GRASShopper

Complete Heap Verification with Mixed Specifications

Ruzica Piskac[1], Thomas Wies[2⋆], and Damien Zufferey[3]

[1]Yale University [2]New York University [3]MIT CSAIL

Abstract. We present GRASShopper, a tool for compositional verification of heap-manipulating programs against user-provided specifications. What makes our tool unique is its decidable specification language, which supports mixing of assertions expressed in separation logic and first-order logic. The user of the tool can thus take advantage of the succinctness of separation logic specifications and the discipline of local reasoning. Yet, at the same time, she can revert to classical logic in the cases where decidable separation logic fragments are less suited, such as reasoning about constraints on data and heap structures with complex sharing. We achieve this combination of specification languages through a translation to programs whose specifications are expressed in a decidable fragment of first-order logic called GRASS. This logic is well-suited for automation using satisfiability modulo theory solvers. Unlike other tools that provide similar features, our decidability guarantees enable GRASShopper to produce detailed counterexamples for incorrect or underspecified programs. We have found this feature to be invaluable when debugging specifications. We present the underlying philosophy of the tool, describe the major technical challenges, and discuss implementation details. We conclude with an evaluation that considers challenging benchmarks such as sorting algorithms and a union/find data structure.

1 Introduction

We present GRASShopper, a new tool for compositional verification of heap manipulating programs against user-provided specifications. GRASShopper takes programs in a C-like procedural language as input. The tool checks that procedures mutually satisfy their contracts, that all memory accesses are safe, and that there are no memory leaks. The unique feature of the input language is that it admits specifications that freely mix assertions expressed in separation logic and first-order logic.

Separation logic (SL) [18] is an extension of Hoare logic for proving the correctness of heap-manipulating programs. SL assertions specify regions in the heap rather than the global state of the heap. This distinction to classical logic gives rise to a discipline of local reasoning where the specification of a program fragment C only concerns C's *footprint*, i.e., the portion of memory on which C operates. This approach typically yields succinct and natural specifications that closely resemble a programmer's intuition about program correctness. Separation logic has therefore spawned extensive research into developing tool support for automated verification of programs against SL specifications [3,4,9,27]. The cores of such tools are specialized theorem provers for checking

⋆ Supported in part by NSF grant CCS-1320583.

entailments between SL assertions [2, 6, 7, 20]. Much of the work on such provers aims at decidable fragments of separation logic to guarantee a robust user experience.

Despite the elegance of separation logic, there are certain situations where it is more appropriate to express specifications in classical logic. This includes, for example, situations in which data structures exhibit complex sharing or involve constraints about data, e.g., arithmetic constraints. Reasoning about such constraints is not directly supported by SL theorem provers. The question is then how to extend these provers without giving up on decidability and completeness guarantees.

Typically, theory reasoning is realized by using a satisfiability modulo theories (SMT) solver that is integrated with the SL entailment procedure [5]. However, the interplay between SL reasoning and theory reasoning is intricate, e.g. equalities inferred by the theory solvers must be propagated back to the SL solver. Guaranteeing completeness of such a combined procedure is brittle and often involves the reimplementation of infrastructure that is already provided by the SMT solver.

In our previous work, we developed a new approach for checking SL entailments that reduces to checking satisfiability of formulas expressed in a decidable fragment of first-order logic [21]. We refer to this fragment as the logic of graph reachability and stratified sets (GRASS). Formulas in this logic express properties of the structure of graphs, such as whether nodes in the graph are inter-reachable, as well as properties of sets of nodes. The combination of these two features enables a natural encoding of the semantics of SL assertions. The advantage of this approach is that we can now delegate all reasoning to the SMT solver, exploiting existing infrastructure for combinations [17] and extensions [25] of first-order theories to handle reasoning about data robustly.

In this paper, we present GRASShopper, a tool which extends our previous work with support for local reasoning. Inspired by implicit dynamic frames [19, 24], we present a translation of programs with mixed separation logic and first-order logic specifications to programs with GRASS specifications. The translation and verification of the resulting program is fully automated. The key challenge in this approach is to ensure that the encoding of SL assertions and the support for local reasoning remains within a decidable logic. To this end, we present a decidable extension of the GRASS logic that suffices to express that reachability information concerning heap paths outside the footprint of a code fragment is preserved by the execution of that code fragment.

We implemented the decision procedure for our extension of GRASS on top of the SMT solver Z3 [8] and integrated this decision procedure into GRASShopper. We used the tool to automatically verify list-manipulating programs such as sorting algorithms whose specifications involve constraints on data. We further considered programs whose specifications are difficult to express in decidable SL fragments alone. One example is the find operation of a union/find data structure. The postcondition of this operation must describe a heap region that consists of an unbounded number of list segments. With our approach we can easily express this postcondition using a quantified constraint in classical logic, while using SL assertions to describe the precondition. The seamless yet robust combination of separation logic and classical logic in a specification language that supports local reasoning is the key contribution of this work.

```
1  struct Node { var data : int; var next: Node; }
2  predicate blseg(x: Node, y: Node, lb: int, ub: int) {
3    x = y ∨ x ≠ y ∗ acc(x) ∗ lb ⩽ x.data ⩽ ub ∗ blseg(x.next, y, lb, ub)
4  }
5  predicate bslseg(x: Node, y: Node, lb: int, ub: int) {
6    x = y ∨ x ≠ y ∗ acc(x) ∗ lb ⩽ x.data ⩽ ub ∗ bslseg(x.next, y, x.data, ub)
7  }
8  procedure quicksort(x: Node, y: Node, ghost lb: int, ghost ub: int) returns (rx: Node)
9    requires blseg(x, y, lb, ub);
10   ensures bslseg(rx, y, lb, ub);
11 { if (x≠y ∧ x.next≠y) {
12     var pivot: Node, z: Node;
13     rx, pivot := split(x, y, lb);
14     rx := quicksort(rx, pivot, lb, pivot.data);
15     z := quicksort(pivot.next, y, pivot.data, ub);
16     pivot.next := z;
17   } else { rx := x; }
18 }
```

Fig. 1. A partial implementation of a quicksort algorithm on singly-linked lists

2 Overview and Running Example

We illustrate our approach through an example that implements a quicksort algorithm for linked lists storing integer values. The implementation and specification is shown in Figure 1. We use the syntax of GRASShopper's input language (modulo mark-up).

