

# A Marriage of Rely/Guarantee and Separation Logic

Viktor Vafeiadis and Matthew Parkinson

University of Cambridge, The Computer Laboratory

November 1, 2006

**Abstract.** In the quest for modular reasoning about shared-variable concurrent algorithms, two alternatives have emerged: rely/guarantee logic and separation logic. The former is very good at describing interference, the latter at reasoning about resource management and memory-allocated data structures.

We propose a combined system which marries the two approaches. We can describe interference naturally (using a relation as in rely/guarantee), and where there is no interference, we can reason locally (as in separation logic). We demonstrate the advantages of the combined approach with an example of verifying a lock-coupling list algorithm, which actually disposes (frees) deleted nodes.

## 1 Introduction

Reasoning about shared variable concurrent programs is difficult, because the interference between the simultaneously executing threads must be taken into account. Our aim is to find methods that allow this reasoning to be done in a modular and composable way.

Rely/guarantee provides a good way of describing interference by having two relations  $R$  and  $G$  which describe the state changes performed by the environment or by the program respectively. This approach, initially suggested by Jones [10], has been quite popular, and has been used both in the derivation and the post-hoc verification of concurrent algorithms. A disadvantage of RG is that the specification of interference is global: it must be checked against every state update, even if it is ‘obvious’ that the update cannot interfere with anything else.

On the other hand, the recent development of separation logic [16, 12] suggests that greater modularity is possible. The  $*$  operator and the frame rule allow one to carve out of the specification all the irrelevant state, and focus only on the state that matters for the execution of a certain component or thread. Initially, separation logic was proposed as a logic dealing with heap mutating programs, but was then extended to also deal with parallel programs. In dealing with concurrent programs, it uses invariants to express thread interaction. This makes expressing the temporal nature of interference often quite difficult and requires many auxiliary variables. Its advantage, however, is that its invariants are *localised*: only those statements acting on the shared state need to be

checked that they preserve the invariant; other statements that just do some local computation need not be considered in the preservation proof.

We propose a certain combination of rely/guarantee and separation logic, which combines their advantages and eliminates some of their weaknesses.

We split the state into two disjoint parts: (i) the shared state which is accessible by all threads, and (ii) the local state which is accessible by a single component. Then, we use rely/guarantee to deal with the shared state, and separation logic to deal with the local state. This is best illustrated by our parallel composition rule:

$$\frac{\begin{array}{l} c_1 \text{ sat } (p_1, R_1, G_1, q_1) \quad (R \vee G_1) \Longrightarrow R_2 \\ c_2 \text{ sat } (p_2, R_2, G_2, q_2) \quad (R \vee G_2) \Longrightarrow R_1 \quad (G_1 \vee G_2) \Longrightarrow G \end{array}}{c_1 \parallel c_2 \text{ sat } (p_1 * p_2, R, G, q_1 * q_2)}$$

This rule is identical to the standard rely/guarantee rule except for the use of  $*$  instead of  $\wedge$  in the conclusion. In our specifications, the preconditions (e.g.  $p_1$ ) and the postconditions (e.g.  $q_1$ ) describe both the local and the shared state. The rely conditions (e.g.  $R_1$ ) and the guarantee conditions (e.g.  $G_1$ ) describe inter-thread interference, namely how the shared state gets modified.

The separating conjunction between assertions about both the local and the shared state splits local state in two parts, but does not divide the shared state.

$$(p_1 * p_2)(local, shared) \stackrel{\text{def}}{=} \exists l_1 l_2. local = l_1 \uplus l_2 \wedge p_1(l_1, shared) \wedge p_2(l_2, shared)$$

The parallel composition rules of rely/guarantee and separation logic are special cases of our parallel composition rule.

- When the local state is empty, then  $p_1 * p_2 = p_1 \wedge p_2$  and we get the standard rely/guarantee rule.
- When the shared state is empty, we do not need to describe its evolution ( $R$  and  $G$  are the identity relation). Then  $p_1 * p_2$  has the same meaning as separation logic  $*$ , and we get the parallel rule of concurrent separation logic without resource invariants (see Section 2.2).

An important aspect of our approach is that the boundaries between the local state and the shared state are not fixed, but may change as the program runs. This may be seen as an instance of the “ownership transfer” concept, which is heavily advocated by concurrent separation logic.

In addition, as we encompass separation logic, we can cleanly reason about dynamically allocated data structures and explicit memory management, avoiding the need to rely on a garbage-collector. We demonstrate this by verifying a lock-coupling list algorithm, which actually dispose (frees) deleted nodes. (see Section 4)

## 2 Technical background

In this paper, we reason about a simple parallel programming language with pointer operations. Commands  $c$  are given by the following grammar,

$$c ::= A \mid c_1; c_2 \mid c_1 \parallel c_2 \mid \text{if } b \text{ then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c \mid \text{await } b \text{ do } c$$

where  $b$  ranges over boolean expressions and  $A$  over simple commands (assignments, memory reads/writes, `dispose`  $e$ , etc.). In the rest of this section, we give a brief overview of the two logics we build on in this paper.

## 2.1 Interference reasoning – Rely/guarantee specifications

*Rely/guarantee* specifications [10] describe the interference between concurrently executing threads. These specifications are then used to prove concurrent algorithms in a compositional manner. Each component  $c$  is assigned a *rely* condition that describes the interference it can tolerate from its environment (namely, the other components of the system). In return, it is assigned a *guarantee* condition that characterises how it can interfere with the others.

The essence of rely/guarantee reasoning is its parallel composition rule. Two components (threads) may be placed in parallel, if and only if, the guarantee condition of the one component implies the rely condition of the other and vice versa.

$$\frac{c_1 \text{ sat } (R_1, G_1) \quad (R \vee G_1) \implies R_2 \quad c_2 \text{ sat } (R_2, G_2) \quad (R \vee G_2) \implies R_1 \quad (G_1 \vee G_2) \implies G}{c_1 \parallel c_2 \text{ sat } (R, G)}$$

Since the interference experienced by thread  $c_1$  can arise from  $c_2$  or the environment of the parallel composition, we have to ensure that the total interference ( $R \vee G_2$ ) is allowed by  $R_1$ . Similarly  $c_2$  must be able to tolerate interference from  $c_1$  and from the environment of the parallel composition. The interference caused by the parallel composition may be caused by either  $c_1$  or  $c_2$ ; so, the total interference  $G$  must include the interferences  $G_1$  and  $G_2$  caused by each component.

