Compositional Resource Invariant Synthesis

Cristiano Calcagno[1], Dino Distefano[2], and Viktor Vafeiadis[3]

[1] Imperial College
[2] Queen Mary University of London
[3] Microsoft Research, Cambridge

Abstract. We describe an algorithm for synthesizing resource invariants that are used in the verification of concurrent programs. This synthesis employs bi-abductive inference to identify the footprints of different parts of the program and decide what invariant each lock protects. We demonstrate our algorithm on several small (yet intricate) examples which are out of the reach of other automatic analyses in the literature.

1 Introduction

Resource invariants are a popular thread-modular verification technique for concurrent lock-based programs. The idea is to associate with each lock a resource invariant, namely an assertion that is true whenever no thread has acquired the lock. When a lock is initialized, we must prove that the associated resource invariant holds. When a thread acquires a lock, it can assume that the corresponding resource invariant holds. When it releases the lock, it must prove that the resource invariant is still true.

In concurrent separation logic (CSL), O'Hearn [9] has adapted the notion of resource invariants by making them record exactly the part of the memory that a given lock protects. His elegant examples show how the ownership of memory cells can be transferred from one thread to another via a resource invariant. CSL provides simple proofs of programs such as the one in Fig. 1, where a memory cell is allocated in one thread and deallocated in another.

The central problem facing any attempt to construct CSL proofs automatically is the synthesis of suitable resource invariants. For instance, consider the two programs in Fig. 2 (taken from [9]) implementing a *one place pointer-transferring buffer*. In the first program, the memory cell `x` is transferred from the first thread to the second one, and can be easily verified once we have guessed the resource invariant $(\mathit{full} \wedge c \mapsto -) \vee (\neg \mathit{full} \wedge \mathsf{emp})$. In the second program, there is no transfer of ownership and the resource invariant is simply emp. To establish a proof for these programs the choice of the resource invariant must mirror the ownership property. O'Hearn does not address the issue of how to come up with the correct resource invariant and states that "ownership is in the eye of the asserter." This is also the approach taken by Smallfoot [3], which required the user to specify the resource invariants.

More recently, Gotsman et al. [7] proposed a very practical, heuristic method for calculating resource invariants. Their method is based on a thread-modular

```
put(x) = with buf when (!full) do {
           c := x; full := true;
         }

get(y) = with buf when (full) do {
           y = c; full = false;
         }
```

Fig. 1. `put(x)` and `get(y)` definitions.

```
    resource buf(c)
x = new();  ||get(y);
put(x);     ||dispose(y);

        resource buf(c)
x = new();              ||
put(x); dispose(x);     ||get(y);
```

Fig. 2. A single element buffer with ownership transfer (left). Without ownership transfer (right).

program analysis to compute resource invariants by a global fixpoint calculation. In order to decide which part of the memory is owned by a thread and which part belongs to a given lock, they use a predetermined reachability heuristic. The problem with this approach is that it relies heavily on an *ad hoc* local heuristic. For instance, in both programs of Fig. 2, at the end of the `put(x)` critical region, we have the state $\mathit{full} \wedge c = x \wedge c \mapsto -$. To verify the left program, we need to associate the memory cell $c \mapsto -$ to the resource. To verify the right program, the same memory cell must remain owned by the first thread. So, in general, the splitting cannot be decided by a purely local heuristic. Instead, the contexts of all conditional critical regions protecting the same resource need to be considered and therefore global methods are required.

In general, designing a method able to synthesize resource invariants in a *thread-modular* and *automatic* manner and susceptible to the ownership policy of the program is very tricky since ownership is a global property of the system. In this paper, we present an algorithm aiming at achieving this goal. Our method is not based on reachability but rather on the idea of *footprint* — i.e., the region of memory that a command requires in order to run safely. By employing the footprint concept, we obtain a more systematic way for computing resource invariants. We describe an algorithm that uses bi-abduction [4] to calculate what state is actually protected by the resource. We show the effectiveness of our algorithm by applying it to all the involved examples given by O'Hearn [9].

2 Informal description of the synthesis algorithm

Intuitively, our algorithm works by guessing an initial set of resource invariants and by iteratively refining the guess until either this is strong enough to prove the program or the algorithm gives up because it cannot find a better refinement of the current guess. More precisely, our algorithm can be described as follows:

1. For each Conditional Critical Region (CCR) in the system we take the empty heap as the initial approximation of the state protected by the resource.
2. The current guess of the Resource Invariants (RI) is used to compute specifications for all the CCRs. This step might refine the current RIs.

3. An attempt is made to prove each thread (separately) using the current guess of RIs and current specifications of CCRs. If a proof can be built, the algorithm stops with success (the current RIs are strong enough to prove memory safety). Otherwise, the current RIs are refined, as described below.
4. The refinement is done by applying bi-abduction [4] on the continuation of the CCR where the previous proof attempt failed. This is done to check whether the program involves ownership transfer.

Note that in step 3, in constructing a proof for the threads, we assume that the user annotates the program with both the association of variable names to resources and preconditions for the threads, but not the resource invariants (or loop invariants). We remark that the association of variables to resources can sometimes be discovered by a tool like Locksmith [10], and it seems likely that bi-abduction might be employed to discover these thread preconditions, just as it was used in [4] to discover procedure preconditions. So in applying our algorithm it is likely that an even greater degree of automation is possible. However, in this paper, we make these assumptions to focus our study on the core algorithmic difficulty of discovering the resource invariants.

3 Basics

3.1 Programming Language

We describe a simple parallel programming language following [9]. Let *Res* be a countable set of resource names. A concurrent program Prg in this language consists of an initialization phase where variables may be assigned a value, a single resource declaration, and a single parallel composition of sequential commands

$$\begin{array}{l}\mathsf{Prg} = \mathit{init};\\ \qquad \texttt{resource}\ r_1(\text{variable list}), \ldots, r_m(\text{variable list})\\ \qquad C_1 \parallel \cdots \parallel C_n\end{array}$$

Sequential commands are defined by the grammar:

$$\begin{array}{rl}\mathit{Comm} ::= & x := E \mid x := [y] \mid [x] := E \mid x := \texttt{new()} \mid \texttt{dispose(x)}\\ & \mid \texttt{skip} \mid C; C \mid \texttt{if}\ B\ \texttt{then}\ C\ \texttt{else}\ C \mid \texttt{while}\ B\ \texttt{do}\ C\ \texttt{endwhile}\\ & \mid \texttt{with}\ r\ \texttt{when}\ B\ \texttt{do}\ C\ \texttt{endwith}\end{array}$$

where $E \in \mathit{PVar} \cup \{\mathsf{nil}\}$ and PVar is a countable set of program variables ranged over by $x, y, z, \ldots$. Sequential commands include standard constructs (assignment, sequential composition, conditional, and iteration), dynamic allocation ($x :=$ `new()`), explicit deallocation (`dispose(x)`), and operations for accessing the heap: look-up ($x := [y]$) and mutation ($[x] := E$). Resources are accessed using CCR commands `with` r `when` B `do` C `endwith`, where B is a (heap-independent) boolean condition and C is a command. A CCR is a unit of mutual exclusion, therefore two `with` commands for the same resource cannot be executed simultaneously. In detail, `with` r `when` B `do` C `endwith` can be executed if the condition B is true and no other CCR for r is currently executing. Otherwise its execution is delayed until both conditions are satisfied.

