Modular Safety Checking for Fine-Grained Concurrency

Cristiano Calcagno[1], Matthew Parkinson[2], and Viktor Vafeiadis[2]

[1] Imperial College, London
[2] University of Cambridge

Abstract. Concurrent programs are difficult to verify because the proof must consider the interactions between the threads. Fine-grained concurrency and heap allocated data structures exacerbate this problem, because threads interfere more often and in richer ways. In this paper we provide a thread-modular safety checker for a class of pointer-manipulating fine-grained concurrent algorithms. Our checker uses ownership to avoid interference whenever possible, and rely/guarantee (assume/guarantee) to deal with interference when it genuinely exists.

1 Introduction

Traditional concurrent implementations use a single synchronisation mechanism, such as a lock, to guard an entire data structure (such as a list or a hash table). This *coarse-grained* synchronisation makes it relatively easy to reason about correctness, but it limits concurrency, negating some of the advantages of modern multi-core or multi-processor architectures. A *fine-grained* implementation permits more concurrency by allowing multiple threads to access the same data structure simultaneously. Of course, this makes it far harder to reason about correctness.

There has been a lot of reseach [9, 11, 1, 24, 29] on verifying coarse-grained concurrent programs, but hardly any on verifying fine-grained concurrency. Recently, we have presented a new logic, RGSep [28], and demonstrated its use in precisely and concisely describing the inter-thread interference of fine-grained concurrent algorithms—thus making the proof of fine-grained concurrent programs easier. The logic merges aspects from both rely/guarantee reasoning [15] and concurrent separation logic [18, 5], giving rise to simple, modular proofs about algorithms with intricate concurrency and dynamic memory management.

The difficulty with RGSep [28] is that it is simply a logic: users of it must manually prove their programs correct with pen and paper. In this paper, we automated a suitable subset of RGSep and implemented a new modular tool that automatically verifies safety properties of a class of intricate concurrent algorithms.

Like ESC/Java [10] and Spec# [2], our tool symbolically simulates the code and produces verification conditions that, if proved valid, imply that the program is correct with respect to the user-supplied pre-/post-condition pair. In doing

so, our tool splits the state (heap) into thread-local state and shared state. We maintain this partition throughout symbolic execution using assertions that describe the partition between the local and the shared state. The assertions are restricted to a subset of separation logic chosen to support a form of symbolic execution, a decidable proof theory for symbolic heaps, and the inference of frames for handling procedure calls [3]. Our prover starts with the precondition, symbolically executes the code deriving a postcondition, and checks that this implies the user-supplied postcondition.

In order to handle fine-grained concurrency modularly, we require the programmer to describe the interference between threads using lightweight annotations. These take the form of actions done by the program; they are much more concise than invariants and much easier to come up with. So far, our tool checks only safety properties: in particular, data integrity, memory leaks, and race-conditions.[3] It does not check liveness properties like termination.

Our technical contributions are:

- to enrich the set of separation logic operators handled automatically (see §2.1);
- a procedure for calculating the interference imposed by the environment, which, given an assertion, computes a weaker assertion that is *stable* under interference and, hence, valid to use in a rely-guarantee proof (see §2.3);
- a symbolic execution for RGSep assertions (see §2.4);
- an automatic safety checker specialised to list-manipulating programs; and
- verification of a series of fine-grained concurrent algorithms.

In Section 3, we describe the tool by example. In particular, we demonstrate a verification of a lock-coupling list algorithm, which highlights (1) dynamic lock allocation; (2) memory deallocation, including locks; and (3) non-nested locking. In Section 4, we evaluate the performance of our tool.

2 The analysis

2.1 RGSep assertions

Our tool splits the state (heap) into thread-local state and shared state, hence our assertions specify a state consisting of two heaps with disjoint domains: the local heap (visible to a single thread), and the shared heap (visible to all threads). Normal formulae, P, specify the local heap, whereas boxed formulae, $\boxed{P}$, specify the shared heap.[4] Note that boxes cannot be nested.

Our tool accepts assertions written in the following grammar:

$$\begin{aligned} A, B &::= E_1{=}E_2 \mid E_1{\neq}E_2 \mid E \mapsto \rho \mid \mathsf{lseg}(E_1, E_2) \mid \mathsf{junk} \\ P, Q, R, S &::= A \mid P \vee Q \mid P * Q \mid P \mathrel{-\!\!\circledast} Q \mid P\lfloor_{E_1,\ldots,E_n} \\ p, q &::= p \vee q \mid P * \boxed{Q} \end{aligned}$$

[3] We allow racy interference where the user specifies it, but not elsewhere.

[4] More generally, we support multiple disjoint regions of shared state, and boxes are annotated with the name r of the region: $\boxed{P}_r$. For clarity of exposition, we present the analysis with respect to a single resource, and omit the subscript.

where E is a pure expression, an expression that does not depend on the heap. All variables starting with an underscore (e.g., $\_\mathsf{x}$) are implicitly existentially quantified. In the assertion $P * \boxed{Q}$, if X contains the set of existential free variables of P and Q, then their scope is $\exists X.\ P * \boxed{Q}$.

The first line contains the usual atomic assertions of separation logic: pure predicates (that do not depend on the heap), heap cells ($E \mapsto \rho$), list segments ($\mathsf{lseg}(E_1, E_2)$), and junk. $E \mapsto \rho$ asserts that the heap consists of a single memory cell with address E and contents ρ, where ρ is a mapping from field names to values (pure expressions); $\mathsf{lseg}(E_1, E_2)$ says that the heap consists of an acyclic linked list segment starting at E_1 and ending at E_2; junk asserts the heap may contain inaccessible state.