## 2.2 Local reasoning – Separation logic

In Hoare logic [8], assertions describe properties of the *whole* memory, and hence specifications, e.g.  $\{P\} - \{Q\}$ , describe a change of the whole memory. This is inherently *global reasoning*. Anything that is not explicitly preserved in the specification could be changed, for example  $\{x = 4\} \text{ y} := 5 \{x = 4\}$ . Here  $y$  is allowed to change, even though it is not mentioned in the specification.<sup>1</sup>

The situation is different in *separation logic* [16]. Assertions describe properties of *part* of the memory, and hence specifications describe changes to *part* of the memory. The rest of the memory is guaranteed to be unchanged. This is the essence of *local reasoning*, specifications describe only the memory used by a command, its footprint.

The strength of separation logic comes from a new logical connective: the separating conjunction,  $*$ .  $P * Q$  asserts the state can be separated into two parts, one described by  $P$  and the other by  $Q$ . The separating conjunction allows us to formally capture the essence of *local reasoning* with following rules:

<sup>1</sup> ‘Modifies clauses’ attempt to fix this problem, but they are neither pretty nor general.

```

{ArrSegbnd(a, first, last, min, max)}
qsort(a, first, last) {
  local pivot;
  if(first < last - 1) {
    pivot = partition(a, first, last);
    qsort(a, first, pivot); || qsort(a, pivot, last);
  }
}
{ArrSegsrt(a, first, last, min, max)}

```

**Fig. 1.** Parallel Quicksort algorithm and specification

$$\frac{\{P\} C \{Q\}}{\{P * R\} C \{Q * R\}} \text{ (Frame)} \qquad \frac{\{P_1\} C_1 \{Q_1\} \quad \{P_2\} C_2 \{Q_2\}}{\{P_1 * P_2\} C_1 || C_2 \{Q_1 * Q_2\}} \text{ (Par)}$$

The first rule says, if  $P$  is separate from  $R$ , and  $C$  transforms  $P$  into  $Q$  then if  $C$  finishes we have  $Q$  and separately still have  $R$ . The second rule says that if two threads have disjoint memory requirements, they can execute safely in parallel, and the post-condition is simply the composition of the two threads' post-conditions.<sup>2</sup>

**Example: Parallel Quicksort** To motivate the use of separation logic, we verify parallel quicksort. Parallel quicksort uses disjoint concurrency, hence it is well suited to a separation logic proof: there is no interference.

We present the algorithm and specification in Figure 1. The pre-condition,  $\text{ArrSeg}_{\text{bnd}}(\mathbf{a}, \text{first}, \text{last}, \text{min}, \text{max})$ , means the heap contains a segment of array  $\mathbf{a}$ , from index  $\text{first}$  to  $\text{last} - 1$ , with values in the interval  $[\text{min}, \text{max}]$ ; and the post-condition,  $\text{ArrSeg}_{\text{srt}}(\mathbf{a}, \text{first}, \text{last}, \text{min}, \text{max})$ , denotes that this array segment is sorted. For simplicity, we omit saying that it is a permutation of the initial array segment. We specify the `partition` function as follows, but omit the source code and proof.

```

{ArrSegbnd(a, first, last, min, max)}
pivot = partition(a, first, last)
{∃X. ArrSegbnd(a, first, pivot, min, X) * ArrSegbnd(a, pivot, last, X, max)}

```

The post-condition specifies the two segments are disjoint, hence, we can sort the two segments in parallel without interference.

```

{ArrSegbnd(a, first, pivot, min, X) * ArrSegbnd(a, pivot, last, X, max)}
  qsort(a, first, pivot); || qsort(a, pivot, last);
{ArrSegsrt(a, first, pivot, min, X) * ArrSegsrt(a, pivot, last, X, max)}

```

<sup>2</sup> Originally, separation logic did not consider global variables as resource; hence the proof rules had nasty side-conditions. Later, this problem was solved by Bornat et al. [3]. By prohibiting assignments to global variables, we finesse the issue.

To verify this algorithm with rely/guarantee, we would need to express that each parallel call modified disjoint elements of array. That is,

$$\begin{aligned} \text{guarantee: } & \forall i. (\mathbf{first} \leq i < \mathbf{last}) \vee \mathbf{a}[i] = \mathit{old}(\mathbf{a}[i]) \\ \text{rely: } & \forall i. (\mathbf{first} \leq i < \mathbf{last}) \Rightarrow \mathbf{a}[i] = \mathit{old}(\mathbf{a}[i]) \end{aligned}$$

In separation logic, local reasoning deals with this interference.

**Brief details** We sketch out separation logic [16], which is used to reason about heap-manipulating programs. Let  $x, y$  and  $z$  range over logical variables, and  $\mathbf{x}, \mathbf{y}$  and  $\mathbf{z}$  over program variables. We define expressions by

$$e ::= x \mid \mathbf{x} \mid e + e \mid n$$

Expressions in separation logic are *pure*: they cannot mention the heap. To describe the heap, we have two atomic predicates:

- $\mathit{emp}$  is an empty heap;
- $e \mapsto e'$  is a heap of one cell with address  $e$  and contents  $e'$ .

We build heap descriptions of multiple cell heaps using  $*$  and  $\mapsto$ . For example,  $e \mapsto e' * f \mapsto f'$  describes two *separate* heap cells: it is impossible that  $e$  and  $f$  could be the same address (logically, if  $e$  and  $f$  are equal, then  $e \mapsto f' * e \mapsto f'$  is false). For compactness, we use two shorthands: (1)  $e \mapsto \_$  means  $\exists x. e \mapsto x$ ; and (2)  $e \mapsto e_1, \dots, e_n$  means  $(e \mapsto e_1) * \dots * (e + n - 1 \mapsto e_n)$ .

Assignment to local variables are treated by the ordinary Hoare axiom,  $\{Q[e/\mathbf{x}]\} \mathbf{x} := e \{Q\}$ , where  $Q[e/\mathbf{x}]$  substitutes  $e$  for all occurrences of  $\mathbf{x}$  in  $Q$ . The other axioms of separation logic are summarised below.

$$\begin{aligned} & \{e \mapsto \_ \} [e] := e' \{e \mapsto e'\} \\ & \{e = y \wedge e \mapsto z\} \mathbf{x} := [e] \{y \mapsto z \wedge \mathbf{x} = z\} \\ & \{\mathit{emp}\} \mathbf{x} := \mathbf{cons}(e_1, \dots, e_n) \{\mathbf{x} \mapsto e_1, \dots, e_n\} \\ & \{e \mapsto \_ \} \mathbf{dispose}(e) \{\mathit{emp}\} \end{aligned}$$

(These are known as the small axioms, because they deal with the smallest heap affected by command. If there is more heap present, the frame rule says that it remains unaffected.)