The procedure quicksort takes two pointers x and y as input, marking the start and end points of the list segment that is to be sorted. This property is expressed by the SL assertion in the precondition of quicksort: the inductive predicate blseg(x, y, lb, ub). The predicate states that x and y are indeed the start and end points of an acyclic list segment. Furthermore, it states that the data values of this list segment are bounded from below and above by the values lb and ub, respectively. These values are passed to quicksort as additional ghost parameters. The atomic predicate acc(x) in the definition of blseg represents a heap region that consists of the single heap cell x. That is, acc(x) means that x is in the footprint of the predicate. Such SL assertions are combined to assertions describing larger heap regions using *spatial conjunction*, denoted by '*'. Spatial conjunction asserts that the composed heap regions are disjoint in memory. Hence, blseg describes an *acyclic* list segment. Note that atomic assertions such as $x = y$ only express constraints on values but describe empty heap regions. In particular, $x = y \vee x \neq y$ is not a tautology. Such constraints are called *pure* in SL jargon. Further note that spatial conjunction binds stronger than classical conjunction and disjunction.

The footprint of blseg(x, y, lb, ub) is also the initial footprint of procedure quicksort which, by induction, consists of all heap cells between x and y, excluding y. The quicksort procedure returns a pointer rx to the head of the sorted list segment, which we specify in the postcondition using the predicate bslseg(rx, y, lb, ub). For exposition purposes, we do not specify that the output list is a permutation of the input list.

In the recursive case, quicksort picks a pivot and splits the list into two segments, one containing all values smaller than pivot.data, and one containing all other values. To simplify the presentation, we have factored out the code for the actual splitting in

```
1  procedure split(x: Node, y: Node, ghost lb: int, ghost ub: int) returns (rx: Node, pivot: Node)
2    requires blseg(x, y, lb, ub) ∗ x ≠ y;
3    ensures blseg(rx, pivot, lb, pivot.data) ∗ blseg(pivot, y, pivot.data, ub);
4    ensures Btwn(next, rx, pivot, y) ∗ pivot ≠ y ∗ lb ⩽ pivot.data ⩽ ub;
```

Fig. 2. Specification of the procedure split used by quicksort

a separate procedure split. After splitting, quicksort recursively calls itself on the two sublists and concatenates the two sorted list segments.

We provide the specification of split but not its implementation. It is shown in Fig. 2. The specification is agnostic to implementation details such as whether only the data values are reordered in the list or the entire nodes. Multiple ensures, respectively, requires clauses in a procedure contract are implicitly connected by spatial conjunction.

The procedure split also demonstrates the convenience of a specification language that allows mixing of separation logic and reachability logic. The conjunct Btwn(next, rx, pivot, y) in the second ensures clause is a predicate in our logic GRASS. The predicate states that the node pivot lies between rx and y on the direct next path connecting the two nodes. That is, the two list segments described by the first ensures clause do not form a panhandle list. A panhandle list can occur if y is a dangling pointer to an unallocated node and split allocates that node and inserts it into the list segment from rx to pivot, thereby creating a cycle. Without the additional reachability constraint, the specification of split would be too weak to prove the correctness of quicksort because the final sorted list segment returned by quicksort must be acyclic. If we used either only separation logic or only reachability logic, the specification of procedure split would be considerably more complicated (assuming we stayed inside decidable fragments).

3 Verifying Programs with GRASShopper

The verification of the input program provided to GRASShopper proceeds in three steps: first we translate the program to an equivalent program whose specification is expressed solely in our first-order logic fragment GRASS; in the second step we encode the translated program into verification conditions (also expressed in GRASS) using standard verification condition generation; finally we decide the generated verification conditions using our GRASS solver. All three steps are fully automated in GRASShopper. We now explain these steps using the quicksort procedure as a running example.

3.1 Translation to GRASS Programs

We first describe the translation of the input program to a GRASS program. The translation must capture the semantics of Hoare triples in separation logic and preserve the ability to reason about correctness locally. For a Hoare triple $\{P\}C\{Q\}$ to be valid in separation logic, the precondition P must subsume the footprint of the program fragment C. That is, P specifies the portion of memory that C is allowed to access. This semantics enables local reasoning, which is distilled into the so-called *frame rule*. The frame rule states that if $\{P\}C\{Q\}$ is valid, then so is $\{P * F\}C\{Q * F\}$ for any SL

assertion F. That is, C does not affect the state of memory regions disjoint from its footprint. The assertion F is referred to as the *frame* of the rule application.

The frame rule enables compositional symbolic execution of program fragments. For example in quicksort, the symbolic state after the call to split in line 13 is described by the postcondition of split. The first subsequent recursive call to quicksort then only operates on the first sublist blseg(rx,pivot,lb,ub) of that symbolic state, leaving blseg(pivot,y,lb,ub) in the frame. The frame rule then implies that this second sublist is not modified by the first recursive call. All such applications of the frame rule for procedure calls are made explicit in the GRASS program.