The second line contains operators for building larger formulae. Disjunction, $P \vee Q$, asserts that the heap satisfies P or Q. Separating conjunction, $P * Q$, asserts that the heap can be divided into two (disjoint) parts, one satisfying P and the other satisfying Q. For notational convenience, we let pure formulae (e.g., $E_1 = E_2$) hold only on the empty heap, and use only one connective ($*$) to express both ordinary conjunction for pure formulas[5], and the separating conjunction between heap formulas. The other two operators are new.

- *Septraction* ($-\!\circledast$) is defined as $h \models (P -\!\circledast Q) \iff \exists h_1\, h_2.\ h_2 = h * h_1$ and $h_1 \models P$ and $h_2 \models Q$. This operation can be thought of as subtraction or differentiation, as it achieves the effect of subtracting heap h_1 satisfying P from the bigger heap h_2 satisfying Q.
- The "dangling" operator $P\lfloor_D$ asserts that P holds and that all locations in the set D are not allocated. This can be defined in separation logic as $P\lfloor_{(E_1,\ldots,E_n)} \iff P \wedge \neg((E_1 \mapsto \_) * \mathit{true}) \wedge \cdots \wedge \neg((E_n \mapsto \_) * \mathit{true})$, but it is better treated as a built-in assertion form, because it is much easier to analyse than $\wedge$ and $\neg$.

(For formal definitions and further detail, please see the technical report on the logic [28].)

Finally, the third line introduces $P * \boxed{Q}$, the novel assertion of RGSep, which does not exist in separation logic. It asserts that the shared state satisfies Q and that the local state is separate and satisfies P.

Extending any separation logic theorem prover to handle the dangling operator, $\lfloor_D$, and septraction, $-\!\circledast$, is relatively simple. The 'dangling' operator can be eliminated from all terms (see Fig. 1), except for terms containing recursive predicates, such as lseg. Recursive predicates require the dangling set D to be passed as a parameter,

$$\mathsf{lsegi}_{tl,\rho}(E_1, E_2, D) \stackrel{\text{def}}{=} (E_1 = E_2) \vee \exists x.\ E_1 \mapsto (tl{=}x, \rho)\lfloor_D * \mathsf{lsegi}_{tl,\rho}(x, E_2, D)$$

[5] Technically, we write for example $E_1{=}E_2$ as an abbreviation for the separation logic formula $(E_1{=}_{\mathrm{SL}}E_2) \wedge \mathit{emp}$, so that $(E_1{=}E_2) * P$ is equivalent to $(E_1{=}_{\mathrm{SL}}E_2) \wedge P$.

$$
\begin{array}{rcl}
(F \mapsto \rho)\lfloor_D & \iff & F \neq D * (F \mapsto \rho) \\
\multicolumn{3}{r}{\text{where } [F{\neq}\{E_1,\ldots,E_n\} \stackrel{\text{def}}{=} F{\neq}E_1 * \cdots * F{\neq}E_n]} \\
\mathsf{lsegi}_{tl,\rho}(E,F,D')\lfloor_D & \iff & \mathsf{lsegi}_{tl,\rho}(E,F,D \cup D') \\
(P * Q)\lfloor_D & \iff & P\lfloor_D * Q\lfloor_D \\
(P \vee Q)\lfloor_D & \iff & P\lfloor_D \vee Q\lfloor_D
\end{array}
$$

$$
\begin{array}{rcl}
(E_1 \mapsto \rho_1) \mathrel{-\!\circledast} (E_2 \mapsto \rho_2) & \iff & E_1 = E_2 * \rho_1 = \rho_2 \\
(E_1 \mapsto \mathtt{tl}{=}E_2, \rho) \mathrel{-\!\circledast} \mathsf{lsegi}_{tl,\rho'}(E,E',D) & \iff & \\
\multicolumn{3}{r}{E_1{\neq}0 * E_1{\neq}D * \rho{=}\rho' * \mathsf{lsegi}_{tl,\rho'}(E,E_1,D)\lfloor_{E'} * \mathsf{lsegi}_{tl,\rho'}(E_2,E',D)\lfloor_{E_1}} \\
\multicolumn{3}{r}{\text{where } [\rho{=}\rho' \stackrel{\text{def}}{=} \forall f \in (dom(\rho) \cap dom(\rho')).\rho(f) = \rho'(f)]} \\
(E \mapsto \rho) \mathrel{-\!\circledast} (P * Q) & \iff & P\lfloor_E * (E \mapsto \rho \mathrel{-\!\circledast} Q) \\
\multicolumn{3}{r}{\vee \ (E \mapsto \rho \mathrel{-\!\circledast} P) * Q\lfloor_E} \\
(E \mapsto \rho) \mathrel{-\!\circledast} (P \vee Q) & \iff & (E \mapsto \rho \mathrel{-\!\circledast} P) \ \vee \ (E \mapsto \rho \mathrel{-\!\circledast} Q) \\
(P * Q) \mathrel{-\!\circledast} R & \iff & P \mathrel{-\!\circledast} (Q \mathrel{-\!\circledast} R) \\
(P \vee Q) \mathrel{-\!\circledast} R & \iff & (P \mathrel{-\!\circledast} R) \vee (Q \mathrel{-\!\circledast} R)
\end{array}
$$

Fig. 1. Elimination rules for $P\lfloor_D$ and septraction ($-\!\circledast$).