- To write to a heap cell that cell must exist in the heap: i.e. you must own it.
- To read a cell  $[e]$  you must own the cell; its contents are copied into variable  $\mathbf{x}$ ; the cell's contents are unchanged; and afterwards you still own it. (The logical variable  $y$  is used in case  $\mathbf{x}$  occurs in  $e$ .)
- $\mathbf{cons}(e_1, \dots, e_n)$  allocates a new block of  $n$  heap cells. We require the heap is initially empty, and the post-condition contains the new block of cells.
- $\mathbf{dispose}(e)$  deallocates a heap cell. We require the heap contains the cell being disposed; after disposal it is no longer contained in the heap.

To summarise, the syntax of separation logic assertions is:

$$P, Q ::= \text{false} \mid P \Rightarrow Q \mid P * Q \mid P \text{-}\ast^{\exists} Q \mid \text{emp} \mid e = e \mid e \mapsto e$$

We encode  $\neg, \wedge, \vee$ , and *true* in the classical way. There remains one component to describe,  $P \text{-}\ast^{\exists} Q$ , the existential magic wand.<sup>3</sup> Intuitively,  $P \text{-}\ast^{\exists} Q$  represents removing  $P$  from  $Q$ . Formally, it means the heap can be extended with a state satisfying  $P$ , and the extended state satisfies  $Q$ .

### 3 The combined logic

#### 3.1 Local and shared state assertions

Our logic uses rely/guarantee to deal with the shared state, and separation logic to deal with the local state. We need our assertions to describe both local and shared states.

We could specify a state using two assertions, one describing the local state and the other the shared state. However, this approach has some significant drawbacks: specifications would be longer, and it is hard to extend to a setting where there are multiple disjoint regions of shared state.

Instead, we consider a unified assertion language that describes both the local and the shared state. This is done by extending the assertion language with ‘boxed’ terms. We could use boxes for both local and shared assertions: for example,  $\boxed{P}_{\text{local}}$  and  $\boxed{P}_{\text{shared}}$ . However, since  $\boxed{P}_{\text{local}} * \boxed{Q}_{\text{local}} \iff \boxed{P * Q}_{\text{local}}$  holds for  $*$ , and all the classical operators, we can omit the  $\boxed{\phantom{P}}_{\text{local}}$  and the “shared” subscript. Hence the syntax of assertion is

$$p, q, r ::= P \mid \boxed{P} \mid p * q \mid \text{false} \mid p \Rightarrow q \mid \exists x. p$$

Semantically, we split the state,  $\sigma$ , of the system into two components: the local state  $l$ , and the shared state  $s$ . Each component state may be thought to be a partial finite function from locations to values. We require that the domains of the two states are disjoint, so that the total state is simply the (disjoint) union of the two states. Assertions without boxes describe purely the local state  $l$ , whereas a boxed assertion  $\boxed{P}$  describes the shared state  $s$ . Formally the semantics of assertions are given below:

$$\begin{aligned} P(s, l) & \stackrel{\text{def}}{=} P(l) \\ \boxed{P}(s, l) & \stackrel{\text{def}}{=} \text{dom}(l) = \emptyset \wedge P(s) \\ (p_1 * p_2)(s, l) & \stackrel{\text{def}}{=} \exists l_1, l_2. (l = l_1 \uplus l_2) \wedge p_1(l_1, s) \wedge p_2(l_2, s) \end{aligned}$$

Note that  $*$  is multiplicative over the local state, but additive over the shared state. Hence,  $\boxed{P} * \boxed{Q} \iff \boxed{P \wedge Q}$ .

<sup>3</sup> Normally, separation logic papers mention  $\text{-}\ast$ , which uses universal quantification in its definition, we can define the existential magic wand in terms of the standard one:

$$P \text{-}\ast^{\exists} Q \stackrel{\text{def}}{=} \neg(P \text{-}\ast \neg Q).$$

The semantics of shared assertions,  $\boxed{P}$ , could alternatively be presented without  $\text{dom}(l) = \emptyset$ . This results in an equally expressive logic, but the definition above leads to shorter assertions in practice.

### 3.2 Describing interference

The strength of rely/guarantee is the careful description of interference between parallel processes. We describe interference in terms of actions  $P \rightsquigarrow Q$  which describe the changes performed to the shared state. These resemble Morgan’s *specification statements* [11], and  $P$  and  $Q$  will typically be linked with some existentially quantified logical variables. (We do not need to mention separately the set of modified shared locations, because these are all included in  $P$ .) The meaning of an action  $P \rightsquigarrow Q$  is that it replaces the part of the shared state that satisfies  $P$  prior to the action with a part satisfying  $Q$ .<sup>4</sup> Consider the following action:

$$\mathbf{x} \mapsto M \rightsquigarrow \mathbf{x} \mapsto N \wedge N \geq M \quad (\text{Increment})$$

It specifies that the value in the heap cell is never decreased. In this action, the pre-condition and post-condition have the same heap footprint. If the domains are different, this indicates a transfer of ownership between the shared state and the local state of a thread. Consider a simple lock with two operations: (Acquire) which changes the lock bit from 0 to 1, and removes the protected object,  $P$ , from the shared state; and (Release) which changes the lock bit from 1 to 0, and replaces the protected object into the shared state. We can represent these two operations formally as

$$\begin{aligned} (\mathbf{x} \mapsto 0) * P \rightsquigarrow \mathbf{x} \mapsto 1 & \quad (\text{Acquire}) \\ \mathbf{x} \mapsto 1 \rightsquigarrow (\mathbf{x} \mapsto 0) * P & \quad (\text{Release}) \end{aligned}$$

An action  $P \rightsquigarrow Q$  represents the modification of some shared state satisfying  $P$  to some state satisfying  $Q$ . Its semantics is the following relation:

$$\llbracket P \rightsquigarrow Q \rrbracket = \{(s_1 * s_0, s_2 * s_0) \mid P(s_1) \wedge Q(s_2)\}$$

It relates some initial shared state  $s_1$  satisfying the precondition  $P$  to a final state  $s_2$  satisfying the postcondition. In addition, there may be some disjoint shared state  $s_0$  which is not affected by the action. In the spirit of separation logic, we want the action specification as ‘small’ as possible, describing  $s_1$  and  $s_2$  but not  $s_0$ , and use the frame rule to perform the same update on a larger state.

