Automatic Verification of Iterated Separating Conjunctions using Symbolic Execution

Peter Müller, Malte Schwerhoff, and Alexander J. Summers

Department of Computer Science, ETH Zurich, Switzerland
{peter.mueller, malte.schwerhoff, alexander.summers}@inf.ethz.ch

Abstract. In permission logics such as separation logic, the iterated separating conjunction is a quantifier denoting access permission to an unbounded set of heap locations. In contrast to recursive predicates, iterated separating conjunctions do not prescribe a structure on the locations they range over, and so do not restrict how to traverse and modify these locations. This flexibility is important for the verification of random-access data structures such as arrays and data structures that can be traversed in multiple ways such as graphs. Despite its usefulness, no automatic program verifier natively supports iterated separating conjunctions; they are especially difficult to incorporate into symbolic execution engines, the prevalent technique for building verifiers for these logics.
In this paper, we present the first symbolic execution technique to support general iterated separating conjunctions. We propose a novel representation of symbolic heaps and flexible support for logical specifications that quantify over heap locations. Our technique exhibits predictable and fast performance despite employing quantifiers at the SMT level, by carefully controlling quantifier instantiations. It is compatible with other features of permission logics such as fractional permissions, abstract predicates, and abstraction functions. Our technique is implemented as an extension of the Viper verification infrastructure.

1 Introduction

Permission logics such as separation logic [17] and implicit dynamic frames [18] associate an access permission with each memory location in order to reason about shared mutable state. Dynamic heap data structures require specifications to denote access permissions to a statically-unknown set of locations. Such specifications are typically expressed in existing tools using recursive predicates [14], which work well so long as the traversal of the data structure matches the definition of the predicate. However, access patterns that do not follow the predicate structure (*e.g.*, traversing a doubly-linked list from the end) or that follow no specific order (*e.g.*, random access into an array) are difficult to handle in existing program verifiers, requiring programmers to provide substantial manual proof steps (for instance, as ghost code) to bridge the mismatch between the program's access pattern and the imposed predicate structure.

Iterated separating conjunction [17] (hereafter, ISC) is an alternative way to denote properties of a set of heap locations, which has for instance been used in by-hand proofs to denote locations of arrays [17], cyclic data structures [3,22], the objects stored in linked lists [7], and graph algorithms [22]. Unlike recursive predicates, an ISC does not prescribe any particular traversal order.

Despite its usefulness and inclusion in early presentations of separation logic, no existing program verifier supports general ISCs directly. Among the tools based on symbolic execution, Smallfoot [2] does not support ISC; VeriFast [21] and jStar [7] allow programmers to encode some forms of ISC via abstract predicates that can be manipulated by auxiliary operations and lemmas (in VeriFast) or tailored rewrite rules (in jStar). For arrays, this encoding is partially supported by libraries. However, in the general case, programmers need to provide the extra machinery, which significantly increases the necessary manual effort. Among the verifiers based on verification condition generation, GrassHopper [15] does not support ISC; Chalice [11] supports only a restricted form (ranging over all objects stored in a sequence). VeriCool uses an encoding that leads to unreliable behaviour of the SMT solver [20, p. 46].

In this paper, we present the first symbolic execution technique that directly supports general forms of ISC. Our technique is compatible with other features of permission logics: it supports fractional permissions [5], such that a heap location may be ranged over by several ISCs, and allows ISC to occur in predicate bodies and in preconditions of abstraction functions [8].

This combination of features allows one to specify and verify challenging examples such as graph-marking algorithms that so far were beyond the scope of automated verifiers based on permission logics (see App. B).

Our main technical contributions are: (1) a novel representation of the partial heaps denoted by an ISC, along with algorithms to manipulate this representation; (2) a technique to preserve across heap changes (to frame) the values of expressions that depend on the unbounded set of heap locations denoted by ISCs; (3) an SMT encoding that carefully controls quantifier instantiations; (4) an implementation of our approach in the Viper verification infrastructure [13].

Outline. In the next section, we explain the main technical challenges our work addresses, and illustrate them with a simple motivating example. Our design for a symbolic heap that can represent permissions described by ISCs is presented in Sec. 3. We explain the symbolic evaluation of expressions and framing with respect to this heap representation in Sec. 4. In Sec. 5, we discuss how we control quantifier instantiations. Sec. 6 presents an evaluation of our implementation. We conclude in Sec. 7.

2 Technical Challenges

Permission logics ensure that a heap location is accessed only when the corresponding permission is held. Dedicated assertions denote the permission to a heap location $e.f$, written as $e.f \mapsto \_$ in separation logic and as the *accessibility predicate* `acc(`$e.f$`)` in implicit dynamic frames; we use the latter in this paper.

These logics include a *separating conjunction* $*$, expressing that the permissions denoted by the two conjuncts must be disjoint. For instance, an assertion **acc**$(x.f) * $**acc**$(y.f)$ implies the disequality $x \neq y$. Many permission logics allow permissions to be split into fractions, and to re-assemble fractions into a full permission. In these logics, any non-zero permission allows read access to a location, whereas write access requires the full permission. When appropriate permissions are held, assertions may also constrain the value of a heap location (for instance, $x.f > 3$); assertions that do not contain accessibility predicates are called *pure*. We use the terms *pure assertion* and *expression* synonymously.