Notation We introduce some notation used throughout the paper. Given a concurrent program Prg, let $CCR(\mathsf{Prg})$ denote the set of all its conditional critical regions. Let $Res(\mathsf{Prg})$ be the set of resources defined in Prg and let $CCR(r, \mathsf{Prg})$, with $r \in Res(\mathsf{Prg})$, be the subset of $CCR(\mathsf{Prg})$ acting on resource r. For $C = \texttt{with}\ r\ \texttt{when}\ B\ \texttt{do}\ C'\ \texttt{endwith}$, we define $guard(C) \stackrel{def}{=} B$, $body(C) \stackrel{def}{=} C'$ and $res(C) \stackrel{def}{=} r$ the guard, the body and the resource of the CCR C, respectively.

3.2 Storage Model and Symbolic Heaps

We describe the storage model and symbolic heaps: a fragment of separation logic formulae suitable for symbolic execution [2, 6]. Let $LVar$ (ranged over by $x', y', z', \dots$) be a set of logical variables, disjoint from program variables $PVar$, to be used in the assertion language. Let $Locs$ be a countably infinite set of locations, and let $Vals$ be a set of values that includes $Locs$. The storage model is given by:

$$Heaps \stackrel{def}{=} Locs \rightharpoonup_{\mathsf{fin}} Vals \qquad Stacks \stackrel{def}{=} (PVar \cup LVar) \to Vals$$
$$States \stackrel{def}{=} Stacks \times Heaps$$

Program states are symbolically represented by special separation logic formulae called *symbolic heaps*. They are defined as follows:

$$\begin{array}{lll}
E & ::= & x \mid x' \mid \mathsf{nil} \qquad\qquad \textit{Expressions} \\
\Pi & ::= & E{=}E \mid E{\neq}E \mid \mathsf{true} \mid \Pi \wedge \Pi \qquad \textit{Pure formulae} \\
S & ::= & E \mapsto E \mid \mathsf{ls}(E, E) \qquad \textit{Basic spatial predicates} \\
\Sigma & ::= & S \mid \mathsf{true} \mid \mathsf{emp} \mid \Sigma * \Sigma \qquad \textit{Spatial formulae} \\
D & ::= & \exists \boldsymbol{x'}.\, (\Pi \wedge \Sigma) \qquad \textit{Disjuncts} \\
H & ::= & D \mid H \vee H \qquad \textit{Symbolic heaps}
\end{array}$$

Expressions are program or logical variables x, x' or nil. Pure formulae are conjunctions of equalities and disequalities between expressions, and describe properties of variables. Spatial formulae specify properties of the heap. The predicate emp holds only in the empty heap where nothing is allocated. The formula $\Sigma_1 * \Sigma_2$ uses the separating conjunction of separation logic and holds in a heap h which can be split into two *disjoint parts* h_1 and h_2 such that Σ_1 holds in h_1 and Σ_2 in h_2. In symbolic heaps some (not necessarily all) logical variables are existentially quantified. The set of all symbolic heaps is denoted by SH. S is a set of basic spatial predicates. In this paper we consider a simple instance of S. However, our algorithm works equally well for other more sophisticated choices of spatial predicates such those described in [1, 5]. The *points-to* predicate $x \mapsto y$ denotes a heap with a single allocated cell at address x with content y, and $\mathsf{ls}(x, y)$ denotes a *non-empty* list segment from x to y (not including y).

3.3 Bi-Abduction

The notion of *bi-abduction* was recently introduced in [4]. It is the combination of two dual notions that extend the entailment problem: *frame inference* and

abduction. Frame inference [2] is the problem of determining a formula $\mathfrak{F}$ (called the *frame*) which we need to add to the conclusions of an entailment $H \vdash H' * \mathfrak{F}$ in order to make it valid. In other words, solving a frame inference problem means to find a description of the extra parts of heap described by H and not by H'. Abduction is dual to frame inference. It consists in determining a formula $\mathfrak{A}$ (called the *anti-frame*) describing the pieces of heap missing in the hypothesis and needed to make an entailment $H * \mathfrak{A} \vdash H'$ valid.

Bi-abduction is the combination of frame inference and abduction. It consists in deriving at the same time interdependent frames and anti-frames.

Definition 1 (Bi-Abduction). *Given two heaps H and H' find a frame $\mathfrak{F}$ and an anti-frame $\mathfrak{A}$ such that $H * \mathfrak{A} \vdash H' * \mathfrak{F}$*

Many solutions are possible for $\mathfrak{A}$ and $\mathfrak{F}$. A criterion to judge the quality of solutions as well as a bi-abductive prover were defined in [4]. A modified version of bi-abduction was proposed in [8].

Bi-abduction was introduced as a useful mechanism to construct compositional shape analyses. Such analyses can be seen as the attempt to build proofs for Hoare triples of a program. More precisely, given a program composed by procedures $p_1(\boldsymbol{x_1}), \ldots, p_n(\boldsymbol{x_n})$ the proof search automatically synthesizes preconditions $P_1, \ldots, P_n$ and postcondition $Q_1, \ldots, Q_n$ such that the following are valid Hoare triples:

$$\{P_1\}\, p_1(\boldsymbol{x_1})\, \{Q_1\}, \ldots, \{P_n\}\, p_n(\boldsymbol{x_n})\, \{Q_n\}$$

The triples are constructed by symbolically executing the program and by composing existing triples. The composition (and therefore the construction of the proof) is done in a bottom-up fashion starting from the leaves of the call-graph and then using their triples to build other proofs for procedures which are on a higher-level in the call-graph. To achieve that, the following special rule for sequential composition —called the Bi-Abductive Sequencing Rule [4]— is used

$$\frac{\{P_1\}\, C_1\, \{Q_1\} \qquad \{P_2\}\, C_2\, \{Q_2\}}{\{P_1 * \mathfrak{A}\}\, C_1; C_2\, \{Q_2 * \mathfrak{F}\}}\ Q_1 * \mathfrak{A} \vdash P_2 * \mathfrak{F} \tag{1}$$

In this paper we show that bi-abduction can be useful to achieve compositional proofs of concurrent programs.

Throughout this paper we will write the frame and anti-frame to be determined in the bi-abduction problem in "frak" fonts (e.g., $\mathfrak{A}, \mathfrak{F}, \mathfrak{B} \ldots$), this will help to distinguish them from the known parts of the entailment.

4 Comparing Resource Invariants

In this section we study the space of possible solutions to the resource invariant inference problem, and define an order used to compare those solutions. An optimal invariant w.r.t. that order always exists.