We subscript the list segments with the linking field, tl, and any common fields, ρ, that all the nodes in the list segment have.[6] We omit the subscript when the linking field is tl and ρ is empty. Unlike the definition of lseg, our new definition is imprecise: $\mathsf{lsegi}(E,E,\{\})$ describes both the empty heap and a cyclic list. The imprecise definition allows us to simplify the theorem prover, as the side-conditions for appending list segments are not needed. A precise list segment, $\mathsf{lseg}(E_1,E_2)$, is just a special case our imprecise list segment, $\mathsf{lsegi}(E_1,E_2,\{E_2\})$. Another benefit of the dangling operator is that some proof rules can be strengthened, removing some causes of incompleteness. For instance, the following application of the proof rule for deallocating a memory cell $\{P * \mathtt{x}{\mapsto}\_\}$ `dispose(x)` $\{P\}$ can be strengthened by rewriting the precondition and obtain $\{P\lfloor_{\mathtt{x}} * \mathtt{x}{\mapsto}\_\}$ `dispose(x)` $\{P\lfloor_{\mathtt{x}}\}$. Similarly, we can eliminate the septraction operator from $P{-\!\circledast}Q$, provided P does not contain any lseg or junk predicates (see Fig. 1). Had we allowed further inductive predicates, such as $\mathsf{tree}(E_1)$, we would have needed an additional rule for computing $(E \mapsto \rho) \mathrel{-\!\circledast} \mathsf{tree}(E_1)$.

Finally, assertions containing boxes are always written in a canonical form, $\bigvee_i (P_i * \boxed{Q_i})$. Given an implication between formulas in this form, we can essentially check implications between normal separation logic formulae, by the following lemma:

$$(P \vdash P') \wedge (Q \vdash Q') \Longrightarrow (P * \boxed{Q} \vdash P' * \boxed{Q'})$$

Furthermore, we can deduce from $P * \boxed{Q}$ all the heap-independent facts, such as $x \neq y$, which are consequences of $P * Q$, since shared and local states are always disjoint.

[6] This is important for the `lazy list` algorithm, as the invariant involves a list where all the nodes are marked as deleted (have a `marked` field set to 1).

2.2 Interference actions and stability

RGSep assertions distinguish between the local and the shared state. Local state belongs to a single thread and cannot be accessed by other concurrent threads. Shared state, however, can be accessed by any thread; hence, the logic models interference from the other threads.

In the style of rely/guarantee, we specify interference as a binary relation between (shared) states, but represent it compactly as a set of actions (updates) to the shared state. In this paper, we do not attempt to infer such actions; instead, we provide convenient syntax for the user to define them.

Consider the following two action declarations:

```
action Lock(x)     [x|->lk=0  ]  [x|->lk=TID]
action Unlock(x)   [x|->lk=TID]  [x|->lk=0  ]
```

Each action has a name, some parameters, a precondition and a postcondition. For instance, `Lock(x)` takes a location `x` whose `lk` field is zero, and replaces it with `TID`, which stands for the current thread identifier (which is unique for each thread and always non-zero). Crucially, the precondition and the postcondition delimit the overall footprint of the action on the shared state. They assert that the action does not modify any *shared* state other than `x`.

We abstract the behaviour of the environment as a set of actions. We say that an assertion S is stable (i.e., unaffected by interference), if and only if, for all possible environment actions `Act`, if S holds initially and `Act` executes then S still holds at the end. Assertions about the local state are stable by construction, because interference does not affect the local state. An assertion $\boxed{S}$ about the shared state is stable under the action $P \rightsquigarrow Q$, if and only if, the following implication holds

$$(P \mathbin{-\!\circledast} S) * Q \implies S.$$

The formula $(P \mathbin{-\!\circledast} S) * Q$ represents the result of executing the environment action $P \rightsquigarrow Q$ on the initial state S: that is to remove P from S and put back Q in its place. This form of environment execution is reminiscent of the idea of execution of specification statements [17]. The crucial difference here is that we do not require the implication $S \Rightarrow (P * \mathit{true})$ to hold, which would amount to checking that the environment can execute $P \rightsquigarrow Q$ on *all* the states described by S.

2.3 Inferring stable assertions

Most often, the postcondition of a critical section obtained by symbolic execution is not stable under interference; therefore, we must find a stable postcondition which is weaker than the original.

Assume for the time being that the rely contains a single action `Act` with precondition P and postcondition Q; later, we will address the general case.

Mathematically, inferring a stable assertion from an unstable assertion S is a straightforward fixpoint computation

$$S_0 = S \qquad S_{n+1} = S_n \vee (P \mathrel{-\!\!\circledast} S_n) * Q,$$

where S_n is the result of at most n executions of `Act` starting from S. This computation, however, does not always terminate, because the assertions can contain an unbounded number of existentially quantified variables, and hence the domain is infinite.

We approximate the fixpoint by using abstract interpretation. Our concrete domain is the set of syntactic Smallfoot assertions, and our abstract domain is a finite subset of normalised Smallfoot assertions that contain a bounded number of existential variables. Both domains are lattices ordered by implication, with *true* as $\top$ and *false* as $\bot$; $\vee$ is join.

We have a lossy abstraction function $\alpha : \mathsf{Assertion} \to \mathsf{RestrictedAssertion}$ that converts a Smallfoot assertion to a restricted assertion, and a concretisation function $\gamma : \mathsf{RestrictedAssertion} \to \mathsf{Assertion}$ which is just the straightforward inclusion (i.e., the identity) function. In our implementation, the abstraction function α is computed by applying a set of abstraction rules, an adaptation of the rules of Distefano et al. [8]. Our abstraction function ensures that any existential variable in the final assertion is reachable from at least two normal (program) variables. Hence, as there is a finite number of program variables and field names, the number of existential variable is also finite. To do this, we convert assertions such as $x \mapsto \_y * \_y \mapsto z$ into a list segment. The details are at the end of this section. Nevertheless, the technique is parametric to any suitable abstraction function.