The translation to a GRASS program proceeds one procedure at a time. Each resulting procedure is equivalent to its counterpart in the input program, modulo auxiliary ghost state. This auxiliary ghost state makes the semantics of separation logic specifications explicit and encodes the applications of the frame rule. Figure 3 shows the result of the translation for the quicksort procedure. The translation works as follows.

Alloc. First, we introduce a global ghost variable Alloc (line 2), which is used to model allocation and deallocation instructions. That is, at any point of execution, Alloc denotes the set of all Node objects that are currently allocated on the heap.

Footprints and Implicit Frame Inference. Each procedure maintains its own footprint throughout its execution using the dedicated local ghost variable FP. That is, at any point of a procedure's execution, FP contains the set of all heap nodes that the procedure has permission to access or modify at that point. Each heap access or modification is therefore guarded by an assert statement that checks whether the modification is permitted by the current footprint (see, e.g., lines 25 and 29). The translation maintains the invariant that footprints contain only allocated nodes. That is, both allocation and deallocation instructions affect FP.

For each procedure call, the footprint of the caller is passed to the callee and the callee returns the new footprint of the caller. That is, it is the callee's responsibility to inform the caller about allocation and deallocation operations that affect the caller's footprint. For this purpose, each procedure is instrumented with an additional ghost input parameter FP_Caller and an additional ghost return parameter FP_Caller'.

The contract of the translated procedure governs the transfer of permissions between caller and callee via the exchanged footprints and ties the footprints to the translations of the separation logic specifications in the original procedure contract. The initial value of FP in the translated procedure is determined by the footprint of the separation logic assertions in the precondition of the input procedure, which itself must be a subset of the callers footprint (line 16).

Note that the ghost variable FP is declared as an implicit ghost input parameter of the procedure (line 13). The semantics of an implicit ghost parameter is that it is existentially quantified across the entire procedure contract[1]. That is, during verification condition generation, the precondition of the contract is asserted at the call site with all implicit ghost parameters existentially quantified. When the solver checks the generated verification condition for this assertion, it needs to find a witness for FP, thereby implicitly inferring the frame of the procedure call that is used in the application of

[1] We adhere to the usual semantics of procedure contracts where input parameters occuring in ensures clauses refer to the initial values of these parameters.

```
1  struct Node { var data : Int; var next: Node; }
2  ghost var Alloc: set<Node>;
3  function blseg_fp(x: Node, y: Node) returns (Footprint: set<Node>) {
4    Footprint = {z: Node :: Btwn(next, x, z, y) ∧ z ≠ y}
5  }
6  predicate blseg_struct(x: Node, y: Node, lb: int, ub: int) {
7    Btwn(next, x, y, y) ∧∀z ∈ blseg_fp(x, y) :: lb ⩽ z.data ⩽ ub
8  }
9  predicate bslseg_struct(x: Node, y: Node, lb: int, ub: int) {
10   blseg_struct(x,y,lb,ub) ∧ ∀z,w ∈ blseg_fp(x, y) :: Btwn(next,z,w,y) ⇒ z.data ⩽ w.data
11 }
12 procedure quicksort(x: Node, y: Node, ghost lb: int, ghost ub: int,
13     ghost FP_Caller: set<Node>, implicit ghost FP: set<Node>)
14     returns (rx: Node, ghost FP_Caller': set<Node>)
15   requires blseg_struct(x, y, lb, ub);
16   requires FP = blseg_fp(x, y) ∧ FP ⊆ FP_Caller;
17   free requires FP_Caller ⊆ Alloc ∧ null ∉ Alloc;
18   modifies next, data, Alloc;
19   ensures bslseg_struct(rx, y, lb, ub);
20   ensures blseg_fp(rx, y) = (Alloc ∩ FP) ∪ (Alloc \ old(Alloc));
21   free ensures FP_Caller' = (FP_Caller \ FP) ∪ (Alloc ∩ FP) ∪ (Alloc \ old(Alloc));
22   free ensures FP_Caller' ⊆ Alloc ∧ null ∉ Alloc;
23   free ensures Frame(old(Alloc), FP, old(next), next) ∧ Frame(old(Alloc), FP, old(data), data);
24 { FP_Caller := FP_Caller \ FP;
25   assert x = y ∨ x ∈ FP;
26   if (x≠y ∧ x.next≠y) {
27     var pivot: Node, z: Node;
28     rx, pivot, FP := split(x, y, lb, FP);
29     assert pivot ∈ FP;
30     rx, FP := quicksort(rx, pivot, lb, pivot.data, FP);
31     z, FP := quicksort(pivot.next, y, pivot.data, ub, FP);
32     pivot.next := z;
33   } else { rx := x; }
34   FP_Caller' := FP_Caller ∪ FP; }
```

Fig. 3. Translation of quicksort program from Figure 1 to an equivalent GRASS program.

the frame rule. After the precondition has been asserted, it is assumed with the implicit ghost parameters replaced by fresh Skolem constants. These Skolem constants then also occur in the assumed postcondition at the call site.