<sup>4</sup> In concurrent separation logic invariants must be *precise* [12], otherwise the logic is unsound. Similarly, we require our actions,  $P \rightsquigarrow Q$ , to be precise, that is  $P$  and  $Q$  to be *precise*. An assertion is precise iff  $(P * \text{true})(l)$ , then there exists unique  $l'$  such that  $P(l')$  and  $l' \subseteq l$ .

In our approach, the rely and guarantee conditions are simply sets of actions. Their semantics as a relation is the reflexive and transitive closure of the union of the semantics of each action in the set.

$$\llbracket P_1 \rightsquigarrow Q_1, \dots, P_n \rightsquigarrow Q_n \rrbracket = (\llbracket P_1 \rightsquigarrow Q_1 \rrbracket \cup \dots \cup \llbracket P_n \rightsquigarrow Q_n \rrbracket)^*$$

### 3.3 Preservation

Rely/guarantee requires that every pre- and post-condition in a proof is stable under the interference of the environment. By construction, any assertion about the local state of a component is unaffected by the interference from other components, as interference can occur only on the shared state. On the other hand, a boxed assertion  $\boxed{S}$  may be affected.

We say that an assertion  $\boxed{S}$  is stable under interference of an action  $P \rightsquigarrow Q$ , iff, whenever  $S$  holds in the initial (shared) state and a (shared) state update satisfying  $\llbracket P \rightsquigarrow Q \rrbracket$  is performed, then the resulting (shared) state still satisfies  $S$ .

Quite remarkably, the following separation logic implication achieves this:

$$(P \multimap^{\exists} S) * Q \Longrightarrow S$$

Informally, it says that if from a state that satisfies  $S$ , we subtract the part of the state satisfying  $P$ , and replace it with some state satisfying  $Q$ , then this should imply that  $S$  holds again. In the case when the action cannot fire, because there is no substate of  $S$  satisfying  $P$ , then  $P \multimap^{\exists} S$  is false and the implication holds trivially.

An assertion  $S$  is stable under interference of a set of actions  $R$  when it is stable under interference of every action in  $R$ .

### 3.4 Specifications and proof rules

A specification of a command  $c$  is a quadruple  $(p, R, G, q)$ , where

- The precondition  $p$  describes the initial state in which  $c$  is executed (both the local and the shared parts of the state).
- The rely  $R$  is a relation (i.e. a set of actions) describing the interference caused by the environment.
- The guarantee  $G$  is a relation describing the changes to the shared state, caused by the program
- The postcondition  $q$  describes the resulting local and shared state, should the execution of  $c$  terminate.

**Definition 1.** *The judgement  $c \text{ sat } (p, R, G, q)$  says that all executions of  $c$  from an initial state satisfying  $p$  and under interference at most  $R$ ,  
(i) do not produce a fault (e.g. access unallocated memory),  
(ii) cause interference at most  $G$ , and,  
(iii) if they terminate, their final state satisfies  $q$ .*

From the definition, we directly get following familiar refinement rule:

$$\frac{p \Rightarrow p' \quad R \Rightarrow R' \quad c \text{ sat } (p', R', G', q') \quad G' \Rightarrow G \quad q' \Rightarrow q}{c \text{ sat } (p, R, G, q)}$$

From separation logic, we inherit the frame rule. This rule says that a program safely running with initial state  $p$  can also be executed with additional local state  $F$  lying around. As the program runs safely without  $F$ , it cannot access  $F$  when it is present; hence,  $F$  is still true at the end.

$$\frac{c \text{ sat } (p, R, G, q)}{c \text{ sat } (p * F, R, G, q * F)}$$

Then, we have a proof rule for each construct in the language. The most complex rule is that of conditional critical regions `await  $b$  do  $c$` .

A simple rule for critical regions would be the following:

$$\frac{\begin{array}{cc} P \text{ stable under } R & Q \text{ stable under } R \\ c \text{ sat } (b \wedge P, \{\}, \{\}, Q) & (P \rightsquigarrow Q) \Longrightarrow G \end{array}}{(\text{await } b \text{ do } c) \text{ sat } (\overline{P}, R, G, \overline{Q})}$$

As in RG, we must check that the precondition and the postcondition are stable under interference from the environment, and that changing the shared state from  $P$  from  $Q$  is allowed by the guarantee  $G$ .

While this rule is sound, it is too weak in two ways. First, it does not allow critical regions to access any local state, as the precondition  $\overline{P}$  requires that the local state is empty. Second, it requires that the critical region changes the entire shared state from  $P$  to  $Q$  and that the guarantee condition allows such a change. Thus, we extend the rule by (i) adding a precondition  $P_2$  and a postcondition  $Q_2$  for the local state, and (ii) allowing the region to change a part  $P_1$  of  $P$  into a part  $Q_1$  of  $Q$ , ensuring that the rest  $F$  does not change.

$$\frac{\begin{array}{ccc} P \text{ stable under } R & (P_1 \rightsquigarrow Q_1) \Longrightarrow G & Q \text{ stable under } R \\ P \Rightarrow P_1 * F & c \text{ sat } (b \wedge (P_1 * P_2), \{\}, \{\}, Q_1 * Q_2) & Q_1 * F \Rightarrow Q \end{array}}{(\text{await } b \text{ do } c) \text{ sat } (\overline{P} * P_2, R, G, \overline{Q} * Q_2)}$$

There is a side-condition to this rule requiring that  $Q$  is a *precise* assertion. This is a technical requirement inherited from concurrent separation logic. It ensures that the splitting of the resultant state into local and shared portions is unambiguous.

We reiterate the parallel composition rule from the introduction. As the interference experienced by thread  $c_1$  can arise from  $c_2$  or the environment of the parallel composition, we have to ensure that this interference  $R \vee G_2$  is allowed by  $R_1$ . Similarly  $c_2$  must be able to tolerate interference from  $c_1$  and from the environment of the parallel composition.

$$\frac{\begin{array}{l} c_1 \text{ sat } (p_1, R_1, G_1, q_1) \quad (R \vee G_1) \implies R_2 \\ c_2 \text{ sat } (p_2, R_2, G_2, q_2) \quad (R \vee G_2) \implies R_1 \quad (G_1 \vee G_2) \implies G \end{array}}{c_1 \parallel c_2 \text{ sat } (p_1 * p_2, R, G, q_1 * q_2)}$$

The precondition and postcondition of the composition are the  $*$  of of the preconditions/postconditions of the individual threads. In essence, this is the conjunction of the shared state assertions, and the separating conjunction of the local state assertions (cf. the semantics of  $*$  in Section 3.1).