Safe Resource Invariants. Given a precondition P which holds before entering a CCR with body C (and ignoring the guard for simplicity), we say that I is a *safe resource invariant* starting from P if the following Hoare triple holds

$$\{P * I\}\, C\, \{I * \mathsf{true}\}$$

In other words, I describes resource large enough for C to execute safely, yet I is weak enough that C can re-establish it. For example, $x \mapsto 3$ is too strong if C is $[x] := 4$ (cannot be re-established), and emp does not describe enough resource for C to execute safely. Perhaps surprisingly, these two requirements are compatible with an order relation which admits an optimal solution, which we describe below.

Best Resource Invariant. If I and I' are resource invariants, we define the pre-order $I \leq I'$, meaning that I is *better* (or *smaller*) than R', as follows:

$$I \leq I' \iff I' \models I * \mathsf{true}$$

When $I \leq I'$ we sometimes say that I' *extends* I. Note that $\leq$ is not antisymmetric as $I \leq I'$ and $I' \leq I$ does not imply $I = I'$. However, it implies $\mathsf{min}(I) = \mathsf{min}(I')$, where min is an operation that removes non-minimal states, defined as follows: $(s,h) \models \mathsf{min}(X) \iff (s,h) \models X$ and $\forall h'.\ s,h' \models X$ implies $h \leq h'$.

Therefore, $\leq$ is antisymmetric modulo the equivalence relation $I \sim I' \iff \mathsf{min}(I) = \mathsf{min}(I')$. For example, $\mathsf{emp} \leq \mathsf{true}$ and $\mathsf{true} \leq \mathsf{emp}$, but $\mathsf{min}(\mathsf{emp}) = \mathsf{min}(\mathsf{true}) = \mathsf{emp}$. We call emp the canonical representative of the equivalence class.

Notice that if I_1 and I_2 are safe resource invariants starting from P, then so is $I_1 \vee I_2$, by direct application of Hoare's disjunction rule. Since $I' \Rightarrow I$ implies $I \leq I'$, it can be readily seen that a (unique modulo $\sim$) minimal resource invariant I_{best} exists, and can be described directly as

$$I_{\text{best}} \stackrel{def}{=} \bigvee_{I \text{ r.i. for all CCR's}} I$$

Hence the best invariant is logically weakest and spatially smallest. The goal of the invariant synthesis algorithm is to find the minimal I which is an invariant for all the CCR's in the program and allows to prove the program race free.

5 The Invariant Synthesis Algorithm

Algorithm 1 computes the set $\mathcal{I}$ of resource invariants for the program Prg or returns failure. $\mathcal{I}$ is a function $\mathcal{I} : \mathit{Res} \to \mathsf{SH}$ associating to each resource r a resource invariant $\mathcal{I}(r)$. The basic idea is to start with the minimal invariant emp and then repeatedly refine it to a bigger one w.r.t. $\leq$ during symbolic execution. The role of (perfect) abduction is to refine it by the *minimum* amount necessary for the symbolic execution to go through. So the informal argument for each

Algorithm 1 InvariantSynthesis(Prg)

1: $\mathcal{I} := \{(r, \bigvee_{C_r \in CCR(r,\mathsf{Prg})}(\mathsf{emp} \wedge \mathit{guard}(C_r))) \mid r \in \mathit{Res}(\mathsf{Prg})\}$;
2: $\mathit{Failed} := \emptyset$;
3: **while** $\mathcal{I} \notin \mathit{Failed}$ **do**
4: $(\mathit{Specs}, \mathcal{I}) := \mathit{CompSpecs}(\mathcal{I})$;
5: **if** $\mathit{ProofSearch}(\mathsf{Prg}, \mathcal{I}, \mathit{Specs})$ fails **then**
6: $\mathit{Failed} := \mathit{Failed} \cup \{\mathcal{I}\}$
7: $C_1; \cdots; C_j := \mathit{FailingPath}(\mathsf{Prg}, \mathcal{I}, \mathit{Specs})$;
8: $\mathcal{I} := \mathsf{RefineOwnership}(C_1; \cdots; C_j, \mathcal{I})$;
9: **else**
10: **return** $\mathcal{I}$
11: **end if**
12: **end while**
13: **return** failure

refinement from I to I' is of the form "if there exists a safe invariant, it must be $\geq I'$". The initial approximation emp models a situation where resources are neither protected nor transferred; only if the program requires it, is the invariant refined into one which does so. More precisely, the basic idea is implemented as follows. Initially the resource invariant of every resource r is initialized to be a disjunction of emp and the guard of its CCRs (Step 1).[1] This gives the first approximation for $\mathcal{I}$. *Specs* is the set of Hoare triples $\{P\}\, C\, \{Q\}$ defining a specification for all CCRs in the program. *Specs* is computed by using the function *CompSpecs* which is applied the current guess of the invariants. *CompSpecs* is explained in detail in Sec. 5.1, and while it generates specifications it may modify $\mathcal{I}$ giving a first refinement. *CompSpecs* returns a set of pairs $(\mathit{Specs}, \mathcal{I}')$ or fails. $\mathit{ProofSearch}(\mathsf{Prg}, \mathcal{I}, \mathit{Specs})$ (see Sec. 5.2) is a procedure that tries to build a separation logic proof of Prg using the specifications *Specs* and the resource invariants $\mathcal{I}$. The set *Failed* contains those invariants for which the algorithm failed to build a proof. The loop starting at step 3 attempts to build a proof with the result of *CompSpecs*. If the proof succeeds, the algorithm terminates with success and returns the computed resource invariants. Otherwise, the algorithm tries to refine the current guess. In that case, the invariant of the failing CCR is refined using the procedure RefineOwnership (see Sec. 5.3). After $\mathcal{I}$ is refined the set of CCR specifications is updated accordingly before attempting a new proof of the program. The algorithm fails in case the refinement process returns an invariant which was tried before with no success. Notice that *CompSpecs* is a partial function, therefore, the algorithm fails also in case *CompSpecs* does not return a value.

[1] The rationale for adding CCRs' guards to the initial invariant is that, when the algorithm refines $\mathcal{I}(r)$ by examining a CCR C_r, the missing part will be added only to the disjunct corresponding to C_r. This disjunct is determined by $\mathit{guard}(C_r)$. Adding $*$-conjuncts only to one disjunct (rather than to all of them) provides us with a better invariant w.r.t. the defined order $\leq$.

5.1 Computing Specifications for CCRs

The computation of CCRs' specifications requires an *abstraction function* for symbolic heaps $\alpha : \mathsf{SH} \longrightarrow \mathsf{SH}$. Given the kind of symbolic heaps used in this paper, it is enough to have α defined as in [6], although our algorithm is not dependent on a specific choice. Moreover, let $[P]^{loc}_{Q}$ be a function that replaces shared variables[2] in P using equalities in Q. $[\cdot]^{loc} : \mathsf{SH}\times\mathsf{SH} \longrightarrow \mathsf{SH}$ is defined as:

$$[P]^{loc}_{Q} = P[x_1/c_1, \cdots, x_n/c_n]$$

where x_i are local variables, c_i are shared variables, and $Q \equiv x_1 = c_1 \wedge \cdots \wedge x_n = c_n \wedge Q'$ and in Q' there are no further equality terms between local and shared variables.[3] Similarly, define $[\cdot]^{sha}$ as the dual function which tries to replace local variables with shared variables.