Verification of many program constructs can be modelled by two basic operations. *Inhaling* an assertion A adds the permissions denoted by A to the current state and assumes the pure assertions in A. *Exhaling* an assertion A checks that the current state satisfies the pure assertions in A; it also checks that the state contains the permissions denoted by A and removes them. As soon as permission to a heap location is no longer held, information about its value cannot be retained. Inhale and exhale can be seen as the permission-aware analogues of assume and assert statements [11]; they are sometimes called produce and consume [19]. Using these operations, a method call (for example) can be encoded by exhaling the method precondition and then inhaling its postcondition.

Building a verification tool for a permission logic requires effective solutions to the following *technical challenges*:

1. How to model the program state, including permissions and values?
2. How to check for a permission in a state?
3. How to add and remove permissions to and from a state?
4. How to evaluate (heap-dependent) expressions in a state?
5. When to preserve (frame) an expression's value across heap changes?

In the remainder of this section, we summarize how existing verifiers solve these challenges for logics without ISC and then explain how providing support for ISC complicates these challenges.

2.1 Smallfoot-style Symbolic Execution

Smallfoot [2] introduced a symbolic execution technique that has become the state-of-the-art way of building verifiers for permission logics. It provides simple and efficient solutions to the technical challenges above: (1) A symbolic state consists of a set of *heap chunks*, and a set of *path conditions*. A heap chunk has the form $o.f \mapsto [v, p]$, mimicking separation logic's points-to predicates. It records a receiver value o, a field name f, a *location value* v representing the value stored in location $o.f$, and a permission amount p. A permission amount is a value between 0 and 1 (inclusive); intermediate values can be used to support fractional permissions. Here, o, v, and p are (immutable) symbolic values. Path conditions are boolean constraints on these symbolic values. An SMT solver is used to answer queries about the path conditions, e.g. equality of symbolic values. (2) Checking for a permission entails iterating through the heap chunks and finding those with

```
method Replace(a: Int[], left: Int, right: Int, from: Int, to: Int)
  requires 0 <= left < right <= a.length
  requires forall i: Int :: left <= i < right ==> acc(a[i])
  ensures  forall i: Int :: left <= i < right ==> acc(a[i])
  ensures  forall i: Int :: left <= i < right ==>
               (old(a[i]) == from ? a[i] == to : a[i] == old(a[i]))
{
  if (right - left <= 1) {
    if(a[left] == from) { a[left] := to }
  } else {
    var mid := left + (right - left) / 2
    Replace(a, left, mid, from, to) || Replace(a, mid, right, from, to)
  }
}
```

Fig. 1. A parallel replace operation on array segments. The second precondition and the first postcondition denote access permissions to the elements of the array. The `forall` quantifier in these conditions denotes an ISC: the body of the quantifier includes accessibility predicates (of the form `acc(a[i])`). The second postcondition uses a regular (pure) quantifier to specify the functional behaviour of the method. Here, `old` expressions let the postcondition refer to values in the prestate; the access permissions for these expressions come from the second precondition.

matching receiver-field pairs. (3) Removing a permission is modelled by removing the corresponding chunk(s), and adding a permission modelled by adding a heap chunk with a fresh symbolic location value. (4) Evaluating a heap lookup $e.f$ yields the location value of the chunk for $e.f$ (and is not permitted if no such chunk exists). (5) Framing the value of such expressions happens implicitly so long as the same heap chunk remains in the symbolic heap.

In order to specify unbounded heap structures, the Smallfoot approach has been extended to handle user-defined recursive predicates. In successor tools such as VeriFast [21], jStar [7], and Viper [13], heap chunks may also represent predicate instances. Smallfoot-style symbolic execution has also been extended to support heap-dependent pure functions in the assertion language [19]. E.g., the operations of a list class may be specified in terms of an `itemAt` function. Such functions include a precondition that requires permission to all locations read by the function body; this information is used to frame function applications.

These extensions increase the expressiveness of permission logics significantly, but are not sufficient to simply specify and automatically reason about important data structures such as arrays and graphs: this requires support for ISCs.

2.2 Iterated Separating Conjunction

Fig. 1 illustrates the usage of ISCs: method `Replace` replaces all occurrences of integer `from` by integer `to` in the segment of array `a` between `left` and `right`. The recursive calls to smaller array segments are performed concurrently using parallel composition ||. The second precondition requires access permissions for all elements in the array segment, and the first postcondition returns these permissions to the caller; both are expressed using ISC. The second postcondition specifies the functional behaviour of the method using an **old**-expression to refer to the prestate of a method; this pure assertion needs heap-dependent expressions under a quantifier.

Verifying the example entails splitting the symbolic state described by the ISC in the precondition in order to exhale the preconditions of the recursive calls, and to re-combine the states resulting from inhaling the postconditions of these calls after the parallel composition, in order to prove the callee's postcondition.