Encoding the Frame Rule. The free requires and ensures clauses in the contract constitute the actual encoding of the frame rule. The free annotation means that the corresponding clause does not need to be checked but can be freely assumed by the callee, respectively, caller. These clauses follow from the soundness of the frame rule and the invariants concerning Alloc and the footprints that are guaranteed by the translation. We discuss the most important parts of the encoding in more detail:

– First, consider the ensures clause in line 20: $\mathsf{blseg\_fp}(\mathsf{rx}, \mathsf{y}) = (\mathsf{Alloc} \cap \mathsf{FP}) \cup (\mathsf{Alloc} \setminus \mathbf{old}(\mathsf{Alloc}))$. This clause states that the footprint of the postcondition, denoted by $\mathsf{blseg\_fp}(\mathsf{rx}, \mathsf{y})$, accounts for all memory in the initial footprint that has

not been deallocated, and all memory that has been freshly allocated (but not deallocated again) during execution of quicksort. This clause thus implies that the procedure does not leak memory.
- Next, consider the ensures clause in line 21: $\mathsf{FP\_Caller}' = (\mathsf{FP\_Caller} \setminus \mathsf{FP}) \cup (\mathsf{Alloc} \cap \mathsf{FP}) \cup (\mathsf{Alloc} \setminus \mathbf{old}(\mathsf{Alloc}))$. This clause states that the new footprint of the caller, FP_Caller', is the caller's old footprint with the initial footprint of quicksort replaced by quicksort's final footprint (as defined in line 20).
- Finally, the clause in line 23 states that the fields next and data are not modified in the frame of the call. We express this using the predicate Frame. The frame of the call is given by the set $\mathsf{old(Alloc)} \setminus \mathsf{FP}$. We discuss the predicate Frame in more detail in the next section, as the choice of its encoding is crucial for the completeness of our translation.

Translation of SL Assertions. Finally, we describe the translation of the SL assertions in the contract of the input procedure. This translation generalizes our previous work on deciding entailment in separation logic of linked lists via reduction to GRASS [21].

First, each inductive SL predicate $\mathsf{p}(\boldsymbol{x})$ in the input program is translated to a GRASS predicate $\mathsf{p\_struct}(\boldsymbol{x})$ and a function $\mathsf{p\_fp}(\boldsymbol{x})$. The predicate $\mathsf{p\_struct}(\boldsymbol{x})$ collects all constraints concerning the structure of the heap region that is described by the SL predicate $\mathsf{p}(\boldsymbol{x})$, while the function $\mathsf{p\_fp}(\boldsymbol{x})$ denotes the footprint of $\mathsf{p}(\boldsymbol{x})$. For example, consider the predicate blseg(x,y,lb,ub) in the input program. As expected, its footprint function blseg_fp(x,y) denotes the set of all nodes z on the next path between x and y, excluding y. This is expressed in terms of a set comprehension. Such set comprehensions are expanded to universally quantified constraints in the back-end solver. Note that if y is not reachable from x in the heap, then blseg_fp(x,y) denotes the empty set. For convenience, we reuse the same footprint function for the translation of the predicate bslseg. The predicate blseg_struct(x,y,lb,ub) states that x is indeed reachable from y (which is expressed by the predicate Btwn(x,y,y)) and that the nodes in the footprint store data values in the interval [lb,ub]. Our tool uses a sound heuristic to generate the translations of the user-defined inductive predicates. The heuristic cannot be complete for arbitrary inductively defined predicates, as the problem of checking entailment for such predicates becomes undecidable. However, our back-end solver is complete for the translations of a large class of predicates describing linked list structures, including the ones in the quicksort example.

With the translation of inductive predicates in place, the translation of an SL assertion H to a GRASS formula is then given by a function $tr(H, X)$, where X is a set variable that denotes the footprint of the assertion. The definition of $tr(H, X)$ is defined recursively on the structure of H as follows:

- if $H = \mathsf{p}(\boldsymbol{x})$, then $tr(H, X) \equiv \mathsf{p\_struct}(\boldsymbol{x}) \wedge X = \mathsf{p\_fp}(\boldsymbol{x})$;
- if $H = \mathsf{acc}(x)$ where x is a node variable, then $tr(H, X) \equiv X = \{x\}$;
- if $H = \mathsf{acc}(Y)$ where Y is a node set variable, then $tr(H, X) \equiv X = Y$;
- if $H = F$ where F is a pure constraint, then $tr(H, X) \equiv F \wedge X = \varnothing$;
- if $H = H_1 * H_2$, then $tr(H, X) \equiv \exists X_1, X_2 :: tr(H_1, X_1) \wedge tr(H_2, X_2) \wedge X = X_1 \uplus X_2$, where X_1, X_2 are fresh node set variables;
- if $H = H_1 + H_2$, then $tr(H, X) \equiv \exists X_1, X_2 :: tr(H_1, X_1) \wedge tr(H_2, X_2) \wedge X = X_1 \cup X_2$, where X_1, X_2 are fresh node set variables.

For convenience, we also include nondisjoint spatial composition in our SL assertion language, which we denote by $H_1 + H_2$. This operator is useful to specify overlayed data structures concisely, respectively, specify alternative views of the same data structure. Note that the *points-to* predicate $\mathsf{x.next} \mapsto \mathsf{y}$ that is commonly used in separation logic fragments is simply a short-hand for the assertion $\mathsf{acc(x)} * \mathsf{x.next} = \mathsf{y}$.

Example 1. In Figure 3, the translation $tr(\mathsf{blseg(x,y,lb,ub)}, \mathsf{FP})$ of the original precondition of the quicksort procedure is the conjunction of the clause in line 15 and the first set equality in the clause in line 16.

Apart from the treatment of inductive predicates, the translation of SL assertions is surprisingly close to the way in which their semantics is traditionally defined. To the expert reader, this might seem problematic, at first. Namely, when checking the generated verification conditions, the back-end solver for GRASS negates some of the resulting constraints to reduce the problem to satisfiability queries. Thus, some of the auxiliary existentially quantified set variables that are introduced in the translation of spatial operators[2] become universally quantified. This might raise concerns about decidability. However, the translation function is defined in such a way that all existentially quantified set variables are uniquely defined by set equalities. That is, the negated constraints of the form $\forall X :: X = T \Rightarrow F$ can be transformed back into equivalent constraints of the form $\exists X :: X = T \wedge F$.