Computing the specification of a single CCR. The computation of a specification for the CCR `with` r `when` B `do` C `endwith` is done by performing a compositional bottom-up analysis ([4] and Sec. 3.3) on the body C. The analysis starts from the following precondition: $B \wedge \mathsf{emp} * \mathcal{I}(r)$.

This is different from [4], where the analysis started with precondition emp. The bottom-up analysis will construct a proof of C by synthesizing P and Q such that the triple

$$\{B \wedge P * \mathcal{I}(r)\}\, C\, \{Q\} \tag{2}$$

holds.[4] Once (2) is computed, a specification for the `with` command is obtained by applying the following new rule (called $\mathsf{BA\text{-}with}$):

$$\frac{\{B \wedge (P * \mathcal{I}(r))\}\, C\, \{Q\}}{\{P * [\mathfrak{A}]^{loc}_{Q}\}\ \texttt{with}\ r\ \texttt{when}\ B\ \texttt{do}\ C\ \texttt{endwith}\ \{\alpha(\exists \boldsymbol{c}.\mathfrak{F})\}}\ Q * \mathfrak{A} \vdash \mathcal{I}(r) * \mathfrak{F}$$

with additional side conditions:

1. no variable occurring free in $[\mathfrak{A}]^{loc}_{Q}$ is modified by C,
2. no other process modifies variables free in $P * [\mathfrak{A}]^{loc}_{Q}$ or $\alpha(\exists \boldsymbol{c}.\mathfrak{F})$.

Starting from a proof of the CCR's body (2), this rule uses bi-abduction to derive two symbolic heaps $\mathfrak{A}$ and $\mathfrak{F}$. The anti-frame $\mathfrak{A}$ needs to be added to the precondition P to re-establish r's resource invariant $\mathcal{I}(r)$. The frame $\mathfrak{F}$ corresponds to the postcondition of the `with` statement. Both frame and anti-frame are massaged before using them in the specification to remove terms related to shared variables. In particular in the anti-frame $\mathfrak{A}$, terms containing shared

[2] Shared variables are those listed in the resources declaration (see Sec 3.1).

[3] $[\cdot]^{loc}$ is a well defined function if a fixed order among local variables is chosen.

[4] The reason for not using a simple forward symbolic execution starting from $\mathsf{emp} * \mathcal{I}(r) \wedge B$ to build a proof of C is that, in general, this precondition is not enough for proving C. Hence a precondition $P \neq \mathsf{emp}$ needs to be derived, and this is done by the bottom-up analysis.

$$\frac{\mathit{Spec}(\mathcal{I}, C_r) = \{P\}\, C_r\, \{Q\}}{L, \mathcal{I} \longrightarrow_1 L, \mathcal{I}[r \leftarrow I_r]} \quad \begin{array}{l} \mathit{Shared}(P) \neq \emptyset \text{ and } \mathcal{I}(r) \leq I_r \text{ and } C_r \notin L \\ I_r = \alpha(\mathcal{I}(r) * P_{\mathit{Shared}}) \end{array}$$

$$\frac{}{L, \mathcal{I} \longrightarrow_2 L, \mathcal{I}[r \leftarrow \bigvee_{i \in X} D_i]} \quad \begin{array}{l} \mathcal{I}(r) = D_1 \vee \ldots \vee D_n \\ X \subseteq \{i \mid 1 \leq i \leq n\} \end{array}$$

$$\frac{\mathit{Spec}(\mathcal{I}, C_r) = \{P\}\, C_r\, \{Q\}}{L, \mathcal{I} \longrightarrow_3 \{C_r\} \cup L, \mathcal{I}} \quad \mathit{Shared}(P) = \emptyset \text{ and } C_r \notin L$$

Table 1. Transition rules for computing *Specs* and (possibly) refine $\mathcal{I}$.

variables are rewritten (when possible) in terms of local variables using known equalities in Q. The frame $\mathfrak{F}$ is simplified by replacing uses of shared variables by local variables whenever possible using the existing equalities, and by dropping pure formulae involving shared variables. This is achieved by first existentially quantifying shared variables in $\mathfrak{F}$ and then applying the abstraction α.

Lemma 1. *The* BA-with *rule is sound.*

The function CompSpecs. The computation of specifications for all the CCRs in the program is performed by *CompSpecs*. Given a set of resource invariants $\mathcal{I}$, this function has type

$$\mathit{CompSpecs} : (\mathit{Res} \to \mathsf{SH}) \longrightarrow \mathcal{P}(\mathsf{SH} \times \mathit{Comm} \times \mathsf{SH}) \times (\mathit{Res} \to \mathsf{SH})$$

and it is defined as

$$\begin{array}{c} \mathit{CompSpecs}(\mathcal{I}) = (\{\mathit{Spec}(\mathcal{I}', C_r) \mid C_r \in \mathit{CCR}(\mathsf{Prg})\}, \mathcal{I}') \\ \text{when } (\emptyset, \mathcal{I}) \longrightarrow_1^* \longrightarrow_2^* \longrightarrow_3^* (\mathit{CCR}(\mathsf{Prg}), \mathcal{I}') \end{array}$$

This definition uses the transition rules in Table 1 in three distinct phases: invariant refinement ($\longrightarrow_1^*$), pruning of disjuncts ($\longrightarrow_2^*$), and checking of the result ($\longrightarrow_3^*$). Let $\mathit{Shared}(P)$ be the set of shared variables occurring in P, and let P_{Shared} be the sub-formula of P containing only shared variables. The invariant of the $\longrightarrow_\_$ rules is that the first component L contains CCRs for which specifications have been (successfully) computed so far. The rules are applied to $\mathit{CCR}(\mathsf{Prg}) \setminus L$ and $\mathcal{I}$ until a specification has been computed for all CCRs. The function $\mathit{Spec}(\mathcal{I}, C_r)$ computes the specification for the CCR C_r w.r.t. $\mathcal{I}$ as described above (i.e., using bottom-up analysis and BA-with). The rule $\longrightarrow_1$ builds a tentative triple $\{P\}\, C_r\, \{Q\}$. The side condition checks the integrity of the obtained precondition P by verifying whether it requires heap described by shared variables. In this case r's resource invariant is not strong enough. The rationale is that if shared state is needed by the critical region this should be provided by the resource invariant and not by the precondition.[5] The rule therefore tries to refine $\mathcal{I}(r)$ by adding the terms with shared variables in P. If the

[5] Recall that precondition computed by bi-abduction corresponds to the footprint of C, therefore it expresses the state needed to run the command.

resulting invariant I_r extends the current guess for r, then this extension is used to replace $\mathcal{I}(r)$. Rule $\longrightarrow_2$ can be applied when $\longrightarrow_1$ cannot refine $\mathcal{I}(r)$ any further. The task of $\longrightarrow_2$ is to remove from $\mathcal{I}(r)$ those disjuncts that cannot be re-established by the CCR's body. Finally, rule $\longrightarrow_3$ records the fact that a spec for C_r has been found, by extending the set L of completed CCRs.