Providing support for ISCs complicates each of the five technical challenges discussed above:

1. Heap chunks must be generalised to denote permission to an unbounded number of locations simultaneously, and encode a symbolic value per location (for instance, to represent the values of each array location in Fig. 1).
2. Exhaling an ISC requires checking permission for an unbounded number of heap locations; these could be spread across multiple heap chunks, as in the case of exhaling the postcondition of `Replace`.
3. Removing permissions from a generalised chunk may affect only some of the locations to which it provides permission. For example, when exhaling the precondition of the first recursive call to `Replace`, the permissions required for the second call must be retained in the symbolic state.
4. Evaluating heap-dependent expressions under quantifiers may rely on symbolic values from multiple heap chunks. For example, proving the second postcondition of `Replace` requires information from both recursive calls.
5. Framing in existing Smallfoot-style verifiers requires that heap-dependent expressions depend only on a bounded number of symbolic values (which can include representations of predicate instances [19]). However, this requirement is too strong for pure quantifiers over heap locations and for functions whose preconditions use ISCs to require access to an unbounded set of locations (see App. C for an example).

Our technique is the first to provide automatic solutions to these challenging problems. Sec. 3 tackles the first 3 problems; Sec. 4 tackles the remaining 2.

3 Treatment of Permissions

We consider the following canonical form of source-level assertion for denoting an ISC: **forall** $x\colon T :: c(x) \Rightarrow$ **acc**$(e(x).f, p(x))$, in which $c(x)$ is a boolean expression, $e(x)$ a reference-typed expression, and $p(x)$ an expression denoting a permission amount. More complex assertions can be desugared into this canonical form, for instance, iterating over the conjunction of two accessibility predicates can be encoded by repeating the quantification over each conjunct. For simplicity, we do not consider nested ISCs, but an extension is possible. Our canonical form is sufficient to directly model quantifying over all receivers in a set (useful for graph examples) or over integer indices into an array, as shown in Fig. 1.

The permission expression $p(x)$ may be a complex expression including conditionals, and need not evaluate to the same value for each instantiation of x.

This enables us to model complex access patterns such as requiring non-zero permission to every nth slot of an array, which is for instance important for GPU verification [4]. ISCs are complemented by unrestricted pure quantifiers over potentially heap-dependent expressions, which are essential for specifying functional properties.

In this section, we present the first key ingredient of our symbolic execution technique: a representation for ISCs as part of the verifier's symbolic state along with algorithms to manipulate this representation.

3.1 Symbolic Heap Representation

As explained in Sec. 2.1, Smallfoot-style heap chunks $o.f \mapsto [v, p]$ consist of a receiver value o, a field name f, a location value v and a permission amount p. A naïve generalisation of this representation would be to make o, v, and p functions of the bound variable of an ISC. However, such a representation has severe drawbacks. Checking whether a heap chunk provides permission to a location $y.f$ (challenge 2 above) amounts to the existential query $\exists x.o(x) = y$; SMT solvers provide poor support for such existential queries. In the presence of fractional permissions, determining *how much* permission such a heap chunk provides is worse still, requiring to calculate the sum of *all* $p(x_i)$ such that x_i satisfies the existential query.

Our design avoids these difficulties with a simple restriction: we require the receiver expressions $e(x)$ in an ISC to be *injective* in x, for all values to which the ISC provides permission. Under this restriction, we can soundly assume that the mapping between the bound variable x and receiver expression $e(x)$ is *invertible* for such values, by some function e^{-1}. We can then represent an ISC over receivers $r = e(x)$ directly, essentially by replacing x by $e^{-1}(r)$ throughout.

Our resulting design is to use *quantified chunks* of the form $r.f \mapsto [v(r), p(r)]$, in which r (which is implicitly bound in such a chunk) plays the role of a quantified (reference-typed) receiver. Such a quantified chunk represents $p(r)$ permission to all locations $r.f$; $p(r)$ may be any expression denoting a permission amount. The *domain* of a quantified chunk is the set of field locations $r'.f$ for which $p(r') > 0$. The *values* of these locations are modelled by the function v, which we call a *value map* and explain in Sec. 4. A *symbolic heap* is a set of quantified chunks; a *symbolic state* is a symbolic heap plus a set of path conditions, as usual.

Under our injectivity restriction, we represent a source-level assertion of the form **forall** $x\colon T :: c(x) \Rightarrow$ **acc**$(e(x).f, p(x))$ using a quantified chunk of the form $r.f \mapsto [v(r), (\underline{c}(e^{-1}(r)) \; ? \; \underline{p}(e^{-1}(r)) : 0)]$ for a suitable value map v and inverse function e^{-1}. Whenever necessary to avoid ambiguity, we use underlined expressions to denote the results of symbolically evaluating corresponding source-level expressions; with the exception of heap-dependent expressions (see Sec. 4.1), this evaluation is orthogonal to the contributions of this paper.