The proof rules for sequential composition, conditional and iterative commands are completely standard.

$$\frac{c_1 \text{ sat } (p, R, G, r) \quad c_2 \text{ sat } (r, R, G, q)}{(c_1; c_2) \text{ sat } (p, R, G, q)}$$

$$\frac{c_1 \text{ sat } (p \wedge b, R, G, q) \quad c_2 \text{ sat } (p \wedge \neg b, R, G, q)}{(\text{if } b \text{ then } c_1 \text{ else } c_2) \text{ sat } (p, R, G, q)}$$

$$\frac{c \text{ sat } (p \wedge b, R, G, p)}{(\text{while } b \text{ do } c) \text{ sat } (p, R, G, p \wedge \neg b)}$$

## 4 Examples

In this section, we build a fine-grained concurrent linked list implementation of a mutable set data structure. We associate one lock per list node rather than have a single lock for the entire list. But before we present the details of the linked list, we focus on the locking mechanism for a single node.

**Locks** In Figure 2, we present the source code for our lock and unlock operations.

```
lock(p) {
  await(p.lock==0){ p.lock = tid; }
}
unlock(p) {
  await(true){ p.lock = 0; }
}
```

**Fig. 2.** Source code for operations to lock and unlock a node. For clarity we use a field notation, hence we encode `p.lock` as `[p]`.

We use three predicates to represent a node in the list: (1)  $N_s(x, v, y)$ , for a node at location  $x$  with contents  $v$  and tail pointer  $y$  and with the lock status set to  $s$ ; (2)  $U(x, v, y)$  for an unlocked node at location  $x$  with contents  $v$  and tail pointer  $y$ ; and (3)  $L_t(x, v, y)$  for a node locked with thread identifier  $t$ . We use  $N_-(x, v, y)$  for a node that may or may not be locked.

$$\begin{array}{l} N_s(x, v, y) \stackrel{\text{def}}{=} x \mapsto s, v, y \\ U(x, v, y) \stackrel{\text{def}}{=} N_0(x, v, y) \quad L_t(x, v, y) \stackrel{\text{def}}{=} N_t(x, v, y) \wedge t > 0 \end{array}$$

The thread identifier parameter in the locked node is required to express that a node can only be unlocked, by the thread that locked it.

$$t \in T \wedge U(x, v, n) \rightsquigarrow L_t(x, v, n) \quad (\text{lock})$$

$$t \in T \wedge L_t(x, v, n) \rightsquigarrow U(x, v, n) \quad (\text{unlock})$$

The (lock) and (unlock) operations are parameterised with a set of thread identifiers,  $T$ . This allows us to use the actions to represent both relies and guarantees. In particular, we take a thread with identifier  $\mathbf{tid}$  to have the guarantee with  $T = \{\mathbf{tid}\}$ , and the rely to use the complement on this set.

We have the following derived rules for the lock primitives. These are derived from the (AWAIT) rule.

$$\frac{\begin{array}{l} P \text{ stable under } R \quad Q \text{ stable under } R \\ P \Rightarrow N_{\mathbf{p}}(\mathbf{p}, v, n) * F \quad L_{\mathbf{tid}}(\mathbf{p}, v, n) * F \Rightarrow Q \end{array}}{\text{lock}(\mathbf{p}) \text{ sat } (\overline{P}, R, G, \overline{Q})}$$

$$\frac{\begin{array}{l} P \text{ stable under } R \quad Q \text{ stable under } R \\ P \Rightarrow L_{\mathbf{tid}}(\mathbf{p}, v, n) * F \quad U(\mathbf{p}, v, n) * F \Rightarrow Q \end{array}}{\text{unlock}(\mathbf{p}) \text{ sat } (\overline{P}, R, G, \overline{Q})}$$

**The Algorithm** Now we build a fine-grained concurrent linked list implementation of a set using the lock mechanism we have defined. The list has operations **add** which adds an element to the set, and **remove** which removes an element from the set. Traversing the list uses *lock coupling*: the lock on one node is not released until the next node is locked. Somewhat like a person climbing a rope “hand-over-hand,” you always have at least one hand on the rope.

We present the implementation in Figure 3. An element is added to the set by inserting it in the appropriate position, while holding the lock of its previous node. It is removed by redirecting the previous node’s pointer, while both the previous and the current node are locked. This ensures that deletions and insertions can happen concurrently in the same list. The algorithm makes two assumptions about the list: (1) it is sorted; and (2) the first and last elements have values  $-\infty$  and  $+\infty$  respectively. This allows us to avoid checking for the end of the list.

First, consider the action of adding a node to the list. We begin by describing an action that ignores the sorted nature of the list:

$$t \in T \wedge L_t(x, u, n) \rightsquigarrow L_t(x, u, m) * U(m, v, n)$$

To add an element to the list, we must have locked the previous node, and then we can swing the tail pointer to the added node. The added node must have the same tail as previous node before the update.

<pre> locate(e) {   local p,c;   p = Head;   lock(p);   c = p.next;   while(c.value &lt; e){     lock(c);     unlock(p);     p = c;     c = p.next;   }   return (p,c); } </pre>	<pre> add(e) {   local x,y,z;   (x,z) = locate(e);   if(z.value != e){     y = cons(0,e,z);     x.next = y;   }   unlock(x); } </pre>	<pre> remove(e) {   local x,y,z;   (x,y) = locate(e);   if(y.value == e){     lock(y);     z = y.next;     x.next = z;     unlock(x);     dispose(y);   } } </pre>
--	---	--

**Fig. 3.** Source code for lock coupling list operations. For clarity, we use a field notation, hence we encode `x.value` and `x.next` as `[x+1]` and `[x+2]`, respectively. We use heap reads in conditional tests for `ifs` and `whiles`, which can be encoded using an additional local variable for the heap read.

To ensure we preserve the sorted order the actual add operation must be specified with respect to the next node as well. We must ensure the value we add is between the next and previous values.

$$\begin{aligned}
& (t \in T) \wedge (u < v < w) \wedge (L_t(x, u, n) * N_s(n, w, y)) \\
& \rightsquigarrow L_t(x, u, m) * U(m, v, n) * N_s(n, w, y) \quad (\text{add node})
\end{aligned}$$

The final action we allow is the deletion of an element from the list. We must lock the node we wish to delete,  $n$ , and its previous node,  $x$ . The tail of the previous node must be updated to the deleted nodes tail,  $m$ .

$$(v < \infty) \wedge (t \in T) \wedge (L_t(x, u, n) * L_t(n, v, m)) \rightsquigarrow L_t(x, u, m) \quad (\text{delete node})$$

We define  $I(T)$  as the four actions given above: (lock), (unlock), (add node) and (delete node).  $I(\{t\})$  allows the thread  $t$ : (i) to lock an unlocked node, (ii) to unlock a node that it had locked, (iii) to insert a node in the list immediately after a node that it had locked, and (iv) if two adjacent nodes in the list are locked by  $t$ , to remove the second node from the list by swinging a pointer past it.