3.2 Frame Axioms and Completeness

We next discuss how we ensure both completeness of the translation to GRASS programs and decidability of checking the generated verification conditions (relative to certain assumptions about the specifications in the input program).

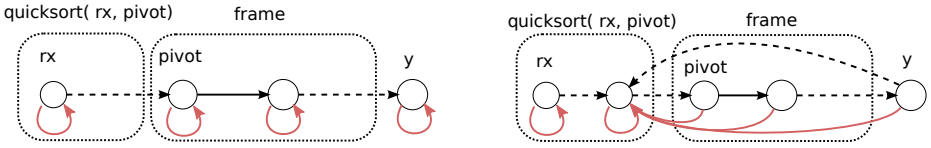
To enable efficient verification condition generation where all case splitting is differed to the back-end SMT solver, we model fields such as next and data as arrays. This allows us to encode field updates conveniently as store operations, which are supported by the array theory in the SMT solver. However, we also need to model the effect of procedure calls on fields, and how modifications of fields affect reachability information captured by the Btwn predicate.

Ultimately, both completeness and decidability hinge on the interpretation of the frame axioms $\mathsf{Frame}(A, FP, f, f')$, which we use to encode the application of the frame rule. Here, A and FP are the values of Alloc and FP before a procedure call, and f and f' are arrays that encode the state of a field such as next before and after the call. In principle, it is sufficient to consider the following interpretation of Frame, which states that the field f is not modified in the frame of the call:

$$\mathsf{Frame}(\mathsf{Alloc}, \mathsf{FP}, f, f') \equiv \forall x \in (\mathsf{Alloc} \backslash \mathsf{FP}) :: x.f = x.f' \tag{1}$$

The translation to GRASS programs that we outlined in the previous section would then be complete if we considered an axiomatic semantics where GRASS formulas are interpreted in a first-order logic with transitive closure. Transitive closure enables

[2] as well as the quantified implicit ghost parameter FP in call-site checks of preconditions



(a) y does not reach nodes in the footprint (b) y reaches the footprint (panhandle list).

Fig. 4. Two of the possible heaps at the call site on line 14. The footprint of the recursive call to quicksort and the portion of the frame that belongs to the caller's footprint are enclosed in dotted boxes. Solid black edges denote next pointers, dashed black edges indicate next paths, and solid red edges represent the ep function.

us to tie the interpretation of a predicate Btwn(next,x,y,z) on a semantic level to the interpretation of next in a given program state. However, the problem of checking the generated verification conditions would be undecidable [11].

An alternative approach is to tie the interpretation of Btwn(next,x,y,z) to the interpretation of next on an axiomatic level. In general, transitive closure cannot be axiomatized in first-order logic. However, we are considering the special case of finite structures, for which first-order axiomatizations of transitive closure exist. In fact, several *reachability logics* for reasoning about heap structures have been proposed that can be decided efficiently (see, e.g., [15, 26]). The problem now is to preserve precise reachability information in the presence of field modifications, i.e., how do Btwn(next,x,y,z) and Btwn(next',x,y,z) relate if next' is obtained from next by some (possibly unbounded) sequence of updates. For single heap updates p.next := q, the effect on the reachability predicate can be encoded using appropriate axioms [15]. However, to preserve reachability information for heap paths in the frame of a procedure call (which may execute an unbounded number of heap updates) we need a more general mechanism.

To preserve reachability information in the frame, we need an interface between the frame and the footprint of the callee that distinguishes the portions of a path belonging to the frame from those portions belonging to the footprint. We define this interface using the *entry point function*. The entry point for a heap node x with respect to a set X and field f, denoted $ep(X, f, x)$, is defined as the first node in X that is reachable from x via f. If such a node does not exist, then $ep(X, f, x) = x$.

Example 2. Figure 4 illustrates two different heap states that may occur at the call site of the recursive call to quicksort on line 14 in Figure 1. The evaluation of the entry point function is depicted by red arrows.

We axiomatize ep in terms of the predicate Btwn as follows:

$$\forall x :: \mathsf{Btwn}(f, x, ep(X, f, x), ep(X, f, x))$$
$$\forall x :: ep(X, f, x) \in X \vee ep(X, f, x) = x$$
$$\forall x, y :: \mathsf{Btwn}(f, x, y, y) \wedge y \in X \Rightarrow ep(X, f, x) \in X \wedge \mathsf{Btwn}(f, x, ep(X, f, x), y)$$

Using the entry point function we can now correctly update the reachability information for paths that cross the boundary into the footprint of the callee. The corresponding

frame axiom for pointer fields such as next is then as follows:

$$\begin{aligned}\mathsf{Frame}(A, FP, f, f') \equiv\ & \forall x \in (A \backslash FP) :: x.f = x.f' \wedge \\ & \forall x, y, z \in (A \backslash FP) :: \mathsf{ReachWO}(f, x, y, ep(FP, f, x)) \Rightarrow \\ & \quad (\mathsf{Btwn}(f, x, z, y) \Leftrightarrow \mathsf{Btwn}(f', x, z, y)) \wedge \\ & \forall x, y, z \in A :: x \notin FP \wedge x = ep(FP, f, x) \Rightarrow \\ & \quad (\mathsf{Btwn}(f, x, y, z) \Leftrightarrow \mathsf{Btwn}(f', x, y, z))\end{aligned}$$

The two additional axioms specify that the order of nodes is preserved for the path segments between any node x and its entry point into FP, respectively, the full path starting in x if no node in FP is reachable from x. The predicate $\mathsf{ReachWO}(f, x, y, z)$ means that x can reach y via f without going through z. We express this as follows:

$$\mathsf{ReachWO}(f, x, y, z) \equiv \mathsf{Btwn}(f, x, y, z) \vee \mathsf{Btwn}(f, x, y, y) \wedge \neg \mathsf{Btwn}(f, x, z, z)$$

For nonpointer fields such as data, equation 1 is already sufficient.