Our injectivity restriction does not limit the data structures that can be handled by our technique, provided specifications are expressed appropriately. The restriction applies to memory *locations*, not to the *values* stored in the locations. Many examples such as ISCs ranging over array indices or elements of

```
inhale(h₀, π₀, forall x: T :: c(x) ⇒ acc(e(x).f,p(x))) ⇝
    Let y be a fresh symbolic constant of type T
    /* Symbolically evaluate source-level expressions                   */
    var (π₁, c̲(y)) := eval(h₀, π₀, c(y))
    var (π₂, e̲(y)) := eval(h₀, π₁ ∪ {c̲(y)}, e(y))
    var (π₃, p̲(y)) := eval(h₀, π₂, p(y))
    var π₄ := π₃ \ {c̲(y)}

    /* Introduce inverse function                                       */
    Let e⁻¹ be a fresh function of type T → Ref
    var π₅ := π₄ ∪ {∀r: Ref · c̲(e⁻¹(r)) ⇒ e̲(e⁻¹(r)) = r}      /* (Inv-1) */
    var π₆ := π₅ ∪ {∀x: T · c̲(x) ⇒ e⁻¹(e̲(x)) = x}             /* (Inv-2) */

    Let v be a fresh value map
    var h₁ := h₀ ∪ {r.f ↦ [v(r), c̲(e⁻¹(r)) ? p̲(e⁻¹(r)) : 0]}
    return (h₁, π₆)

exhale(h₀, π₀, forall x: T :: c(x) ⇒ acc(e(x).f,p(x))) ⇝
    Let y be a fresh symbolic constant of type T
    /* Symbolically evaluate source-level expressions (as above)        */

    /* Check injectivity of receiver expression                         */
    Let y₁, y₂ be fresh symbolic constants of type T
    check π₄ ⊨ c̲(y₁) ∧ c̲(y₂) ∧ e̲(y₁) = e̲(y₂) ⇒ y₁ = y₂

    /* Introduce inverse function (as above)                            */

    /* Remove permissions                                               */
    var h₁ := remove(h₀, π₆, f, (λr · c̲(e⁻¹(r)) ? p̲(e⁻¹(r)) : 0))
    return (h₁, π₆)
```

Fig. 2. Symbolic execution rules for inhaling and exhaling ISCs. The **check** instruction submits a query to the SMT solver. If the proof obligation does not hold, it aborts with a verification failure. The `eval` function evaluates an expression in a symbolic state and yields updated path conditions and the resulting symbolic expression, see Sec. 4. In both rules, the constraint $\underline{c}(y)$ is temporarily added to the path conditions used during the evaluation of $e(y)$ and $p(y)$; these expressions may be well-formed only under this additional constraint.

a set naturally satisfy the restriction. Ranges that may contain duplicates (for instance, the fields of all objects stored in an array) can be encoded by mapping them to a set (thereby ignoring multiplicities) or by using complex permission expressions p that reflect multiplicities appropriately.

3.2 Inhaling and Exhaling Permissions

Using the symbolic heap design explained above, we define the operations for inhaling and exhaling ISCs in Fig. 2. The `inhale` operation takes a symbolic heap h_0, path conditions π_0, and an ISC, and returns an updated heap and path conditions. Following the encoding described in the previous subsection, the operation introduces a (fresh) inverse function e^{-1}, which is constrained as the partial inverse of the (evaluated) receiver expression $\underline{e}(x)$ by adding the

```
def remove(h_0, π_0, f, q):
    Let h_f ⊆ h_0 be all chunks in the given state for field f
    var h'_f := ∅                                          /* Processed chunks */
    var q_needed := q                              /* Permissions still to take */
    foreach (r.f ↦ [v_i(r), q_i(r)]) ∈ h_f do:
        /* Determine the permissions to take from this chunk                  */
        var q_current := (λr · min(q_i(r), q_needed(r)))

        /* Decrease the permissions still needed                              */
        q_needed := (λr · q_needed(r) − q_current(r))

        /* Add an updated chunk to the processed chunks                       */
        h'_f := h'_f ∪ {r.f ↦ [v_i(r), (q_i(r) − q_current(r))]}

    /* Check that sufficient permissions were removed                         */
    check π_0 ⊨ ∀r · q_needed(r) = 0

    return (h_0\h_f) ∪ h'_f
```

Fig. 3. The remove operation. The argument q maps references to permission amounts. The operation checks that the symbolic heap contains at least $q(r)$ permission for each location $r.f$ and removes it.

constraints INV-1 and INV-2 to the path conditions. We will discuss controlling the instantiation of these quantifiers (and others introduced by our technique) in Sec. 5. The fresh value map v models the (thus far unknown) values of the heap locations in the domain of the new quantified chunk, which is added to the symbolic heap h_0.

To encode our example (Fig. 1) in a tool without native array support, we model the array slots as a set of ghost objects, each with a field `val` (representing the slot's value). That is, an array location $a[i]$ is modelled by the location `A(i).val`, where `A` is an injective function mapping indices to these ghost objects. Full details of the encoding of the running example are given in App. C. Following Fig. 2, inhaling the second precondition (at the start of checking the method body) entails introducing an inverse function a^{-1} mapping array locations back to corresponding indices, and then adding a quantified chunk $r.\mathtt{val} \mapsto [v(r), (\underline{\mathtt{left}} \leqslant \underline{a}^{-1}(r) < \underline{\mathtt{right}}\ ?\ 1 : 0)]$. Correspondingly, at the program point after the two recursive calls, the symbolic heap will contain two quantified chunks: one for each array segment.