We can use separation to describe the structure of the shared list. The following predicate,  $ls(x, A, y)$ , describes a list segment starting at location  $x$  with the final tail value of  $y$ , and with contents  $A$ . We use  $\cdot$  as a list separator.

$$\begin{aligned}
ls(x, A, y) \stackrel{\text{def}}{=} & (x = y \wedge A = \emptyset \wedge emp) \\
& \vee (\exists v z B. x \neq y \wedge A = v \cdot B \wedge N_-(x, v, z) * ls(z, B, y))
\end{aligned}$$

Note, as we use separation logic we do not need any reachability predicates, our predicate is simply a recursively defined predicate. The use of  $*$  insures the list is acyclic.

We have three basic properties of a list segment: (1) it does not contain the end marker; (2) an element can be added to the end, provided its tail is not in the list; and (3) two lists can be appended, provided the end marker of the second list is not contained in the first.

**Definition 2.**  $P \downarrow_x \stackrel{\text{def}}{=} P \wedge \neg(x \mapsto \_ * \text{true})$

**Lemma 1.**  $ls(w, A, z) \iff ls(w, A, z) \downarrow_z$

**Lemma 2.**  $ls(w, A, x) \downarrow_y * N_s(x, v, y) \Rightarrow ls(w, A \cdot v, y)$

**Lemma 3.**  $ls(w, A, x) \downarrow_y * ls(x, B, y) \Rightarrow ls(w, A \cdot B, y)$

Finally, we give a lemma that enables us to delete a node from a list segment.

**Proposition 1.**

$$\begin{aligned} (N_s(x, v, y) \text{--}^* \exists ls(w, A, z)) \\ \iff \exists B, C. (A = B \cdot v \cdot C) \wedge (ls(w, B, x) \downarrow_z * ls(y, C, z) \downarrow_x) \end{aligned}$$

The algorithm works on sorted lists with the first and last values being  $-\infty$  and  $+\infty$  respectively. We define  $s(A)$  to represent this restriction on a logical list  $A$ .

$$\begin{aligned} \text{sorted}(A) &\stackrel{\text{def}}{=} \begin{cases} \text{true} & (A = \epsilon) \vee (A = a \cdot \epsilon) \\ (a < b) \wedge \text{sorted}(b \cdot B) & (A = a \cdot b \cdot B) \end{cases} \\ s(A) &\stackrel{\text{def}}{=} (\exists B. A = -\infty \cdot B \cdot +\infty) \wedge \text{sorted}(A) \wedge \text{emp} \end{aligned}$$

We present the outline for the proofs in Figures 4 and 5. The outline presents the intermediate assertions in the proof. We must prove that the assertions about shared state are stable under the rely. We present a single example here

**Lemma 4.**  $\boxed{\exists A. ls(\text{Head}, A, \text{nil}) * s(A)}$  is stable.

*Proof.* We must prove (3.3) for the four actions in the rely: (lock), (unlock), (add node) and (delete node).

(lock) We must show that:

$$\begin{aligned} ((U(x, v, n) \text{--}^* \exists ls(y, A, z)) * L_t(x, v, n)) \Rightarrow ls(y, A, z) \\ ((U(x, v, n) \text{--}^* \exists s(A \cdot u \cdot v \cdot B)) * L_t(x, v, n)) \Rightarrow s(A \cdot u \cdot v \cdot B) \end{aligned}$$

The first follows as

$$\begin{aligned} &(U(x, v, n) \text{--}^* \exists ls(y, A, z)) * L_t(x, v, n) \\ \Rightarrow &(ls(y, B, x) \downarrow_z * ls(n, C, z) * L_t(x, v, n)) \wedge (A = B \cdot v \cdot C) \\ \Rightarrow &(ls(y, B, x) \downarrow_z * ls(x, v \cdot C, z)) \wedge (A = B \cdot v \cdot C) \\ \Rightarrow &ls(y, A, z) \end{aligned}$$

```

locate(e) {
  local p, c;
  { $\exists A. ls(\text{Head}, A, \text{nil}) * s(A) \wedge -\infty < e$ }
  p = Head;
  { $\exists ZB. ls(\text{Head}, \{\}, p) * N(p, -\infty, Z) * ls(Z, B, \text{nil}) * s(-\infty \cdot B) \wedge -\infty < e$ }
  lock(p);
  { $\exists Z. \exists B. ls(\text{Head}, \{\}, p) * L(p, -\infty, Z) * ls(Z, B, \text{nil}) * s(-\infty \cdot B) \wedge -\infty < e$ }
   $\ll c = p.\text{next}; \gg$ 
  { $\exists B. ls(\text{Head}, \{\}, p) * L(p, -\infty, c) * ls(c, B, \text{nil}) * s(-\infty \cdot B) \wedge -\infty < e$ }
  { $\exists uv. \exists ZAB. ls(\text{Head}, A, p) * L(p, u, c) * N(c, v, Z) * ls(c, B, \text{nil}) * s(A \cdot u \cdot v \cdot B) \wedge u < e$ }
  while ( $\ll c.\text{value} \gg < e$ ) {
    { $\exists uv. \exists ZAB. ls(\text{Head}, A, p) * L(p, u, c) * N(c, v, Z) * ls(c, B, \text{nil}) * s(A \cdot u \cdot v \cdot B) \wedge u < e \wedge v < e$ }
    lock(c);
    { $\exists uvZ. \exists AB. ls(\text{Head}, A, p) * L(p, u, c) * L(c, v, Z) * ls(Z, B, \text{nil}) * s(A \cdot u \cdot v \cdot B) \wedge v < e$ }
    unlock(p);
    { $\exists vZ. \exists AB. ls(\text{Head}, A, c) * L(c, v, Z) * ls(Z, B, \text{nil}) * s(A \cdot v \cdot B) \wedge v < e$ }
    p = c;
    { $\exists uZ. \exists AB. ls(\text{Head}, A, p) * L(p, u, Z) * ls(Z, B, \text{nil}) * s(A \cdot u \cdot B) \wedge u < e$ }
     $\ll c = p.\text{next}; \gg$ 
    { $\exists u. \exists AB. ls(\text{Head}, A, p) * L(p, u, c) * ls(c, B, \text{nil}) * s(A \cdot u \cdot B) \wedge u < e$ }
    { $\exists uv. \exists ZAB. ls(\text{Head}, A, p) * L(p, u, c) * N(c, v, Z) * ls(Z, B, \text{nil}) * s(A \cdot u \cdot v \cdot B) \wedge u < e$ }
  }
  { $\exists uv. \exists ZAB. ls(\text{Head}, A, p) * L(p, u, c) * N(c, v, Z) * ls(Z, B, \text{nil}) * s(A \cdot u \cdot v \cdot B) \wedge u < e \wedge e \leq v$ }
  return (p, c);
}