3.3 Deciding the Verification Conditions

The verification conditions that are generated from the GRASS programs are augmented with theory axioms to encode the semantics of predicates such as Btwn as well as operations on sets. The resulting formulas are in first-order logic, checked for (un)satisfiability modulo first-order theories that are natively supported by SMT solvers, e.g., linear arithmetic and free function symbols. The generated formulas contain both existential and universal quantifiers, however, no $\forall\exists$ quantifier alternations. To ensure that we can use the SMT solver as an actual decision procedure for checking satisfiability of the generated formulas, we preprocess these quantifiers before we pass the formula to the SMT solver. Preprocessing depends on the kind of the quantifier:

- Existentially quantified subformulas are simply skolemized. We implemented optimization such as maximizing the scope of existential quantifiers and reusing existentially quantified variables as much as possible to minimize the number of generated Skolem constants.
- Universally quantified subformulas are first hoisted to the top level of the formula (by introducing propositional variables as place holders) and then further processed depending on their type. We distinguish three types that we further describe below.

Effective Propositional Fragment (EPR). The EPR fragment (aka the Bernays-Schönfinkel-Ramsey class) consists of formulas in which universally quantified variables do not occur below function symbols. This fragment can be decided quite efficiently using Z3's model-based quantifier instantiation mechanism. Hence, all EPR formulas are passed directly to Z3. For formulas that are not in EPR, we make a finer distinction.

Stratified Sort Fragment. If universal quantified variables appear below function symbols, then instantiating these variables may create new ground terms, which in turn can be used for instantiation, causing the SMT solver to diverge. One special case, though, are axioms satisfying stratified sort restrictions [1]. Examples of such formulas are the quantified constraints in the predicates blseg_struct and bslseg_struct of Figure 3. The

sort of the quantified variables z and w is Node, while the sort of the instantiated terms z.data and w.data is int. Since we do not quantify over int variables, the generated ground terms do not enable new quantifier instantiations. Formulas in the stratified sort fragment are directly passed to Z3.

Local Theory Extensions. The remaining quantified constraints are more difficult. In general, we provide no completeness guarantee for our handling of quantifiers because we allow users to specify unrestricted quantified pure constraints in their specifications. However, we can guarantee completeness for specifications written in separation logic for linked lists mixed with quantifier-free pure GRASS constraints (as well as some types of user-specified quantified constraints). We designed our translation carefully so that the remaining quantified formulas are in decidable fragments (in particular, the frame and theory axioms). To decide these fragments, we build on local theory extensions [25]. Local theory extensions are described by axioms for which instantiation can be restricted to ground terms appearing in the verification condition (or some finite set of ground terms that can be computed from this formula). We preprocess such axioms by partially instantiating all variables below function symbols with the relevant sets of ground terms. The partially instantiated axioms are then in the EPR fragment and passed to Z3. We discuss one example of a local theory extension in more detail below. To reduce the number of generated partial instances, we compute the congruence closure for the ground part of the verification condition to group ground terms into equivalence classes. We then only need to consider one representative term per equivalence class during instantiation.

Example 3. One example of a local theory extension is the theory extension defining the entry point functions in Section 3.2 together with the generated frame axioms concerning ep. Note that in all models of this extension, the entry point function is idempotent for fixed X and f. Hence, we only need to instantiate these axioms once for each Node ground term x. One potential problem may arise from the interactions between the ep functions for different footprint sets and fields. That is, instantiating one ep term for one X, f and ground term t may expose a new entry point $e = ep(X', f', ep(X, f, t))$ for another pair X', f' such that, in some model, e is different from all previously generated ground terms. However, such a situation cannot occur if all footprints are defined by a union of a bounded number of list segments. This holds true for separation logic of linked lists. Even in the general case, the counterexamples that witness incompleteness are rather degenerate and we doubt they can occur in actual program executions.

4 Mixing Separation Logic and First-Order Logic Specifications

The key advantage of our approach is that it allows the user to seamlessly mix SL and GRASS specifications. Some data structures are difficult to specify in separation logic because they involve complex sharing, or their footprints are not easily definable using simple inductive predicates. In Figure 5, we show the specifications of the find and union procedures of a union-find data structure implemented as a forest of inverted trees. This data structure exhibits both of the above problems.

Complex Sharing. A path that goes from a node to its representative in a union-find structure can be expressed as a list segment. However, describing the entire structure is

```
1  predicate lseg_set(x: Node, y: Node, X: set<Node>) {
2    (x = y ∗ X = ∅) ∨ (x ≠ y ∗ acc(x) ∗ x ∈ X ∗ lseg_set(x.next, y, X \ {x}))
3  }
4  procedure find(x: Node, ghost root_x: Node, implicit ghost X: set<Node>)
5    returns (res: Node)
6    requires lseg_set(x, root_x, X) ∗ root_x.next ↦ null;
7    ensures res = root_x ∗ acc(X) ∗ (∀ z ∈ X :: z.next = res) ∗ res.next ↦ null;
8
9  procedure union(x: Node, y: Node, ghost root_x: Node, ghost root_y: Node,
10                 implicit ghost X: set<Node>, implicit ghost Y: set<Node>)
11   requires lseg_set(x, root_x, X) + lseg_set(y, root_y, Y);
12   requires root_x.next ↦ null + root_y.next ↦ null;
13   ensures (acc(X) + acc(Y)) ∗ (root_y.next ↦ null + acc(root_x));
14   ensures (∀ z ∈ X :: z.next = root_x) ∗ (∀ z ∈ Y :: z.next = root_y);
15   ensures root_x = root_y ∨ root_x.next = root_y;
```

Fig. 5. Operations on a union-find data structure with mixed specifications

more difficult. For instance, in the union procedure if x and y are in different equivalence classes, then the two paths in the data structure are disjoint. However, if they are in the same class, then their paths may be partially shared. It is difficult to express this in traditional SL fragments without explicitly distinguishing the two cases. We can cover both cases conveniently using the spatial connective $+$ for nondisjoint union.