The `exhale` operation is initially similar to `inhale`, one difference being that the injectivity of the receiver expression is checked before defining the inverse function. Removing permissions is more complex than adding permissions because it may involve updates to many existing quantified chunks in the symbolic state. This operation is delegated to the auxiliary operation `remove`, shown in Fig. 3.

`remove` takes as inputs an initial symbolic heap h_0 and path conditions π_0, a field name f, and a function q that yields for each reference r the permission amount for location $r.f$ to be removed. `remove` fails with a verification error if the initial heap does not contain the permissions in q, and otherwise returns an updated symbolic state. This is achieved by iterating over all available chunks for field f, greedily taking as much of the still-required permissions (q_{needed}) as possible from the current chunk ($q_{current}$). Updating the chunks is expressed via pointwise-defined functions describing the corresponding permission amounts; they involve permission arithmetic, but no existential quantifiers, and can be handled efficiently by the underlying SMT solver. After this iteration, `remove` checks that all requested permissions have been removed.

In our array example (Fig. 1), we exhale the second precondition before each recursive call; this requires finding the appropriate permissions from the (single) quantified chunk in the state at this point, and removing them. Dually, when exhaling the postcondition at the end of the method body, all permissions from both of the two quantified chunks yielded by the recursive calls must be removed: the iteration in the `remove` algorithm achieves this.

3.3 Integrating Predicates with Iterated Separating Conjunctions

Predicates are a standard feature of verification tools for permission logics (including the Viper infrastructure on which our implementation is built); they integrate simply with our support for ISCs. Fig. 4 shows an example of a predicate definition, parameterised by a set of nodes, that defines a graph in terms of ISCs and closure properties over the given set of nodes. The Viper tools on which we build require explicit ghost operations to exchange a predicate instance $P(e)$ for its body (via an operation **`unfold`** $P(e)$), and vice versa (via an operation **`fold`** $P(e)$); this is a standard way to handle possibly-recursive predicates. In terms of the underlying verifier, an operation **`fold`** $P(e)$ essentially corresponds to **`exhale`** $P_{body}(e)$ followed by **`inhale`** $P(e)$, and dually for **`unfold`** $P(e)$. Since our support for ISCs is expressed in terms of inhale and exhale rules, it naturally integrates with Viper's existing way of handling predicates; our implementation supports predicates with ISCs in their bodies.

Our implementation does not yet support predicates inside ISCs, but our presented technique extends straightforwardly to support this. Inhaling an ISC which ranges over predicate instances yields, just as for accessibility predicates for fields, a new quantified chunk. An **`unfold`** of a predicate belonging to such a chunk can be handled by exhaling the predicate instance (removing it from the chunk's permissions), and then inhaling the predicate's body. Folding an instance corresponds to inhaling a quantified predicate chunk that provides permissions to the single instance. We plan to extend our implementation to also support this feature combination.

4 Treatment of Symbolic Values

So far we have addressed the first three technical challenges described in Sec. 2 by presenting a novel heap representation for ISCs together with algorithms that let the verifier efficiently add, as well as check for and remove permissions. In this section we present our solution to the remaining two challenges, concerned with the evaluation and framing of expressions.

```
predicate Graph(nodes: Set[Ref]) {
     (forall n: Ref :: n in nodes ==> acc(n.left))
  && (forall n: Ref :: n in nodes ==> acc(n.right))
  && (forall n: Ref :: n in nodes && n.left != null ==> n.left  in nodes)
  && (forall n: Ref :: n in nodes && n.right != null ==> n.right in nodes)
}
```

Fig. 4. A predicate defining a graph in terms of ISCs and closure properties over a given set of nodes (that form the graph).

def summarise(h_0, f):
  Let $h_f \subseteq h_0$ be all quantified chunks in the given heap for field f
  Let v be a fresh value map
  var $def := \varnothing$ /* Value summary path conditions */
  var $perm := \lambda r \cdot 0$ /* Permission summary */
  foreach $(r.f \mapsto [v_i(r), q_i(r)]) \in h_f$ **do:**
    $def := def \cup \{\forall r \cdot 0 < q_i(r) \Rightarrow v(r) = v_i(r)\}$ /* (VmDefEq) */
    $perm := \lambda r \cdot (perm(r) + q_i(r))$
  return $(v, def, perm)$

Fig. 5. The `summarise` operation introduces a fresh value map for field f and constrains it according to the value maps of all heap chunks for f. It also returns a function summarising the permissions held for the field f.

