, C.C.: Programming from Specifications, 2nd edn. Prentice Hall, London (1994)
48. Moszkowski, B.C.: Executing Temporal Logic Programs. Cambridge University Press, Cambridge (1986)
49. Moszkowski, B.: Executing temporal logic programs. In: Brookes, S.D., Roscoe, A.W., Winskel, G. (eds.) CONCURRENCY 1984. LNCS, vol. 197, pp. 111–130. Springer, Heidelberg (1985). https://doi.org/10.1007/3-540-15670-4_6
50. O'Hearn, P.W.: Resources, concurrency and local reasoning. Theoret. Comput. Sci. **375**(1–3), 271–307 (2007). https://doi.org/10.1016/j.tcs.2006.12.035
51. Owicki, S.: Axiomatic proof techniques for parallel programs. Ph.D. thesis, Department of Computer Science, Cornell University (1975)
52. Owicki, S.S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Inform. **6**(4), 319–340 (1976). https://doi.org/10.1007/BF00268134
53. Park, D.: On the semantics of fair parallelism. In: Bjøorner, D. (ed.) Abstract Software Specifications. LNCS, vol. 86, pp. 504–526. Springer, Heidelberg (1980). https://doi.org/10.1007/3-540-10007-5_47
54. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, pp. 46–57. IEEE (1977)
55. Prensa Nieto, L.: Verification of parallel programs with the Owicki-Gries and rely-guarantee methods in Isabelle/HOL. Ph.D. thesis, Institut für Informatic der Technischen Universitaet München (2001)

56. Nieto, L.P.: The rely-guarantee method in Isabelle/HOL. In: Degano, P. (ed.) ESOP 2003. LNCS, vol. 2618, pp. 348–362. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36575-3_24
57. Sanan, D., Zhao, Y., Lin, S.W., Yang, L.: CSim^2: compositional top-down verification of concurrent systems using rely-guarantee. ACM Trans. Program. Lang. Syst. **43**(1), 1–46 (2021)
58. Schellhorn, G., Tofan, B., Ernst, G., Pfähler, J., Reif, W.: RGITL: a temporal logic framework for compositional reasoning about interleaved programs. Ann. Math. Artif. Intell. **71**(1–3), 131–174 (2014). https://doi.org/10.1007/s10472-013-9389-z
59. Schellhorn, G., Tofan, B., Ernst, G., Reif, W.: Interleaved programs and rely-guarantee reasoning with ITL. In: Combi et al. [13], pp. 99–106 (2011). https://doi.org/10.1109/TIME.2011.12
60. Schellhorn, G.: Extending ITL with interleaved programs for interactive verification. In: Combi et al. [13] (2011). https://doi.org/10.1109/TIME.2011.31
61. Schellhorn, G., Travkin, O., Wehrheim, H.: Towards a thread-local proof technique for starvation freedom. In: Ábrahám, E., Huisman, M. (eds.) IFM 2016. LNCS, vol. 9681, pp. 193–209. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33693-0_13
62. Simpson, H.R.: Four-slot fully asynchronous communication mechanism. Comput. Digit. Tech. IEE Proc. E **137**(1), 17–30 (1990)
63. Simpson, H.R.: New algorithms for asynchronous communication. IEE Proc. Comput. Digit. Technol. **144**(4), 227–231 (1997)
64. Stølen, K.: Development of parallel programs on shared data-structures. Ph.D. thesis, Manchester University (1990). Available as UMCS-91-1-1 or revised version as https://breibakk.no/kst/PhD-thesis.htm
65. Stølen, K.: A method for the development of totally correct shared-state parallel programs. In: Baeten, J.C.M., Groote, J.F. (eds.) CONCUR 1991. LNCS, vol. 527, pp. 510–525. Springer, Heidelberg (1991). https://doi.org/10.1007/3-540-54430-5_110
66. Stølen, K.: Shared-state design modulo weak and strong process fairness. In: Diaz, M., Groz, R. (eds.) Formal Description Techniques, V, Proceedings of the IFIP TC6/WG6.1 Fifth International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, FORTE 1992, Perros-Guirec, France, 13–16 October 1992. IFIP Transactions, vol. C-10, pp. 479–498. North-Holland (1992)
67. Tofan, B., Schellhorn, G., Ernst, G., Pfähler, J., Reif, W.: Compositional verification of a lock-free stack with RGITL. In: Proceedings of International Workshop on Automated Verification of Critical Systems, Electronic Communications of EASST, vol. 66, pp. 1–15 (2013)
68. Vafeiadis, V.: Modular fine-grained concurrency verification. Ph.D. thesis, University of Cambridge (2007)
69. Vafeiadis, V., Parkinson, M.: A Marriage of rely/guarantee and separation logic. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 256–271. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74407-8_18
70. Wickerson, J., Dodds, M., Parkinson, M.: Explicit stabilisation for modular rely-guarantee reasoning. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 610–629. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11957-6_32
71. Xu, Q.: A theory of state-based parallel programming. Ph.D. thesis, Oxford University (1992)
72. Xu, Q., de Roever, W.P., He, J.: The rely-guarantee method for verifying concurrent programs. Formal Aspects Comput. **9**, 149–174 (1997)