Lemma 2. *If the number of program variables in* Prg *is finite then the transition system defined in Table 1 is finite.*

The immediate consequence of this lemma is that *CompSpecs* can be effectively computed by a fixed-point computation which applies systematically the rules avoiding to re-apply them to previously visited states. Hence we have:

Corollary 1. *The computation of CompSpecs terminates.*

We now illustrate with some examples the derivation of specifications for CCRs.

Example 1. Assume the resource invariant $I \equiv (\neg \mathit{full} \wedge \mathsf{emp}) \vee (\mathit{full} \wedge \mathsf{emp})$. We show the induced specifications for the CCRs in Fig. 1. Using emp as precondition, for `put(x)` we have the following derivation:

$$\begin{array}{l} \{\neg \mathit{full} \wedge \mathsf{emp} * I\} \\ c := x; \\ \{\neg \mathit{full} \wedge c{=}x \wedge \mathsf{emp}\} \\ \mathit{full} := \mathit{true}; \\ \{\mathit{full} \wedge c{=}x \wedge \mathsf{emp}\} \end{array}$$

That is we have the triple $\{\neg \mathit{full} \wedge \mathsf{emp} * I\}\, c := x; \mathit{full} := \mathit{true}\, \{\mathit{full} \wedge c{=}x \wedge \mathsf{emp}\}$. From this, the bi-abduction engine is queried to derive $\mathfrak{F}$ and $\mathfrak{A}$ for the entailment

$$\mathit{full} \wedge c{=}x \wedge \mathsf{emp} * \mathfrak{A} \vdash I * \mathfrak{F}$$

The solution is $\mathfrak{A} \equiv \mathsf{emp}$ and $\mathfrak{F} \equiv c{=}x \wedge \mathsf{emp}$. This is further simplified to remove terms with shared variables: $[\mathsf{emp}]^{loc}_{c=x \wedge I} = \mathsf{emp}$ and $\alpha(\exists c.\, c{=}x \wedge \mathsf{emp}) = \mathsf{true} \wedge \mathsf{emp}$. Therefore by the application of the rule BA-with we obtain the specification

$$\{\mathsf{emp}\}\, \texttt{put(x)}\, \{\mathsf{emp}\}$$

Similarly for the CCR `get(y)`, using emp as precondition of BA-with we have:

$$\begin{array}{l} \{\mathit{full} \wedge \mathsf{emp} * I\} \\ y := c; \mathit{full} := \mathit{false} \\ \{\neg \mathit{full} \wedge y = c \wedge \mathsf{emp}\} \end{array}$$

Now we appeal to bi-abduction for the query $\neg \mathit{full} \wedge y = c \wedge \mathsf{emp} * \mathfrak{A} \vdash I * \mathfrak{F}$. The solution is $\mathfrak{A} \equiv \mathsf{emp}$ and $\mathfrak{F} \equiv y{=}c \wedge \mathsf{emp}$ and hence after the simplification of $[\cdot]^{loc}$ and α and by BA-with rule we obtain the specification

$$\{\mathsf{emp}\}\, \texttt{get(y)}\, \{\mathsf{emp}\}.$$

```
alloc(x) = with mm when (true) do {
             if (f=nil) then x := new();
             else x := f; f:=[x];
           }
```

```
dealloc(y) = with mm when (true) do {
               [y] := f; f:= y;
             }
```

Fig. 3. `alloc(x)` and `dealloc(y)` definitions.

Example 2. Consider now a different resource invariant $I \equiv (\neg full \wedge \mathsf{emp}) \vee (full \wedge c \mapsto -)$. As in the previous example, we show the induced specifications for the CCRs in Fig. 1, using this invariant instead. For `put(x)` we can derive:

$$\begin{array}{l} \{\neg full \wedge \mathsf{emp} \wedge I\} \\ c := x; full := true; \\ \{full \wedge c{=}x \wedge \mathsf{emp}\} \end{array}$$

That is we have the triple $\{\neg full \wedge \mathsf{emp} * I\}\, c := x; full := true\, \{c{=}x \wedge full \wedge \mathsf{emp}\}$. The bi-abduction engine derives for the question $c{=}x \wedge full \wedge \mathsf{emp} * \mathfrak{A} \vdash I * \mathfrak{F}$ the solution $\mathfrak{A} \equiv c \mapsto -$ and $\mathfrak{F} \equiv c{=}x \wedge \mathsf{emp}$. By simplifying the anti-frame we obtain $[c \mapsto -]^{loc}_{c=x \wedge full} = x \mapsto -$ whereas for the frame we have $\alpha(\exists c.\, c{=}x \wedge \mathsf{emp}) = \mathsf{true} \wedge \mathsf{emp}$. Therefore the application of BA-with gives the specification $\{x \mapsto -\}\, \mathtt{put(x)}\, \{\mathsf{emp}\}$.

Similarly for `get(y)` we have:

$$\begin{array}{l} \{full \wedge \mathsf{emp} * I\} \\ \{full \wedge c \mapsto -\} \\ y := c; full := false \\ \{\neg full \wedge y{=}c \wedge c \mapsto -\} \end{array}$$

When posed the query $\neg full \wedge y{=}c \wedge c \mapsto - * \mathfrak{A} \vdash I * \mathfrak{F}$ the bi-abduction engine finds the solutions $\mathfrak{A} \equiv \mathsf{emp}$ and $\mathfrak{F} \equiv y{=}c \wedge c \mapsto -$. $\mathfrak{A}$ is already simplified, whereas $\mathfrak{F}$ is simplified to $\alpha(\exists c.\, y{=}c \wedge c \mapsto -) = y \mapsto -$. Hence BA-with returns the specification $\{\mathsf{emp}\}\, \mathtt{get(y)}\, \{y \mapsto -\}$.

Example 3. We now consider a more involved example that shows the computation of the function *CompSpecs*. Here we use the memory manager described in [9] and reported in Fig. 3. We start by computing the specification of `alloc(x)` using $I_0 \equiv \mathsf{true} \wedge \mathsf{emp}$. We can prove the triple

$$\{P_0\}\, \mathtt{alloc(x)}\, \{x \mapsto -\}$$

where $P_0 \equiv (f{=}\mathsf{nil} \wedge \mathsf{emp}) \vee (f \mapsto -)$. However, the precondition specifies properties of the shared variable f, so we need to apply rule $\longrightarrow_1$ of Table 1. The invariant is refined by adding P_0 to the current I_0 and then abstraction α:

$$I_1 = \alpha(I_0 * P_0) = (f{=}\mathsf{nil} \wedge \mathsf{emp}) \vee (f \mapsto f')$$

where we have explicitly named the existential variable f' because it will be used in the next iteration. When recomputing the specification of `alloc(x)` using I_1 we obtain the following triple:

$$\{P_1\}\,\texttt{alloc(x)}\,\{x \mapsto -\}$$

where $P_1 \equiv (f{=}\mathsf{nil} \wedge \mathsf{emp}) \vee (f{\neq}\mathsf{nil} \wedge f'{=}\mathsf{nil} \wedge \mathsf{emp}) \vee (f{\neq}\mathsf{nil} \wedge f' \mapsto -)$. Again by rule $\longrightarrow_1$ we obtain

$$\begin{aligned} I_2 = \alpha(I_1 * P_1) &= \alpha((f{=}\mathsf{nil} \wedge \mathsf{emp}) \vee (f \mapsto \mathsf{nil}) \vee (f \mapsto f' * f' \mapsto -)) \\ &= (f{=}\mathsf{nil} \wedge \mathsf{emp}) \vee (f \mapsto \mathsf{nil}) \vee \mathsf{ls}(f, f') \end{aligned}$$