4.1 Symbolic Evaluation of Heap-Dependent Expressions

Quantified chunks $r.f \mapsto [v(r), q(r)]$ represent value information via the value map v. The existence of such a chunk in a symbolic heap allows the evaluation of a read of field f for any receiver in the domain of the heap chunk, to an application of the value map. Intuitively, v represents a partial function from this domain to values (of the type of the field f). Since SMT solvers typically do not natively support partial functions, we model value maps as under-specified total functions from the receiver reference (the field f is fixed) to the type of f. We *apply* these functions only to references whose f field location is in the chunk's domain. This is why the `exhale` algorithm (Fig. 2) does not need to explicitly remove information about the values stored in the locations whose permissions are removed; the underlying total function still represents appropriate values for the new (smaller) domain.

Summarising Value Maps. Inhaling permissions adds a fresh heap chunk with a fresh value map (see Fig. 2). Therefore, a symbolic heap may contain multiple chunks for the same field, each with its own value map. In the presence of fractional permissions, the domains of these chunks may overlap such that the value of one location $x.f$ may be represented by multiple value maps. Similarly, the value of $x.f$ may be represented by multiple maps when the receiver x is quantified over and the permissions to different instantiations of the quantifier are recorded in different chunks. Therefore, all of these value maps need to be considered when evaluating such a field access.

In order to incorporate information from all relevant chunks, and provide a simple translation for field-lookups, we summarise the value maps for all chunks for a field f *lazily* before we evaluate an expression $e.f$. This summarisation is defined by the `summarise` operation in Fig. 5. For each quantified chunk with the appropriate field, it equates a newly-introduced value map with the value map in the chunk at all locations in the chunk's domain. Analogously, it builds up a permission expression summarising the permissions held per receiver, across all heap chunks for the field f; we use this to check whether a field access is permitted. Note that the definition of `summarise` does not depend on path conditions, only on the symbolic heap; it can be computed without querying the SMT solver.

Symbolic Evaluation. Symbolic evaluation of expressions is defined by an operation `eval`, which takes a symbolic heap, path conditions, and an expression, and yields updated path conditions and the symbolic value of the expression; the cases for field lookup and pure quantifiers are given in Fig. 6 (some additional cases can be found in App. A). Using the `summarise` operation, we can simply define the evaluation of a field lookup, as shown first in Fig. 6. To evaluate such an expression, we check that at least some permission to the field location is held in the current symbolic heap, and use the value map generated by `summarise` to define the value of the field lookup. Via the path conditions generated by `summarise`, any properties known about the value maps of the corresponding quantified chunks will also be known about the resulting symbolic value. Our implementation memoizes `summarise`, avoiding the duplication of the function declarations and path conditions defining the value and permission maps.

Evaluating pure quantifiers is handled by replacing the bound variable with a fresh constant and evaluating the quantifier body. Additional path conditions generated during this recursive evaluation might mention the fresh constant; these are universally quantified over when returning the path conditions.

Inhale, Exhale, and Field Writes. Inhaling and exhaling pure boolean expressions is implemented by first symbolically evaluating the expression and then either adding the resulting symbolic expression to the path conditions or checking it, respectively (see App. A).

A field write $e_1.f$ `:=` e_2 is desugared as: **`exhale acc`**`(`$e_1.f$`);` **`inhale acc`**`(`$e_1.f$`);` **`inhale`** $e_1.f$ `==` e_2. The exhale checks that the heap has the required permission and removes it; the inhales create a new chunk with the previously-removed permission and constrain the associated value map such that it maps receiver $\underline{e_1}$ to the value of $\underline{e_2}$. For example, the field write `a[left] := to` in Fig. 1 is executed in a symbolic heap with a single quantified chunk that provides full permissions to each array location. After the field write has been executed, the heap contains two quantified chunks: the initial one, still providing full permissions to each array location *except for* `a[left]` (and with an unchanged value map), and a second one that provides permissions to `a[left]` only, with a fresh value map representing the updated value.

```
eval(h_0, π_0, e.f) ⤳
    var (π_1, e̲) := eval(h_0, π_0, e)
    var (v, def, perm) := summarise(h_0, f)
    check π_1 ⊨ 0 < perm(e̲)
    return (π_1 ∪ def, v(e̲))

eval(h_0, π_0, forall x :: e(x)) ⤳
    Let y be a fresh symbolic constant
    var (π_1, e̲(y)) := eval(h_0, π_0, e(y))
    return ({b ∈ π_1 | y ∉ FV(b)} ∪ {∀x · (⋀_{b∈π_1, y∈FV(b)} b[x/y])}, ∀ x · e̲(x))
```

Fig. 6. Symbolic evaluation of field reads and pure quantifiers.

4.2 Framing Heap-Dependent Expressions

Permissions provide a straightforward story for framing the values of heap locations (and pure quantifiers over these): so long as the symbolic state contains *some* permission to a field location, its value will be preserved. However, framing heap-dependent functions is more complicated [19,8]. The value of a function can be framed so long as all locations the function depends on remain unchanged. To express a function's dependency on the heap, its precondition must require permission to all locations its implementation may read. For any given function application, the symbolic values of these locations are called the *snapshots* of the function application. Consequently, two function applications yield the same result if they take the same arguments and have equal snapshots. One can thus model a heap-dependent function at the SMT level by a function taking snapshots as additional arguments [19].