Structural constraints expressed in first-order logic. When path compaction is used in the find procedure, then the postcondition of find is not expressible in terms of a bounded number of inductive predicates. The reason is that path compaction turns a list segment of unbounded length into an unbounded number of points-to predicates. Therefore, expressing the postcondition requires some form of universal quantification. We can express this quite easily using the constraint $F \equiv \forall z \in \mathsf{X} :: z.\mathsf{next} = \mathsf{root\_x}$, where X is the initial footprint of the procedure described by an SL assertion. Note that the additional predicate acc(X) in the postcondition specifies that X is also the final footprint of the procedure. Hence, F only constrains the structure of the heap region that is captured by the footprint. Note that this example also uses implicit ghost parameters of procedures to existentially quantify over the explicit footprint X.

When mixing separation logic and classical logic, then additional well-formedness checks are needed to guarantee that reachability predicates and other heap-dependent pure formulas do not constrain heap regions outside of the footprint that is specified by the nonpure SL assertions. Otherwise, the application of the frame rule would become unsound. However, these additional checks can be automated in the same manner as the checks of the actual verification conditions.

5 Implementation and Evaluation

We have implemented all the features described in this paper in GRASShopper. The tool is implemented in OCaml and available under a BSD license. The source code distribution including all benchmarks can be downloaded from the project web page [10].

GRASShopper takes as input an annotated C-like program and generates verification conditions, which are checked using a back-end SMT solver. The solver is integrated

Benchmarks	# LOC	# VCs	time in s
SLL (loop)	156	56	1.9
SLL (rec.)	142	70	3.1
sorted SLL	171	55	6.6
DLL	195	59	11
sorting algorithms	230	98	15
union-find	35	8	4.8
SLL.filter (deref. null pointer)		7	0.4
DLL.insert (missing update)		8	3.1
quicksort (underspec. split)		12	0.9
union-find (bug in postcond.)		4	12.8

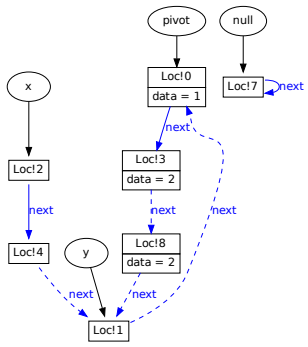


Fig. 6. The left-hand side shows the summary of the experiments for the collections of correct benchmarks as well as some benchmarks that contain bugs in the code or specification. The right-hand side shows the generated counterexample for the underspecified quicksort program

via the standard interface defined by SMT-LIB 2. Currently, we use Z3 [8] as back-end solver but we are working on incorporating CVC4 as well as other solvers.

Evaluation. We have collected 37 examples of correct heap-manipulating programs working over singly and doubly-linked lists. This includes basic manipulations of the data structures (traverse, dispose, copy, reverse, concat, filter, remove, insert) for the singly-linked lists (SLL) and doubly-linked lists (DLL). The singly-linked list examples come in three flavors: an imperative style loop-based implementation, a recursive implementation, and one based on sorted lists. Beyond these benchmarks, we implement four different sorting algorithms (insertion sort, merge sort, quicksort, strand sort) and a union-find data structure. In addition, we applied the tool to programs that contain bugs or have incorrect specifications. The table in Fig. 6 summarizes our results. The table shows the number of lines of code for each set of examples, the total number of verification conditions, and the total running time of GRASShopper on those examples. All examples have been correctly verified, respectively, falsified. The number of lines of code includes the specifications but excludes the definitions of the data structures.

Counterexample Generation. When a verification condition cannot be proved, i.e., the formula sent to the SMT solver is satisfiable, GRASShopper uses the model returned by the solver to construct a counterexample. Due to the preprocessing of quantifiers, the model returned by the SMT solver is actually a partial model of the GRASS formula. This means that instead of having all pointer fields defined, some of them are summarized by reachability constraints. These reachability constraints encode paths of unbounded length in the heap. From this information we construct a graph in Graphviz format that represents an entire family of counterexamples.

For example, when we were writing the quicksort example in Fig. 1, we had to iterate a few times before we obtained a correct version. At some point, we had a postcondition for split that was missing the Btwn predicate, as described in Section 2. The corresponding counterexample produced by GRASShopper is shown in Fig. 6. The graph clearly shows the panhandle list. The full counterexample also includes valuations for the footprint sets of the caller and callee. The final footprint FP_Caller' returned by split

is $\{\mathsf{Loc!0}, \mathsf{Loc!1}, \mathsf{Loc!2}, \mathsf{Loc!3}, \mathsf{Loc!4}, \mathsf{Loc!8}\}$ and the footprint that was expected by the postcondition of quicksort is $\{\mathsf{Loc!2}, \mathsf{Loc!4}\}$. The two sets should be equal.

6 Related Work and Conclusion

Since the pioneering work on the Smallfoot tool [2, 3], several efficient decision procedures for entailment checking in separation logic of linked lists have been developed [7, 20]. Other procedures target more expressive fragments, e.g., nested lists [6] or structures with tree backbones [12]. Currently, GRASShopper only supports structures with a flat list backbone but we are working on extending the tool to handle more complex data structures.