```

**Fig. 4.** Outline verification of `locate`. Atomic reads and writes to shared state are enclosed in  $\ll$  and  $\gg$ . These atomic reads and writes could be made local by adding auxiliary state, however, this complicates the proof.

```

add(e) {
  local x,y,z;
  { $\exists A. ls(\text{Head}, A, \text{nil}) \wedge s(A)$ }  $\wedge -\infty < e$ 
  (x,z) = locate(e);
  { $\exists uv. \exists ZAB. ls(\text{Head}, A, x) * L(x, u, z) * N(z, v, Z) * ls(Z, B, \text{nil}) * s(A \cdot u \cdot v \cdot B)$ }  $\wedge u < e \wedge e \leq v$ 
  if ( $\ll z.\text{value} \gg \neq e$ ) {
    { $\exists uv. \exists ZAB. ls(\text{Head}, A, x) * L(x, u, z) * N(z, v, Z) * ls(Z, B, \text{nil}) * s(A \cdot u \cdot v \cdot B)$ }  $\wedge u < e \wedge e < v$ 
    y = cons(0, e, z);
    { $\exists uv. \exists ZAB. ls(\text{Head}, A, x) * L(x, u, z) * N(z, v, Z) * ls(Z, B, \text{nil}) * s(A \cdot u \cdot v \cdot B)$ }  $* U(y, e, z) \wedge u < e \wedge e < v$ 
     $\ll x.\text{next} = y; \gg$ 
    { $\exists uv. \exists ZAB. ls(\text{Head}, A, x) * L(x, u, y) * N(y, e, z) * ls(z, B, \text{nil}) * s(A \cdot u \cdot e \cdot B)$ }
  }
  unlock(x);
  { $\exists v. \exists A. ls(\text{Head}, A, \text{nil}) * s(A)$ }
}

remove(e) {
  local x,y,z;
  { $\exists A. ls(\text{Head}, A, \text{nil}) * s(A)$ }  $\wedge -\infty < e \wedge e < +\infty$ 
  (x,y) = locate(e);
  { $\exists uv. \exists ZAB. ls(\text{Head}, A, x) * L(x, u, y) * N(y, v, Z) * ls(Z, B, \text{nil}) * s(A \cdot u \cdot v \cdot B)$ }  $\wedge u < e \wedge e \leq v \wedge e < +\infty$ 
  if ( $\ll y.\text{value} \gg == e$ ) {
    { $\exists u. \exists ZAB. ls(\text{Head}, A, x) * L(x, u, y) * N(y, e, Z) * ls(Z, B, \text{nil}) * s(A \cdot u \cdot e \cdot B)$ }  $\wedge e < +\infty$ 
    lock(y);
    { $\exists u. \exists ZAB. ls(\text{Head}, A, x) * L(x, u, y) * L(y, e, Z) * ls(Z, B, \text{nil}) * s(A \cdot u \cdot e \cdot B)$ }  $\wedge e < +\infty$ 
     $\ll z = y.\text{next}; \gg$ 
    { $\exists u. \exists AB. ls(\text{Head}, A, x) * L(x, u, y) * L(y, e, z) * ls(z, B, \text{nil}) * s(A \cdot u \cdot e \cdot B)$ }  $\wedge e < +\infty$ 
     $\ll x.\text{next} = z; \gg$ 
    { $\exists u. \exists AB. ls(\text{Head}, A, x) * L(x, u, z) * ls(z, B, \text{nil}) * s(A \cdot u \cdot B)$ }  $* L(y, e, z)$ 
    unlock(x);
    { $\exists A. ls(\text{Head}, A, \text{nil}) * s(A)$ }  $* L(y, e, z)$ 
    dispose(y);
    { $\exists A. ls(\text{Head}, A, \text{nil}) * s(A)$ }
  }
  { $\exists A. ls(\text{Head}, A, \text{nil}) * s(A)$ }
}

```

**Fig. 5.** Outline verification of `add` and `delete`. Again, atomic reads and writes to shared state are enclosed in  $\ll$  and  $\gg$ .

and the second as

$$\begin{aligned}
& (U(x, v, n) \dashv\ast \exists s(A)) * L_t(x, v, n) \\
& \Rightarrow \text{false} * L_t(x, v, n) \\
& \Rightarrow \text{false} \\
& \Rightarrow L_{\text{tid}}(y, u, z)
\end{aligned}$$

(unlock) This follows in a very similar way to (lock).

(add node) We will ignore the case where we delete from  $s(A)$  as this follows trivially. Assume  $u < v < w$  and simplifying because  $L_t(x, u, m) * U(m, v, n) * N_s(n, w, y) \Rightarrow ls(x, u \cdot v \cdot w, y)$

$$\begin{aligned}
& \left( \left( \begin{array}{c} L_t(x, u, n) \\ * N_s(n, w, y) \end{array} \right) \dashv\ast \exists ls(\text{Head}, A, \text{nil}) \right) * s(A) * ls(x, u \cdot v \cdot w, y) \\
& \Rightarrow \left( N_s(n, w, y) \dashv\ast \exists \left( \begin{array}{c} ls(\text{Head}, B, x) \\ * ls(n, C, \text{nil}) \end{array} \right) \right) * s(B \cdot u \cdot C) * ls(x, u \cdot v \cdot w, y) \\
& \Rightarrow \left( N_s(n, w, y) \dashv\ast \exists \left( \begin{array}{c} ls(\text{Head}, B, x) \\ * N_{s'}(n, w', y') \\ * ls(y', C', \text{nil}) \end{array} \right) \right) * s(B \cdot u \cdot w' \cdot C') * ls(x, u \cdot v \cdot w, y) \\
& \Rightarrow ls(\text{Head}, B, x) * ls(y, C', \text{nil}) * s(B \cdot u \cdot w \cdot C') * ls(x, u \cdot v \cdot w, y) \\
& \Rightarrow ls(\text{Head}, B \cdot u \cdot v \cdot w \cdot C', \text{nil}) * s(B \cdot u \cdot w \cdot C') \\
& \Rightarrow ls(\text{Head}, B \cdot u \cdot v \cdot w \cdot C', \text{nil}) * s(B \cdot u \cdot v \cdot w \cdot C') \\
& \Rightarrow \exists A. ls(\text{Head}, A, \text{nil}) * s(A)
\end{aligned}$$