ISCs complicate this approach because a function whose precondition contains an ISC may depend on an unbounded set of heap locations. The values of these locations cannot be represented by a fixed number of snapshots. It is also not possible to represent them as a value map since these are modelled at the SMT level as total functions, causing two problems. First, requiring equality of total functions would include locations the heap-dependent function does not actually depend on; since the values for these locations are under-specified, the equality check would often fail even when the function value could be soundly framed. Second, a function cannot be used as a function argument, nor compared for equality in the first-order logic supported by SMT solvers.

We address the first problem by modelling snapshots as *partial* functions called *partial value maps*, and the second by applying *defunctionalisation* [16]. That is, we model a partial value map for a field f of type T as a value of an (uninterpreted) type PVM, together with a function $domain_f\colon\ PVM \to Set[Ref]$ for the domain of the partial value map, and a function $apply_f\colon\ PVM \times Ref \to T$ for the result of applying a partial value map to a receiver reference. We also include an extensionality axiom for partial value maps, allowing us to prove equality when two partial value maps are equal as partial functions.

$\forall r\colon Ref \cdot$	$\{\, v(r)\}\{\, v_i(r)\}\quad 0 < q_i(r) \Rightarrow v(r) = v_i(r)$	/* (VmDefEq) */
$\forall r\colon Ref \cdot$	$\{e^{-1}(r)\}\quad \underline{c}(e^{-1}(r)) \Rightarrow \underline{e}(e^{-1}(r)) = r$	/* (Inv-1) */
$\forall x\colon T \cdot$	$\{e(x)\}\quad \underline{c}(x) \Rightarrow e^{-1}(\underline{e}(x)) = x$	/* (Inv-2) */

Fig. 7. Example triggers used in our SMT encoding.

Following the prior work, we model a heap-dependent function via a function at the SMT level, with a partial value map as additional snapshot argument for each ISC required in the function's precondition. For each application of such a function, we check that the current state contains all permissions required by the function precondition. If this is the case, we process each ISC in the precondition in turn. For an ISC for a field f, we employ the `summarise` operation (Fig. 5) to summarise the value information v for the field f in the current symbolic state, and introduce a fresh constant *pvm* of type PVM. We constrain $\mathit{domain}_f(\mathit{pvm})$ to yield the set of references in the domain of the ISC, and for all receivers r in this domain, assume $\mathit{apply}_f(\mathit{pvm}, r) = v(r)$. *pvm* is then used as a snapshot argument to the translated function.

5 Controlling Quantifier Instantiations

When generating quantifiers for an SMT solver, it is important to carefully control their instantiation [8,10,12] by providing syntactic triggers. A quantifier $\forall x \cdot P(x)$ may be decorated with a *trigger* $\{f(x)\}$, which instructs the solver to instantiate x with a term e only if $f(e)$ is a term encountered by the solver during the current proof effort. Triggers must be chosen carefully: enabling too few instantiations may cause examples to fail unexpectedly, while too many may lead to unreliable performance or even non-termination of the solver (see also Sec. 6).

We carefully select triggers for all quantifiers generated by our technique (although we have omitted them from the presentation so far). Fig. 7 shows three representative examples. The path condition VmDefEq relates the value map introduced by the `summarise` operation to the value maps of heap chunks (Fig. 5). The two triggers express alternatives: they allow instantiating the path condition if *either* of the two value maps have been applied to the term instantiating r. This design allows us to derive relationships between two evaluations of an expression, which introduce two summary value maps. Instantiating VmDefEq in both directions allows us to relate these value maps via the value maps of heap chunks.

The next two examples define the inverse function of a receiver expression (see Fig. 2). The trigger $e^{-1}(r)$ for Inv-1 is essential for relating occurrences of the inverse function to the original expression e. The case of Inv-2 is almost symmetrical, but with extra technicalities. Since e comes from the source program, it may not be an expression allowed as a trigger. Trigger terms must typically include at least one function application (if $e(x)$ were simply x, this could not be used), and no built-in operators such as addition. In the former case, we use $v(x)$ as a trigger, where v is the value map of the relevant chunk; the quantifier

Program	Size (LOC)	Time (s)	w/o memoization	w/o triggers
`arraylist`	114	1.93	−7.29%	−16.53%
`quickselect`	132	2.51	+24.44%	−4.23%
`binary-search`	47	0.31	+14.15%	−8.94%
`graph-copy`	120	1.81	+14.93%	+21.21%
`graph-marking`	53	1.71	+41.29%	−30.95%
`longest-common-prefix`	34	0.19	+6.51%	−10.73%
`max-elimination`	59	0.50	+45.41%	−0.07%
`max-standard`	53	0.24	+16.40%	+2.43%
`parallel-replace`	56	0.27	+3.71%	−6.12%

Fig. 8. Performance evaluation of our implementation on verification challenges. Lines of code (LOC) does not include blank lines and comments. Column "Time (s)" gives runtimes of the base version of our implementation; columns "w/o memoization" and "w/o triggers" show the % difference in time relative to these.