A further iteration of $\longrightarrow_1$ produces the same P_1 and

$$I_3 = (f{=}\mathsf{nil} \wedge \mathsf{emp}) \vee (f \mapsto \mathsf{nil}) \vee \mathsf{ls}(f, f') \vee \mathsf{ls}(f, \mathsf{nil})$$

The candidate I_3 is a fixpoint w.r.t. $\longrightarrow_1$ but it still produces the same P_1, therefore rule $\longrightarrow_3$ cannot be applied yet. This is caused by the disjunct $\mathsf{ls}(f, f')$, which is too weak: starting from $\mathsf{ls}(f, f')$ the candidate invariant I_3 cannot be re-established. But now, rule $\longrightarrow_2$ can fire to remove disjunct $\mathsf{ls}(f, f')$ and obtain

$$I_3 \longrightarrow_2 I_4 = (f{=}\mathsf{nil} \wedge \mathsf{emp}) \vee (f \mapsto \mathsf{nil}) \vee \mathsf{ls}(f, \mathsf{nil})$$

Now rule $\longrightarrow_3$ can be applied, so I_4 is a resource invariant for $\texttt{alloc}(x)$. The final specification of `alloc(x)` using I_4 is $\{\mathsf{emp}\}\,\texttt{alloc(x)}\,\{x \mapsto -\}$.

Finally, I_4 directly allows us to obtain $\{y \mapsto -\}\,\texttt{dealloc(y)}\,\{\mathsf{emp}\}$ as specification for `dealloc(y)`.

5.2 Proof Search

This phase attempts to build a compositional proof of the program by trying to prove each thread in isolation. The building process is done using the bottom-up analysis which starts from the beginning of the thread and tries to construct a valid Hoare triple by symbolically executing the program as described in Sec. 3.3. Let the concurrent program be

$$\mathsf{Prg} = \mathit{init};\ \texttt{resource}\ r_1(\boldsymbol{x_1}), \ldots, r_m(\boldsymbol{x_m});\ C_1 \parallel \cdots \parallel C_n$$

Given P_{C_i}, a precondition for the thread C_i we can execute a proof search for C_i by *ProofSearch*. This procedure uses the Bi-Abductive Sequencing rule to build the proof but requires that at every application of this rule we have $\mathfrak{A} \equiv \mathsf{emp}$. This condition ensures that a proof for the thread C_i can actually be built from the precondition P_{C_i}. In fact, it provides us with a notion of failure for a proof attempt. We say that the proof search for $C_i = C_i'; C_i''$ (from P_{C_i}) *fails* if by an application of Bi-Abductive Sequencing Rule we obtain the triple

$$\{P_{C_i} * \mathfrak{A}\}\,C_i'\,\{Q\}$$

Algorithm 2 RefineOwnership$(C_1; \ldots; C_j; C, \mathcal{I})$

1: $\rho = \{i \in [1, j] \mid C_i \text{ is a CCR}\}$;
2: **do**
3: $\quad k := \max \rho$;
4: $\quad \rho := \rho \setminus \{k\}$
5: $\quad I' := \mathsf{RefOwn}((C_1; \cdots; C_k), (C_{k+1}; \cdots; C_j; C))$
6: **while** $\mathcal{I}(res(C_k)) = I' \ \wedge \ \rho \neq \emptyset$;
7: **return** $\mathcal{I}[res(C_k) \leftarrow I']$;

for some $Q \in \mathsf{SH}$ and $\neg(\mathfrak{A} \equiv \mathsf{emp})$. We are usually interested in the shortest prefix C'_i which makes the proof fail. The synthesis algorithm uses this notion of failure to detect when and where the invariant needs to be refined because of possible ownership transfer.[6]

5.3 Refining Resource Invariants for ownership transfer

Algorithm 2 defines the procedure RefineOwnership. It is called by InvariantSynthesis when the proof search fails, typically because some ownership transfer is needed for the program to be safe, but is not enabled by the current invariants $\mathcal{I}$. RefineOwnership takes as parameter a sequence of commands containing a CCR for which a proof attempt has failed. Consider the sequence $C_1; \cdots; C_j; C$ where the failure of the proof occurred in C. Let $\rho \subseteq [1, j]$ be the indexes of all the CCRs in the sequence. The algorithm starts from the last CCR, i.e. C_k where $k = \max \rho$, and tries to refine its invariant using function RefOwn. If no refinement is possible (i.e. the invariant remains unchanged), then the algorithm tries to refine the invariant of the previous CCR in the sequence, and so on until no further CCR exists.

We now describe how the function $\mathsf{RefOwn}((\hat{C}; C_r), \hat{C}')$ operates, where $C_r \equiv$ `with` r `when` B `do` C'' `endwith` is the CCR whose invariant will be refined, and the $\hat{C}$ notation is used for sub-sequences of the failing sequence. Let P be the precondition of the current thread, and let $\{P\}\,\hat{C}\,\{Q\}$ the result of the forward analysis just before C_r and $\{B \wedge (Q * \mathcal{I}(r))\}\,C''\,\{Q'' * \mathcal{I}(r)\}$ the results of forwards analysis until before exiting the CCR C_r. Let also $\{P'\}\,\hat{C}'\,\{Q'\}$ be the result of spec inference for the continuation $\hat{C}'$. We can then define

$$\mathsf{RefOwn}((\hat{C}; C_r), \hat{C}') \stackrel{def}{=} ((B \wedge [\mathfrak{A}]^{sha}_{(Q'' * \mathcal{I}(r))}) \vee (\neg B \wedge \mathsf{emp})) * \mathcal{I}(r) \quad \text{if } (Q'' * \mathcal{I}(r)) * \mathfrak{A} \vdash (P' * \mathcal{I}(r)) * \mathfrak{F}$$

where recall that $[\cdot]^{sha}$, defined in Sec. 5.1, tries to replace local variables with shared variables.

Intuitively RefOwn takes a trace ending in a CCR C_r and its continuation $\hat{C}'$, and returns a refined resource invariant for r which is updated only for

[6] Clearly the proof can fail for other reasons than the resource invariant. Other issues for failure can be manifested in the fact that $\neg(\mathfrak{A} \equiv \mathsf{emp})$.

the part related to C_r and which takes into account the heap needed by $\hat{C}'$. The refinement is computed by solving a bi-abduction question involving the symbolic state inside C_r before releasing the invariant, and the precondition of the continuation suitably augmented with the invariant. In addition, only the part of the anti-frame $\mathfrak{A}$ involving shared variables is taken to refine the invariant.