The fixpoint can be computed in the abstract domain as follows:

$$S_0 = \alpha(S) \qquad S_{n+1} = S_n \vee \alpha((P \mathrel{-\!\!\circledast} S_n) * Q).$$

In the general case we have n actions $\mathtt{act}_1, \ldots, \mathtt{act}_n$. Two natural algorithms are to interleave the actions during the fixpoint computation, or to stabilise one action at a time. We found that the latter strategy gives better execution times.

We now present an example of stabilisation. Consider stabilising an assertion $x \mapsto lk = 0 * y \mapsto lk = \mathtt{TID}$ with the `Lock` and `Unlock` actions from Section 2.2. Before stabilising, we replace variable `TID` in the specification of the actions with a fresh existentially quantified variable $\_tid$, and add assumptions $\_tid \neq 0$ and $\_tid \neq \mathtt{TID}$. The idea is that any thread might be executing in parallel with our thread, and all we know is that the thread identifier cannot be 0 (by a design choice) and it cannot be `TID` (because `TID` is the thread identifier of our thread). Stabilisation will involve the first and third rules in Figure 1. In most cases, an inconsistent assertion would be generated by adding one of the following equalities: $0 = 1$, $0 = \_tid$, $\mathtt{TID} = \_tid$. In the following fixpoint computation we

do not list those cases.

$$
\begin{aligned}
S_0 &\iff \alpha(x \mapsto lk = 0 * y \mapsto lk = \texttt{TID}) = x \mapsto lk = 0 * y \mapsto lk = \texttt{TID} \\
&\qquad\qquad \text{action } \texttt{lock} \\
S_1 &\iff S_0 \vee \alpha(\_tid \neq 0 * \_tid \neq \texttt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \texttt{TID}) \\
&\iff S_0 \vee (\_tid \neq 0 * \_tid \neq \texttt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \texttt{TID}) \\
&\iff \_tid \neq \texttt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \texttt{TID} \\
&\qquad\qquad \text{action } \texttt{lock} \\
S_2 &\iff S_1 \vee \alpha(\_tid' \neq 0 * \_tid' \neq \texttt{TID} * x \mapsto lk = \_tid' * y \mapsto lk = \texttt{TID}) \\
&\iff S_1 \vee (\_tid' \neq 0 * \_tid' \neq \texttt{TID} * x \mapsto lk = \_tid' * y \mapsto lk = \texttt{TID}) \\
&\iff (\_tid \neq \texttt{TID} * x \mapsto lk = \_tid * y \mapsto lk = \texttt{TID}) \iff S_1 \\
&\qquad\qquad \text{action } \texttt{unlock} \\
S_3 &\iff S_2 \vee \alpha(x \mapsto lk = 0 * y \mapsto lk = \texttt{TID}) \\
&\iff S_2 \vee (x \mapsto lk = 0 * y \mapsto lk = \texttt{TID}) \\
&\iff S_2
\end{aligned}
$$

In this case, we do not need to stabilise with respect to `lock` again, since `unlock` produced no changes.

Adaptation of Distefano et al.'s abstraction We assume that variable names are ordered, so that existential variables are smaller than normal variables. We present an algorithm that abstracts a formula $P = (A_1 * \dots A_n)$ where each A_i is an atomic formula.

1. Rewrite all equalities $E = F$, so that E is a single variable, which is 'smaller' than all the variables in F.
2. For each equality $E = F$ in P, substitute any other occurences of E in P by F; if E is an existential variable, discard the equality.
3. For each A_i describing a memory structure (i.e., a cell or a list segment) starting at an existential address, find all other terms that can point to that address. If there are none, A_i is unreachable; replace it with junk. If there is only one, then try to combine them into a list segment. If there are more than one, so A_i is shared, leave them as they are.
   To combine two terms into a list segment, we use the following implication:

   $$L_{\texttt{tl},\rho_1}(E_1, \texttt{\_x})\lfloor_{D_1} * L_{\texttt{tl},\rho_2}(\texttt{\_x}, E_2)\lfloor_{D_2} \implies \mathsf{lseg}_{\texttt{tl},\rho_1\cap\rho_2}(E_1, E_2)\lfloor_{D_1\cap D_2}$$

   where $L_{\texttt{tl},\rho}(E, F)\lfloor_D$ is $lseg_{\texttt{tl},\rho}(E, F)\lfloor_D$ or $E \mapsto (\texttt{tl} = F, \rho) * E \neq D$. This is a generalisation of Distefano et al.'s rule, because our list segments record common fields ρ of nodes, and the set D of disjoint memory locations. In addition, because our list segments are imprecise, we do not need Distefano et al's side condition that E_2 is `nil` or allocated separately.
4. Put the formulae in a canonical order, by renaming their existential variables. This is achieved by, first, ordering atomic formulas by only looking at their shape, while ignoring the ordering between existential variables, and then, renaming the existential variables based on the order they appear in the ordered formula.

If we simply ran this analysis as described above, we would lose too much information and could not prove even the simplest programs. This is because the analysis would abstract $x{\mapsto}(lk{=}\texttt{TID}, tl{=}\_y){*}lseg(\_y, z)$ into $lseg(x, z)$, forgetting that the node x was locked! Instead, before we start the fixed point calculation, we replace existential variables in such $\mapsto$ assertions containing occurrences of `TID` with normal variables to stop the abstraction rules from firing. As the number of normal variables does not increase during the fixpoint computation, the analysis still terminates. At the end of the fixed point calculation, we replace them back with existential variables. Our experiments indicate that this simple heuristic gives enough precision in practice. We have also found that turning dead program variables into existential variables before starting the fixed point calculation significantly reduces the number of cases and speeds up the analysis.