(delete node)

We will ignore the case where we delete from  $s(A)$  as this follows trivially. Assume  $u < +\infty$  and simplifying because  $L_t(x, u, y) \Rightarrow ls(x, u, y)$

$$\begin{aligned}
& \left( \left( \begin{array}{c} L_t(x, u, n) \\ * L_t(n, w, y) \end{array} \right) \dashv\ast \exists ls(\text{Head}, A, \text{nil}) \right) * s(A) * ls(x, u, y) \\
& \Rightarrow \left( L_t(n, w, y) \dashv\ast \exists \left( \begin{array}{c} ls(\text{Head}, B, x) \\ * ls(n, C, \text{nil}) \end{array} \right) \right) * s(B \cdot u \cdot C) * ls(x, u, y) \\
& \Rightarrow \left( L_t(n, w, y) \dashv\ast \exists \left( \begin{array}{c} ls(\text{Head}, B, x) \\ * N_{s'}(n, w', y') \\ * ls(y', C', \text{nil}) \end{array} \right) \right) * s(B \cdot u \cdot w' \cdot C') * ls(x, u, y) \\
& \Rightarrow ls(\text{Head}, B, x) * ls(y, C', \text{nil}) * s(B \cdot u \cdot w \cdot C') * ls(x, u, y) \\
& \Rightarrow ls(\text{Head}, B \cdot u \cdot C', \text{nil}) * s(B \cdot u \cdot w \cdot C') \\
& \Rightarrow ls(\text{Head}, B \cdot u \cdot C', \text{nil}) * s(B \cdot u \cdot C') \\
& \Rightarrow \exists A. ls(\text{Head}, A, \text{nil}) * s(A)
\end{aligned}$$

Although the proof of stability is long, the proof steps are largely automatic. In future work, we plan to build a tool to automate these checks by modifying Smallfoot [2], a separation logic theorem prover.

## 5 Semantics and soundness

[In preparation]

## 6 Related work

Owicki & Gries [13] introduced the concept of non-interference between the proofs of parallel threads. Their method is not compositional and does not permit top-down development of a proof because the final check of interference-freedom may fail rendering the whole development useless.

To address this problem, Jones [10] introduced the compositional rely/guarantee method [10]. In the VDM-style, Jones opted for ‘two-state’ postconditions; other authors [18, 15] have chosen single-state postconditions. Several authors [18, 15, 5] have proved the soundness and relative completeness of rely/guarantee; Prensas’s proof [15] is machine checked by the Isabelle theorem prover. The completeness results are all modulo the introduction of auxiliary variables. Abadi and Lamport [1] have adapted RG to temporal logic and have shown its soundness for safety specifications.

Separation logic [16, 12] takes a different approach to interference by forbidding it except in critical regions [9]. An invariant,  $I$ , is used to describe the shared state. This is a simple case of our system where the interference specifications (i.e.  $R$  and  $G$ ) are restricted to a very simple relation,  $I \rightsquigarrow I$ . Brookes has shown concurrent separation logic to be sound [4].

There have been attempts to verify fine-grained concurrent algorithms using both separation logic and rely/guarantee. Vafeiadis *et al.* [17] verify several list algorithms using rely/guarantee. Their proofs require reachability predicates to describe lists and they cannot deal with the disposal of nodes. Parkinson *et al.* [14] verify a non-blocking stack algorithm using concurrent separation logic. Their proof requires a lot of auxiliary state to encode the possible interference. With the logic presented in this paper much of the auxiliary state can be removed, and hence the proof becomes clearer.

## 7 Conclusion

We have presented a marriage of rely/guarantee with separation logic. This logic allows us to give a clear and simple proof of a lock-coupling list algorithm. Moreover, we inherit the ability to deal with memory disposal. We do not know of any other approach that can deal with disposal without adding complications to the proof.

We are currently investigating automation. For a restricted class of assertions, it is possible to build a proof checker in the style of Smallfoot [2] that performs the stability checks and checks a proof outline. In addition, we believe that the proof outline and the interference specifications may be inferred using abstract interpretation as done by Distefano, O’Hearn and Yang [6] for heap analysis.

## References

1. M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. Prog. Lang. Syst.*, 17(3):507–534, May 1995.

2. J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In *APLAS*, pages 52–68, 2005.
3. R. Bornat, C. Calcagno, and H. Yang. Variables as resource in separation logic. *Electr. Notes Theor. Comput. Sci.*, 155:247–276, 2006.
4. S. D. Brookes. A semantics for concurrent separation logic. In *CONCUR*, volume 3170 of *Lecture Notes in Computer Science*, pages 16–34. Springer, 2004.
5. J. W. Coleman and C. B. Jones. A structural proof of the soundness of rely/guarantee rules. Technical Report CS-TR-987, School of Computing Science, Newcastle University, Oct. 2006.
6. D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, volume 3920 of *LNCS*, pages 287–302, 2006.
7. M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Prog. Lang. Syst.*, 15(5):745–770, November 1993.
8. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
9. C. A. R. Hoare. Towards a theory of parallel programming. *Operating Systems Techniques*, 1971.
10. C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.
11. C. Morgan. The specification statement. *ACM Trans. Program. Lang. Syst.*, 10(3):403–419, 1988.
12. P. W. O'Hearn. Resources, concurrency and local reasoning. In *CONCUR*, volume 3170 of *Lecture Notes in Computer Science*, pages 49–67. Springer, 2004.
13. S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Commun. ACM*, 19(5):279–285, May 1976.
14. M. J. Parkinson, R. Bornat, and P. W. O'Hearn. Modular verification of a non-blocking stack. In *POPL*, 2007. to appear.
15. L. Prensa Nieto. The rely-guarantee method in Isabelle/HOL. In *European Symposium on Programming (ESOP'03)*, volume 2618 of *LNCS*, pages 348–362. Springer, 2003.
16. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.
17. V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearisable objects. In *Proc. Symp. on Principles and Practice of Parallel Programming*. ACM Press, 2006.
18. Q. Xu, W. P. de Roever, and J. He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.