Soundness and Termination. We now give some results about our invariant generation method. For space reasons the proofs are reported in the appendix.

Theorem 1. *The* InvariantSynthesis *algorithm is sound.*

Corollary 2. *If* InvariantSynthesis(Prg) *returns a set* $\mathcal{I}$ *then* Prg *is race-free.*

Theorem 2. *Algorithm 1* InvariantSynthesis *terminates provided that the underlying forward analysis does.*

5.4 Full Examples

Example 4. We describe the execution of the synthesis algorithm on the program on the left side of Fig. 2 which performs transfer of ownership. The first approximation of the resource invariant for resource *buf* is $I_0 = I_{put} \vee I_{get}$ where

$$I_{put} = \neg \mathit{full} \wedge \mathsf{emp} \qquad I_{get} = \mathit{full} \wedge \mathsf{emp} \tag{3}$$

Using I_0 we obtain the first approximation of `put(x)` and `get(y)` specifications (see Ex. 1 for the detailed derivation of these specs):

$$\{\mathsf{emp}\}\,\texttt{put(x)}\,\{\mathsf{emp}\} \qquad \{\mathsf{emp}\}\,\texttt{get(y)}\,\{\mathsf{emp}\} \tag{4}$$

We then execute the *ProofSearch* procedure of both threads using I_0 and emp as preconditions. By Bi-Abductive Sequencing Rule (1) for the LHS thread we have:

$$\frac{\{\mathsf{emp}\}\,x = \texttt{new()}\,\{x \mapsto -\} \qquad \{\mathsf{emp}\}\,\texttt{put(x)}\,\{\mathsf{emp}\}}{\{\mathsf{emp}\}\,x = \texttt{new()};\texttt{put(x)}\,\{x \mapsto -\}}$$

by taking $\mathfrak{A} \equiv \mathsf{emp}$ and $\mathfrak{F} \equiv x \mapsto -$. Since $\mathfrak{A}$ is emp, no refinement of I is required and this completes the proof of the LHS thread. For the RHS we have:

$$\frac{\{\mathsf{emp}\}\,\texttt{get(y)}\,\{\mathsf{emp}\} \qquad \{y \mapsto -\}\,\texttt{dispose(y)}\,\{\mathsf{emp}\}}{\{y \mapsto -\}\,\texttt{get(y)};\texttt{dispose(y)}\,\{\mathsf{emp}\}} \tag{5}$$

However, we obtain this derivation by the anti-frame $\mathfrak{A} \equiv y \mapsto -$, and by our notion of failure of the proof search introduced in Sec. 5.2 this means that we cannot actually prove the RHS thread. The algorithm starts the refinement of the invariant by inspecting the RHS and using the body of the CCR `get(y)`:[7]

$$\{(c = c' \wedge y = y' \wedge \mathsf{emp}) * (\mathit{full} \wedge I_0)\}\,\texttt{y=c;full=false}\,\{c = c' \wedge y = c' \wedge \neg \mathit{full} \wedge \mathsf{emp}\}$$

[7] As in [4], we use auxiliary variables to record the initial value of program variables.

According to the definition of RefOwn we have to solve

$$(c = c' \wedge y = c' \wedge \neg \mathit{full} \wedge \mathsf{emp}) * \mathfrak{A} \vdash (I_0 * y \mapsto -) * \mathfrak{F}$$

Here we have $\mathfrak{A} \equiv y \mapsto -$ and $[\mathfrak{A}]^{sha}_{(c=c' \wedge y=c' \wedge \neg \mathit{full} \wedge \mathsf{emp})} \equiv c \mapsto -$. Following the algorithm, we extend the *full* disjunct of I_0 to obtain a new candidate invariant:

$$I_1 = (\neg \mathit{full} \wedge \mathsf{emp}) \vee (\mathit{full} \wedge c \mapsto -) \tag{6}$$

CompSpecs then updates the specifications for `put(x)` and `get(y)` using the new invariant and the rule BA-with. As shown in Ex. 2 we obtain:

$$\{x \mapsto \}\, \mathtt{put(x)}\, \{\mathsf{emp}\} \qquad\qquad \{\mathsf{emp}\}\, \mathtt{get(y)}\, \{y \mapsto -\} \tag{7}$$

The algorithm then uses the new specs in an attempt to prove LHS and RHS.

$$\frac{\{\mathsf{emp}\}\, x = \mathtt{new()}\, \{x \mapsto -\} \qquad \{x \mapsto -\}\, \mathtt{put(x)}\, \{\mathsf{emp}\}}{\{\mathsf{emp}\}\, x = \mathtt{new()}; \mathtt{put(x)}\, \{\mathsf{emp}\}}$$

$$\frac{\{\mathsf{emp}\}\, \mathtt{get(y)}\, \{y \mapsto -\} \qquad \{y \mapsto -\}\, \mathtt{dispose(y)}\, \{\mathsf{emp}\}}{\{\mathsf{emp}\}\, \mathtt{get(y)}; \mathtt{dispose(y)}\, \{\mathsf{emp}\}}$$

This time the proof succeeds, and the algorithm returns I_1 as resource invariant.

Example 5. Here we discuss the execution of the synthesis algorithm on the program on the right of Fig. 2, which does not involve ownership transfer. As in Ex. 5 the algorithm initializes the resource invariant for *buf* to $I_0 = I_{put} \vee I_{get}$, where I_{put} an I_{get} are defined as in (3). Moreover, the initial specs for `put(x)` and `get(y)` are again as in (4). The forward analysis then easily proves the following triples (at each step Bi-Abductive Sequencing rule gets $\mathfrak{A} \equiv \mathsf{emp}$) :

$$\{\mathsf{emp}\}\, x = \mathtt{new()}; \mathtt{put(x)}; \mathtt{dispose(x)}\, \{\mathsf{emp}\} \qquad\qquad \{\mathsf{emp}\}\, \mathtt{get(y)}\, \{\mathsf{emp}\}$$

Hence the algorithm returns I_0 as a suitable resource invariant for this program.