2.4 Symbolic Execution of Atomic Blocks

We discharge verification conditions by performing a form of symbolic execution [16, 3] on symbolic states, and then check that the result implies the given postcondition. Symbolic heaps are formulae of the form

$$A_1 * \ldots * A_m * \boxed{\bigvee_i B_{1,i} * \ldots * B_{n_i,i}}$$

where each A_i and $B_{i,j}$ is an atomic formula. The A part of a symbolic heap describes the local state of the thread, and the B part – inside the box – describes the shared part. We use disjunction for boxed assertions to represent more compactly the result of stabilisation, and avoid the duplication of the shared part for each disjunct. Symbolic states are finite sets of symbolic heaps, representing their disjunction. This kind of setup is typical of work on shape analysis using separation logic [8].

We allow operations to contain non-pure (accessing the heap) expressions in guards and assignments, by translating them into a series of reads to temporary variables followed by an assignment or a conditional using pure expressions, e.g. `assume(x->tl==0)` would be translated to `temp = x->tl; assume(temp==0)`, for a fresh variable `temp`. We omit the obvious details of this translation.

Except for atomic blocks, the symbolic execution is pretty standard: the shared component is just passed around. For atomic blocks more work is needed. We describe in detail the execution of an atomic block `atomic(B) {C} as Act(x)` starting from symbolic precondition $X * \boxed{S}$. Intuitively, the command is executed atomically when the condition `B` is satisfied. The annotation `as Act(x)` specifies that command C performs shared action `Act` with the parameter instantiated with x. Suppose that `Act` was declared as `action Act(x) [P] [Q]`. Our task is to find the postcondition Ψ in the following Hoare triple:

$$\{X * \boxed{S}\}\ \texttt{atomic(B) C as Act(x);}\ \{\Psi\}$$

Our algorithm consists of 4 parts, corresponding to the premises of the following inference rule.

$$\frac{\{X{*}S\}\ \texttt{assume(B)}\ \{X{*}P{*}F\} \quad \{X{*}P\}\ \texttt{C}\ \{X'\} \quad X' \vdash Q{*}Y \quad \mathit{stab}(Q{*}F) = R}{\{X * \boxed{S}\}\ \texttt{atomic(B) C as Act(x);}\ \{Y * \boxed{R}\}}$$

Step 1. Add shared state S to the local state, and call the symbolic execution and theorem prover to infer the frame F such that $[X * S]\texttt{assume(B)}[X * P * F]$. This step has the dual function of checking that the action's precondition P is implied, and also inferring the leftover state F, which should not be accessed during the execution of `C`. The symbolic execution of `assume(B)` removes cases where `B` evaluates to false. Notice that the evaluation of `B` can access the shared state. If this step fails, the action's precondition cannot be met, and we report an error.

Step 2. Execute the body of the atomic block symbolically starting with $X * P$. Notice that F is not mentioned in the precondition: because of the semantics of Hoare triples in separation logic, this ensures that command `C` does not access the state described by F, as required by the specification of `Act`.

Step 3. Call the theorem prover to infer the frame Y such that $X' \vdash Q * Y$. As before, this has the effect of checking that the postcondition Q is true at the end of the execution, and inferring the leftover state Y. This Y becomes the local part of the postcondition. If the implication fails, the postcondition of the annotated action cannot be met, and we report an error.

Step 4. Combine the shared leftover F computed in the first step with the shared postcondition Q, and stabilise the result $Q * F$ with respect to the execution of actions by the environment as described in Section 2.2.

Precision Similar to resource invariants of concurrent separation logic, RGSep requires that R in the rule above is *precise*[19], or in the presence of memory leaks *supported*. We postulate that checking for precision is unnecessary, if we drop the rule of conjunction from RGSep, which we happen not to use in the analysis. We are investigating a formal proof of soundness for that form of RGSep.

Read-only atomics We have a simplified rule for read-only atomic regions that does not require an action specification:

$$\frac{\{S\}\ \texttt{C}\ \{X'\} \quad stab(X') = R \quad \texttt{C is read-only.}}{\{\boxed{S}\}\ \texttt{atomic C}\ \{\boxed{R}\}}$$

3 Example: Lock Coupling List

We demonstrate, by example, that our tool can automatically verify the safety of a fine-grained concurrent linked list. We associate one lock per list node rather than have a single lock for the entire list. The list has operations `add` which adds an element to the list, and `remove` which removes an element from the list. 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.

Figure 2 contains the annotated input to our tool. Next, we informally describe the annotations required, and also the symbolic execution of our tool. In the tool the assertions about shared states are enclosed in `[ ... ]` brackets, rather

```
action Lock(x)      [x|->lk=0,tl=_w  ] [x|->lk=TID,tl=_w]
action Unlock(x)    [x|->lk=TID,tl=_w] [x|->lk=0,tl=_w]
action Add(x,y)     [x|->lk=TID,tl=_w] [x|->lk=TID,tl=y * y|->tl=_w]
action Remove(x,y) [x|->lk=TID,tl=y * y|->lk=TID,tl=_z] [x|->lk=TID,tl=_z]

ensures: [a!=0 *lseg(a,0)]
init() { a = new(); a->tl = 0; a->lk = 0; }

lock(x) { atomic(x->lk == 0) { x->lk = TID; } as Lock(x); }
unlock(x) { atomic { x->lk = 0; } as Unlock(x); }
```

```
requires: [a!=0 * lseg(a,0)]
ensures:  [a!=0 * lseg(a,0)]
add(e) { local prev,curr,temp;
  prev = a;
  lock(prev);                          } (a)
  atomic { curr = prev->tl; }          } (b)
  if (curr!=0)
    atomic { temp = curr->hd; }
  while(curr!=0 && temp<e) {
    lock(curr);
    unlock(prev);                      } (c)
    prev = curr;
    atomic { curr = prev->tl; }
    if (curr!=0)
      atomic { temp = curr->hd; }
  }
  temp = new();
  temp->lk= 0;
  temp->hd = e;                        } (d)
  temp->tl = curr;
  atomic { prev->tl = temp; }          } (e)
         as Add(prev,temp);
  unlock(prev);
}
```

```
requires: [a!=0 * lseg(a,0)]
ensures:  [a!=0 * lseg(a,0)]
remove(e) { local prev,curr,temp;
  prev = a;
  lock(prev);
  atomic { curr = prev->tl; }
  if (curr!=0)
    atomic { temp = curr->hd; }
  while(curr!=0 && temp!=e) {
    lock(curr);
    unlock(prev);
    prev = curr;
    atomic { curr = prev->tl; }
    if (curr!=0)
      atomic { temp = curr->hd; }
  }
  if (curr!=0) {
    lock(curr);
    atomic { temp = prev->tl; }
    atomic { prev->tl = temp; }        } (f)
          as Remove(prev,curr);
    dispose(curr);
  }
  unlock(prev);
}
```