In our previous work [21], we proposed an approach to deciding entailment in separation logic via a reduction to first-order logic and presented a technique for frame inference. However, this technique relied on model enumeration, which is very expensive. We now propose an alternative where the frame rule is encoded in the SMT query.

Qiu et al. [22] introduced DRYAD a logic to specify heap shapes. To reason about DRYAD formulas, they use natural proofs, a heuristic to bound the proof search space. For instance, the unfolding of recursive definitions is limited to the ground terms in the formulas. This is similar to our approach of quantifier instantiation based on local theory extensions, but without completeness guarantees.

Closely related to our approach is the work on using effectively propositional logic (EPR) for reasoning about programs that manipulate linked lists [13, 14]. As in this paper, the authors of [14] use idempotent entry point functions to express that heap paths in the frame of a procedure call do not change. Their approach yields a sound and complete procedure for modular checking of EPR specifications. We have developed the same idea independently, motivated by the goal of verifying programs with specifications that mix separation logic with first-order theories. The union/find data structure has also been considered in [14]. Beside the different motivation, the main technical difference between our work and [14] is that we are not restricted to programs with acyclic lists. Incidentally, the more general reachability predicate that we use for reasoning about cycles yields a simpler encoding of the frame rule.

Our SL translation and the handling of the frame rule is in part inspired by work on implicit dynamic frames [19, 24]. Per se, the implicit dynamic frames approach provides no decidability guarantees for the first-order logic fragment used by the SL encoding. In particular, tools such as VeriCool [23] and Chalice [16], which are based on this approach, use pattern-based quantifier instantiation heuristics to check the resulting verification conditions. These heuristics are in general incomplete and often fail to produce models for satisfiable formulas. Instead, we designed the target fragment of our SL encoding carefully so that decidability is preserved by the translation while still admitting efficient implementations on top of SMT solvers. We find the ability of our implementation to produce counterexamples invaluable when debugging specifications.

References

1. A. Abadi, A. Rabinovich, and M. Sagiv. Decidable fragments of many-sorted logic. In *LPAR*, pages 17–31, Berlin, Heidelberg, 2007. Springer-Verlag.

2. J. Berdine, C. Calcagno, and P. O'Hearn. A decidable fragment of separation logic. In *FSTTCS*, 2004.
3. J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, 2005.
4. J. Berdine, B. Cook, and S. Ishtiaq. SLAyer: Memory Safety for Systems-Level Code. In *CAV*, 2011.
5. M. Botincan, M. J. Parkinson, and W. Schulte. Separation logic verification of C programs with an SMT solver. *Electr. Notes Theor. Comput. Sci.*, 254:5–23, 2009.
6. A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. Accurate invariant checking for programs manipulating lists and arrays with infinite data. In *ATVA*. Springer, 2012.
7. B. Cook, C. Haase, J. Ouaknine, M. Parkinson, and J. Worrell. Tractable reasoning in a fragment of separation logic. In *CONCUR*. Springer, 2011.
8. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.
9. K. Dudka, P. Peringer, and T. Vojnar. Predator: A practical tool for checking manipulation of dynamic data structures using separation logic. In *CAV*, 2011.
10. GRASShopper tool wep page. `http://cs.nyu.edu/wies/software/grasshopper`. Last accessed: October 2013.
11. N. Immerman, A. M. Rabinovich, T. W. Reps, S. Sagiv, and G. Yorsh. The boundary between decidability and undecidability for transitive-closure logics. In *CSL*. Springer, 2004.
12. R. Iosif, A. Rogalewicz, and J. Simácek. The tree width of separation logic with recursive definitions. In *CADE*, volume 7898 of *LNCS*, pages 21–38. Springer, 2013.
13. S. Itzhaky, A. Banerjee, N. Immerman, A. Nanevski, and M. Sagiv. Effectively-propositional reasoning about reachability in linked data structures. In *CAV*. Springer, 2013.
14. S. Itzhaky, O. Lahav, A. Banerjee, N. Immerman, A. Nanevski, and M. Sagiv. Modular reasoning on unique heap paths via effectively propositional formulas. In *POPL*, 2014.
15. S. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using SMT solvers. In *POPL*, 2008.
16. K. R. M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with chalice. In *FOSAD*, volume 5705 of *LNCS*, pages 195–222. Springer, 2009.
17. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM TOPLAS*, 1(2):245–257, 1979.
18. P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proc. CSL, Paris 2001*, volume 2142 of *LNCS*, 2001.
19. M. J. Parkinson and A. J. Summers. The relationship between separation logic and implicit dynamic frames. *Logical Methods in Computer Science*, 8(3), 2012.
20. J. A. N. Pérez and A. Rybalchenko. Separation logic + superposition calculus = heap theorem prover. In *PLDI*, pages 556–566. ACM, 2011.
21. R. Piskac, T. Wies, and D. Zufferey. Automating Separation Logic Using SMT. In *CAV*, volume 8044 of *LNCS*, pages 773–789. Springer, 2013.
22. X. Qiu, P. Garg, A. Stefanescu, and P. Madhusudan. Natural proofs for structure, data, and separation. In *PLDI*, pages 231–242, 2013.
23. J. Smans, B. Jacobs, and F. Piessens. VeriCool: An automatic verifier for a concurrent object-oriented language. In *FMOODS*, volume 5051 of *LNCS*, pages 220–239. Springer, 2008.
24. J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP*, volume 5653 of *LNCS*, pages 148–172. Springer, 2009.
25. V. Sofronie-Stokkermans. Hierarchic reasoning in local theory extensions. In *CADE*, 2005.
26. N. Totla and T. Wies. Complete instantiation-based interpolation. In *POPL*. ACM, 2013.
27. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn. Scalable shape analysis for systems code. In *CAV*, 2008.