Example 6. We now discuss a complex program which combines the one-place pointer transferring buffer and the memory manager [9]:

```
alloc(x);  ||  get(y);
put(x);    ||  dealloc(y);
```

Step 1 of Algorithm 1 initializes the resource invariants to

$$I^0_{buf} = (\neg \mathit{full} \wedge \mathsf{emp}) \vee (\mathit{full} \wedge \mathsf{emp}) \qquad\qquad I^0_{mm} = \mathsf{true} \wedge \mathsf{emp}$$

CompSpecs derives specifications for the CCRs, and, as seen in Ex. 3, it refines I^0_{mm} to obtain a resource invariant I^1_{mm} for the CCRs of resource mm. We have

$$\{\mathsf{emp}\}\, \mathtt{put(x)}\, \{\mathsf{emp}\} \qquad\qquad \{\mathsf{emp}\}\, \mathtt{get(y)}\, \{\mathsf{emp}\}$$
$$\{\mathsf{emp}\}\, \mathtt{alloc}(x)\, \{x \mapsto -\} \qquad\qquad \{y \mapsto -\}\, \mathtt{dealloc}(y)\, \{\mathsf{emp}\}$$

$$I^1_{mm} = (f{=}\mathsf{nil} \wedge \mathsf{emp}) \vee (f \mapsto \mathsf{nil}) \vee \mathsf{ls}(f, \mathsf{nil})$$

As in Ex. 1, using such specifications we can derive a proof for the LHS:

$$\frac{\{\texttt{emp}\}\,\texttt{alloc}(x)\,\{x\mapsto -\} \qquad \{\texttt{emp}\}\,\texttt{put(x)}\,\{\texttt{emp}\}}{\{\texttt{emp}\}\,\texttt{alloc}(x);\texttt{put(x)}\,\{x\mapsto -\}}$$

However, we cannot derive a proof for RHS since we get a non-empty anti-frame:

$$\frac{\{\texttt{emp}\}\,\texttt{get(y)}\,\{\texttt{emp}\} \qquad \{y\mapsto -\}\,\texttt{dealloc}(y)\,\{\texttt{emp}\}}{\{y\mapsto -\}\,\texttt{get(y)};\texttt{dealloc}(y)\,\{\texttt{emp}\}}$$

Therefore, refinement is required. This is done as in Ex. 4 where we get $I_{buf} \equiv (\neg \mathit{full} \wedge \texttt{emp}) \vee (\mathit{full} \wedge c \mapsto -)$ and specifications $\{x\mapsto -\}\,\texttt{put(x)}\,\{\texttt{emp}\}$ and $\{\texttt{emp}\}\,\texttt{get(y)}\,\{y\mapsto -\}$. Using them, both LHS and RHS are then proved.

6 Related Work

Our method for computing resource invariants uses bi-abduction, a technique which was recently introduced in [4] for discovering specifications of sequential programs. For simplicity, we have assumed that each resource declaration is annotated with the set of global variables it protects. Such annotations need not be given always by the user, as they can often be inferred by systems such as Locksmith [10].

The only shape analysis based on concurrent separation logic that attempts to calculate resource invariants is the thread-modular shape analysis by Gotsman et al. [7]. This uses a heuristic to decide how to partition the state into local and shared after each critical region. As a result, this method cannot use the same heuristic to verify both programs in Fig. 2.

Note that these small programs can be verified with analyses that are not thread-modular: e.g. by considering all thread interleavings as in Yahav [12], or by keeping track of the correlations between the local states of each pair of threads as in Segalov et al. [11]. The drawback of such analyses is that they do not scale very well to large programs. In contrast, as we use compositional techniques for discovering resource invariants, we are hopeful that our algorithm will scale to large programs.

7 Conclusion

In this paper, we have proposed a sound method for automating concurrent separation logic proofs by synthesizing suitable resource invariants. Our method is thread-modular in that it requires isolated inspection of sequential threads instead of the global parallel composition. The strength of our method relies on the ability to address one of the main open issues in the automation of proofs for concurrent separation logic. This is the ability to discern, in a thread local way, the cases where the resource invariant needs to describe the transfer of ownership (among threads), from those cases where no transfer should be involved.

This inherent complication has been described by O'Hearn by the expression "ownership is in the eye of the asserter". The technique proposed in this paper pushes the state of the art in automatic generation of proofs towards the more ideal situation where "ownership is in the eye of the mechanical method". We believe that this will open-up interesting possibilities for achieving more scalable automatic techniques for concurrent programs.

Acknowledgements. We thank Peter O'Hearn and Noam Rinetzky for invaluable comments.

References

1. J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang. Shape analysis of composite data structures. In *CAV'07*, 2007.
2. J. Berdine, C. Calcagno, and P. O'Hearn. Symbolic execution with separation logic. In *APLAS'05*, 2005.
3. J. Berdine, C. Calcagno, and P. O'Hearn. Smallfoot: Automatic modular assertion checking with separation logic. In *4th FMCO*, pp115-137, 2006.
4. C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, pages 289–300. ACM, 2009.
5. B. Chang, X. Rival, and G. Necula. Shape analysis with structural invariant checkers. In SAS'07, 2007.
6. D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS'06*, 2006.
7. A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Thread-modular shape analysis. In *PLDI*, 2007.
8. B. Gulavani, S. Chakraborty, G. Ramalingam and A. Nori. Bottom-up Shape Analysis. To appear in *SAS 2009*.
9. P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1-3):271–307, May 2007.
10. P. Pratikakis, J. S. Foster, and M. Hicks. Context-sensitive correlation analysis for detecting races. In *PLDI*, June 2006.
11. M. Segalov, T. Lev-Ami, R. Manevich, G. Ramalingam, and M. Sagiv. Efficiently inferring thread correlations. Unpublished, 2009.
12. E. Yahav. Verifying safety properties of concurrent java programs using 3-valued logic. In *POPL*, pages 27–40, 2001.

A Proofs

Lemma 1 *The* BA-with *rule is sound.*

Proof. BA-with is derivable from the Frame, Conseq, and WithRes rules of Concurrent Separation Logic, by observing that $Q * \mathfrak{A} \iff Q * [\mathfrak{A}]^{loc}_{Q}$ and that $\mathfrak{F} \implies \alpha(\exists \boldsymbol{c}.\mathfrak{F})$. □

Lemma 2 *If the number of program variables in* Prg *is finite then the transition system defined in Table 1 is finite.*

Proof. To see why the transition system has a finite number of states, it is enough to observe that $|L| \leq |\mathsf{Prg}|$. Moreover, we consider resource invariants and specifications which do not contain (redundant) equivalent disjuncts. Then, given that each disjunct is in canonical form (by the abstraction α), and that there are only a finite number of those (see Proposition 9 in [6]), we have that the number of possible resource invariants and specifications is finite. □

Theorem 1 *The* InvariantSynthesis *algorithm is sound.*

Proof. First notice that the set *Specs* built by the algorithm contains valid Hoare triples. Secondly, if the algorithm terminates without failure then the procedure *ProofSearch*(Prg, $\mathcal{I}$, *Specs*) builds a proof of Prg in Concurrent Separation Logic. Therefore, the soundness of the algorithm follows directly from the soundness of Concurrent Separation Logic[8]. □

Theorem 2 *Algorithm 1* InvariantSynthesis *terminates provided that the underlying forward analysis does.*

Proof. Algorithm 2 RefineOwnership terminates since it inspects a finite number of CCRs. Since the underlying forward analysis terminates by assumption, and since *CompSpecs* terminates by Corollary 1, then termination of InvariantSynthesis follows from the observation that the loop starting at line 3 is executed a finite number of times. To see why, observe that there is only a finite number of symbolic heaps in canonical form. Therefore there is only a finite number of pairs (*Specs*, $\mathcal{I}$). Moreover, *Failed* ensures that only invariants which did not fail before are inspected. Hence InvariantSynthesis terminates. □

[8] As in [7], we take advantage of the fact that the Conjunction Rule of Hoare Logic is not used in the proof to lift the restriction that resource invariants be precise [9].