Fig. 2. Lock-coupling list. Annotations are in italic font.

than a box. For example, in the assertion `x|->hd=9 * [y|->hd=10]`, the cell at `x` is local whereas that at `y` is shared.

Note that we calculate loop invariants with a standard fixed-point computation, which uses the same abstraction function as for stabilisation.

We proceed by explaining the highlighted parts of the verification (a)-(f).

Executing an atomic block (a) First, we illustrate the execution of an atomic block by considering the first `lock` in the `add` function, following the rule in the previous section. (Step 1) We execute the guard and find the frame.

```
prev==a * a!=0 * lseg(a,0)
  assume(prev->lk == 0);
prev==a * a!=0 * prev|->lk:0,tl:_z * lseg(_z,0)
```

The execution unrolls the list segment, because `a!=0` ensures that the list is not empty. Then, we check that the annotated action's precondition holds, namely `prev|->lk=0,tl=_w`. (Any variable starting with an underscore, such as `_w`, is an existential variable quantified across the pre- and post-condition of the action.) The checking procedure computes the leftover formula – the *frame* – obtained by removing cell `prev`. For this atomic block the frame is `lseg(_z,0)`. The frame is not used by the atomic block, and hence remains true at the exit of the atomic block.

Next (Step 2), we execute the body of the atomic block starting with the separate conjunction of the local state and the precondition of the action, so `prev==a * a!=0 * prev|->lk:0,tl:_z * _w=_z` in total. At the end, we get `prev==a * a!=0 * prev|->lk:TID,tl:_z * _w==_z`.

(Step 3) We try to prove that this assertion implies the postcondition of the action plus some local state. In this case, all the memory cells were consumed by the postcondition; hence, when exiting the atomic block, no local state is left.

(Step 4) So far, we have derived the postcondition `[prev|->lk=TID,tl=_z * lseg(_z,0)]`, but we have not finished. We must *stabilise* the postcondition to take into account the effect of other threads onto the resulting state. Following the fixed point computation of Section 2.3, we compute a weaker assertion that is stable under interference from all possible actions of other threads. In this case, the initial assertion was already stable.

Executing a read-only atomic block (b) The next atomic block only reads the shared state without updating it. Hence, we require no annotation, as this action causes no interference. Symbolic execution proceeds normally, allowing the code to access the shared state. Again, when we exit the region, we need to stabilise the derived post-condition.

Stabilisation (c) Next we illustrate how stabilisation forgets information. Consider unlocking the `prev` node within the loop. Just before unlocking `prev`, we have the shared assertion:

$$\boxed{\mathsf{lseg}(\mathtt{a},\mathtt{prev}) * \mathtt{prev}{\mapsto}(lk{=}\mathtt{TID}, tl{=}\mathtt{curr}) * \mathtt{curr}{\mapsto}(lk{=}\mathtt{TID}, tl{=}\_\mathtt{z}) * \mathsf{lseg}(\_\mathtt{z},0)}.$$

This says that the shared state consists of a list-segment from `a` to `prev`, two adjacent locked nodes `prev` and `curr`, and a list segment from `_z` to `nil`. Just after unlocking the node, before stabilisation, we get:

$$\boxed{\mathsf{lseg}(\mathtt{a},\mathtt{prev}) * \mathtt{prev}{\mapsto}(lk{=}0, tl{=}\mathtt{curr}) * \mathtt{curr}{\mapsto}(lk{=}\mathtt{TID}, tl{=}\_\mathtt{z}) * \mathsf{lseg}(\_\mathtt{z},0)}.$$

Stabilisation first forgets that $\mathtt{prev}{\rightarrow}lk = 0$, because another thread could have locked the node; moreover, it forgets that `prev` is allocated, because it could have been deleted by another thread. The resulting stable assertion is:

$$\boxed{\mathsf{lseg}(\mathtt{a},\mathtt{curr}) * \mathtt{curr}{\mapsto}(lk{=}\mathtt{TID}, tl{=}\_\mathtt{z}) * \mathsf{lseg}(\_\mathtt{z},0)}.$$

Local updates (d) Next we illustrate that local updates do not need to consider the shared state. Consider the code after the loop in `add`. As `temp` is local, the creation of the new cell and the two field updates affect only the local state. These commands cannot affect the shared state. Additionally, as `temp` is local state, we know that no other thread can alter it. Therefore, we get the following symbolic execution:

```
[a!=0 * lseg(a,prev) * prev|->lk=TID,tl=curr * lseg(curr,0)]
  temp = new();  temp->lk = 0;  temp->val = e; temp->tl = z;
[a!=0 * lseg(a,prev) * prev|->lk=TID,tl=curr * lseg(curr,0)]
              * temp|->lk=0,val=e,tl=curr
```

Transfering state from local to shared (e) Next we illustrate the transfer of state from local ownership to shared ownership. Consider the atomic block with the Add annotation:

```
[a!=0 * lseg(a,prev) * prev|->lk=TID,tl=curr * lseg(curr,0)]
      * temp|->lk=0,tl=curr
  atomic { prev->tl = temp } as Add(prev,temp);
[a!=0 * lseg(a,prev) * prev|->lk=TID,tl=temp
              * temp|->tl=curr * lseg(curr,0)]
```

We execute the body of the atomic block starting with the separate conjunction of the local state and the precondition of the action, so `prev|->lk=TID,tl=curr * temp|->lk=0,tl=curr` in total. At the end, we get `prev|->lk=TID,tl=temp * temp|->lk=0,tl=prev` and we try to prove that this implies the postcondition of the action plus some local state. In this case, all the memory cells were consumed by the postcondition; hence, when exiting the atomic block, no local state is left. Hence the cell `temp` is transferred from local state to shared state.

Transfering state from shared to local (f) This illustrates the transfer of state from shared ownership to local ownership, and hence that shared state can safely be disposed. Consider the atomic block with a `Remove` annotation.

```
[lseg(a,prev) * prev|->lk=TID,tl=curr
           * curr|->lk=TID,tl=temp * lseg(temp,0)]
  atomic {  prev->tl = temp; } as Remove(x,y);
[lseg(a,prev) * prev|->lk=TID,tl=temp * lseg(temp,0)]
           * curr|->lk=TID,tl=temp
```

Removing the action's precondition from the shared state leaves the frame `lseg(a,prev) * lseg(temp,0)`. Executing the body with the action's precondition gives `prev|->lk=TID,tl=temp * curr|->lk=TID,tl=temp` and we try to prove that this implies the postcondition of the action plus some local state. The action's postcondition requires `prev|->lk=TID,tl=temp`; so the remaining `curr|->lk=TID,tl=temp` is returned as local state. This action has taken shared state, accessible by every thread, and made it *local* to a single thread. Importantly, this means that the thread is free to dispose this memory cell, as no other thread will attempt to access it.

Program	LOC	LOA	Act	#Iter	#Prover calls	Mem(Mb)	Time (sec)
lock coupling	50	9	4	365	3879	0.47	3.9
lazy list	58	16	6	246	8254	0.70	13.5
optimistic list	59	13	5	122	4468	0.47	7.1
blocking stack	36	7	2	30	123	0.23	0.06
Peterson's	17	24	10	136	246	0.47	1.35

Fig. 3. Experimental results

```
[lseg(a,x) * x|->lk=TID,tl=z * lseg(z,0)] * y|->lk=TID,tl=z
  dispose(y);
[lseg(a,x) * x|->lk=TID,tl=z * lseg(z,0)]
```

Summary Our example has illustrated fine-grained locking, in particular

- dynamically allocated locks
- non-nested lock/unlock pairs
- disposal of memory (including locks)

Other examples we handle include optimistic reads from shared memory and lazy deletions.

4 Experimental Results

Our implementation is based on SmallfootRG, our extension of the separation logic tool called Smallfoot [4]. The tests were executed on a Powerbook G4 1.33 GHz with 786MB memory running OSX 10.4.8. The results are reported in Figure 3. For each example we report: the number of lines of code (LOC) and of annotation (LOA); the number of user-provided actions (Actions); the total number of iterations for all the fixpoint calculations for stabilisation (#Iter); the number of calls to the underlying theorem prover during stabilisation (#Prover calls); the maximum memory allocated during execution (Mem (Mb)), and the total execution time (Time (sec)).

We have tested our tool on a number of fine-grained concurrency examples. The first three (lock coupling, lazy list, optimistic list), taken from [27], all implement the data structure of a set as a singly linked list with a lock per node.

- `lock coupling` The main part of the algorithm was described in Section 3. When traversing the list, locks are acquired and released in a "hand over hand" fashion.
- `lazy list` An algorithm by Heller et al [13], which traverses the list without acquiring any locks; at the end it locks the relevant node and validates the node is still in the list. Deletions happen in two steps: nodes are first marked as deleted, then they are physically removed from the list.
- `optimistic list` Similar to lazy list, it traverses the list without acquiring any locks; at the end it locks the relevant node and re-traverses the list to validate that the node is still in the list.

The next two examples are simpler: `blocking stack` simply acquires a lock before modifying the shared stack; and `peterson` [22] is a well-known mutual exclusion algorithm.

We have a final example of Simpson's `4Slot` [26], which implements a wait-free atomic memory cell with a single reader and a single writer. This algorithm has been verified in both our tool, and Smallfoot. In our new tool it takes under 4 minutes, while in the original Smallfoot it took just under 25 minutes. Also, the specification of the invariant for Smallfoot is over twice as long as the action specification for our tool.[7]

Program	**Lines of Annotation**	**Time (sec)**
4Slot (our tool)	42	221
4Slot (Smallfoot)	80	1448

Smallfoot requires the same invariant about shared state at every program point. Our tool calculates all the pertinent shared states at each atomic block, so when it enters an atomic block it does not need to consider as many possibilities as Smallfoot.

Apart from the 4Slot algorithm, we believe our tool takes an acceptable amount of time to verify the algorithms discussed in this section. Our examples have demonstrated the disposal of memory (lock-coupling list and blocking stack), the optimistic reading of values and leaking memory (lazy and optimistic list algorithms), and classic mutual exclusion problems (Peterson's and Simpson's algorithm).

5 Conclusion and Related Work

The main challenge for automatic verification of thread-based concurrent programs is finding effective techniques to reason about the interference between threads. Fine-grained concurrency and deep heap updates exacerbate the problems. Several techniques have been proposed to support modular reasoning in the presence of concurrency. These include partial-order reduction [6], thread-modular model checking [11], assume-guarantee reasoning [15], and spatial separation with concurrent separation logic [18, 5, 21].

Flanagan, Freund, Qadeer and Seshia [9] have a tool, called Calvin, which uses rely/guarantee to reason about concurrent programs. It is built on top of ESC/Java [10]. Unlike our tool, they must check interference on every instruction as they do not have the dynamic partitioning between thread local and shared state.

The Zing [1] model checker has been used to verify some simple fine-grained concurrency examples [24]. The difficulty here is that the environment considered (*i.e.*, the inputs to the program and the actions of the surrounding environment) must be modelled concretely due to the fact that Zing is an *explicit-state* rather

[7] The specification for both could be simplified if Smallfoot directly supported arrays in the assertion language.

than symbolic model checker. In practice this means that the proofs in [24] consider only a limited subset of the potential set of inputs and environment actions.

Yahav and Sagiv [29] use shape analysis to verify a non-blocking queue algorithm. They provide specific instrumentation predicates for the test program in order to guide the automatic analyser. With our approach, the user specifies the atomic actions instead. Their tool does not attempt to decrease the level of interleaving. In principle, it should be possible to combine our technique with their generic analyser with the aim to reduce the number of shapes that they need to consider because of the interference between threads.

Our tool employs a new technique which takes aspects from both assume-guarantee and concurrent separation logic. We are not aware of any other tools doing that. We were also unable to find in the literature case studies of automatic[8] verification tools on a suite of fine-grained concurrent programs. Our work builds on mechanisms developed for program verification with separation logic [3], and the subsequent abstract interpretation techniques for local shape analysis [8].

We have demonstrated that our approach is effective for proving safety and data structure integrity of several list-manipulating algorithms with fine-grained concurrency. In the future, we want to consider other data structures, such as trees and arrays, and to push the barrier towards proving full correctness. We are also interested in inferring automatically the specifications of the actions operating on the shared state, perhaps borrowing from the ideas from thread modular shape analysis [12] or abstraction refinement [14]. More generally, we speculate that the combination of separation logic and rely/guarantee might help producing more effective verification tools for many more classes of concurrent programs.

Acknowledgements. We are grateful to the East London Massive for helpful discussions, and to Byron Cook and Alan Mycroft and Peter O'Hearn for comments on earlier drafts of the paper. We gratefully acknowledge the financial support of the EPSRC and Royal Academy of Engineering and a scholarship from the Cambridge Gates Trust.

References

1. T. Andrews, S. Qadeer, S. K. Rajamani, and Y. Xie. Zing: Exploiting program structure for model checking concurrent software. In *CONCUR*, pages 1–15, 2004.
2. M. Barnett, K.R.M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS*, volume 3362 of *LNCS*. Springer, 2004.
3. J. Berdine, C. Calcagno, and P.W. O'Hearn. Symbolic execution with separation logic. In *APLAS*, volume 3780 of *LNCS*. Springer, 2005.
4. J. Berdine, C. Calcagno, and P.W. O'Hearn. Smallfoot: Automatic modular assertion checking with separation logic. In *4th FMCO*, pages 115–137, 2006.

[8] An interactive proof of full correctness of the lazy list algorithm using PVS was presented in [7].

5. S. D. Brookes. A semantics for concurrent separation logic. In *CONCUR*, volume 3170 of *LNCS*, pages 16–34, London, August 2004. Springer. Extended version to appear in *Theoretical Computer Science*.
6. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
7. R. Colvin, L. Groves, V. Luchangco, and M. Moir. Formal verification of a lazy concurrent list-based set. In *18th CAV*, 2006.
8. D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *16th TACAS*, pages 287–302, 2006.
9. C. Flanagan, S. N. Freund, S. Qadeer, and S. A. Seshia. Modular verification of multithreaded programs. *Theor. Comput. Sci.*, 338(1-3):153–183, 2005.
10. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *PLDI*, pages 234–245, New York, NY, USA, 2002. ACM Press.
11. C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN*, 2003.
12. A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Thread-modular shape analysis. In *PLDI*, 2007.
13. S. Heller, M. Herlihy, V. Luchangco, M. Moir, B. Scherer, and N. Shavit. A lazy concurrent list-based set algorithm. In *9th OPODIS*, December 2005.
14. T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *15th CAV*, pages 262–274. LNCS 2725, Springer, 2003.
15. C. B. Jones. Specification and design of (parallel) programs. *IFIP Conference*, 1983.
16. J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
17. C. Morgan. The specification statement. In *ACM Trans. Program. Lang. Syst.*, volume 10(3), pages 403–419, 1988.
18. P. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, 2007. Preliminary version in CONCUR'04, LNCS 3170.
19. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *31st POPL*, pages 268–280, 2004.
20. S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Comm. ACM*, 19(5):279–285, 1976.
21. M. Parkinson, R. Bornat, and P. O'Hearn. Modular verification of a non-blocking stack. In *34th POPL*, 2007.
22. G. L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981.
23. A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13(1), 45–60, 1981.
24. S. Qadeer, S. K. Rajamani, and J. Rehof. Summarizing procedures in concurrent programs. *SIGPLAN Not.*, 39(1):245–255, 2004.
25. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th LICS*, pages 55–74. IEEE, 2002.
26. H. Simpson. Four-slot fully asynchronous communication mechanism. *IEE Proceedings*, 137(1):17–30, January 1990.
27. V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearisable objects. In *11th PPoPP*, pages 129–136. ACM, 2006.
28. V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In *18th CONCUR*, 2007.
29. E. Yahav and M. Sagiv. Automatically verifying concurrent queue algorithms. *Electr. Notes Theor. Comput. Sci.*, 89(3), 2003.