	No.	Size	Time		w/o memoization		w/o triggers	
	Files	Mean	Mean	Max	Mean	Max	Mean	Max
Program Set	(#)	(LOC)	(s)	(s)	(±)	(s)	(±)	(s)
VerCors	65	104	0.72	11.81	+0.92%	15.71	-4.40%	8.83
Regressions	82	34	0.22	3.41	+0.58%	3.81	-2.24%	3.38

Fig. 9. Performance evaluation of our implementation on two sets of programs: the "VerCors" set contains (non-trivial) programs generated by the VerCors tool, "Regressions" contains (usually simple) regression tests; column "Files" displays the number of files per program set. All input files are available in the supplementary material.

will then be instantiated whenever we look up a value from the chunk, which is when we need the definition of the inverse function. In the latter case, we resort to allowing the underlying tools select trigger terms.

6 Evaluation

We have implemented our technique as an extension of the Viper verification infrastructure [13]; an online version is available [1]. To evaluate the performance of our technique, we ran experiments with three kinds of input programs: (i) 9 hand-coded verification problems involving arrays and graphs, including our running example (see App. B for details), (ii) 65 examples generated by the VerCors project at the University of Twente [4], which uses our implementation to encode GPU verification problems, and (iii) 82 additional regression tests.

Fig. 8 shows the results for (i), and Fig. 9 those for (ii) and (iii). We performed our experiments on an Intel Core i7-4770 3.40GHz with 16GB RAM machine running Windows 7 x64 with an SSD. The reported times are averaged over 10 runs of each verification (with negligible standard deviations). Timings do not include JVM start-up: we persist a JVM across test runs using the Nailgun tool.

Our experiments show that our implementation is consistently fast: all examples verify in a few seconds (we also observed consistent runtimes, that is, negligible standard deviations). Since SMT encodings sometimes exhibit worse performance for *failed* verification attempts, we also tested 4 variants of each example from Fig. 8 in which we seeded errors; in all cases the errors were detected with lower runtimes (the verifier halts as soon as an error is detected).

To measure the effect of memoizing calls to `summarise`, we disabled this feature and measured the difference in runtimes over the same inputs. As shown in the "w/o memoization" columns, disabling this optimisation typically increases the runtime, but not enormously; a likely explanation for the relatively small difference is that `summarise` performs the iteration over quantified chunks efficiently, without querying the SMT solver. The number of quantified chunks in a given symbolic state is also typically kept small: the tool performs modular verification per method/loop body, and we eagerly remove any quantified chunks that no longer provide permissions (after an exhale).

To evaluate the importance of our use of triggers for controlling quantifier instantiations (see Sec. 5), we also compare with a variant of our implementation in which triggers are omitted, leaving this task to the underlying tools (that is, Viper and Z3 [6]). The relative times are shown in the "w/o triggers" columns. We observe that this variant typically *improves* verification time. However, the triggers chosen automatically by Viper and Z3 are too strict: 7% of the programs (11 out of the 156 original programs) fail spuriously in this version. This, as well as a general reduction in quantifier instantiations, explains the effect on the runtime: the longest-running example in our base implementation (averaging 11.82s) takes only 3s without our triggers, but wrongly fails to verify. The longest-running example in the variant without triggers takes 8.83s but also has a high standard deviation of 4.71s, suggesting that performance also becomes unpredictable when triggers are selected automatically. The triggers that we choose thus avoid spurious errors and provide predictable, fast performance.

7 Conclusions and Future Work

We have presented the first symbolic execution technique that supports ISCs. This feature provides the possibility of specifying random-access data structures and provides an alternative mechanism to recursive definitions which is essential in the common case when a data structure can be traversed in multiple ways. Our technique generalises Smallfoot-style symbolic execution and is, thus, applicable to other verifiers for permission logics using this common implementation technique.

As future work, we plan to build on our verification technique in four ways. First, we plan to extend our technique to support predicates under ISCs, as discussed in Sec. 3.3. Second, we plan to combine our verification technique with inference techniques that make use of ISCs, such as the shape analysis developed by Lee *et al.* [9]. Third, we plan to support `foreach` statements that perform an operation (e.g. unfolding a predicate) on each instance of a quantifier without requiring a loop (and invariant). Such statements require permissions that can be expressed using ISCs. Fourth, we plan to integrate support for aggregates in pure assertions [10], which provide another means for specifying functional properties over locations described by an ISC.

References

1. Online appendix / implementation. `http://viper.ethz.ch/examples/cav16/`.
2. J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, volume 4111 of *LNCS*, pages 115–137. Springer, 2006.
3. L. Birkedal, N. Torp-Smith, and J. C. Reynolds. Local reasoning about a copying garbage collector. In N. D. Jones and X. Leroy, editors, *POPL*, pages 220–231. ACM, 2004.
4. S. Blom and M. Huisman. The VerCors tool for verification of concurrent programs. In C. B. Jones, P. Pihlajasaari, and J. Sun, editors, *FM*, volume 8442 of *LNCS*, pages 127–131. Springer, 2014.
5. J. Boyland. Checking interference with fractional permissions. In *SAS*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.
6. L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
7. D. Distefano and M. J. Parkinson J. jStar: Towards practical verification for Java. In *OOPSLA*, pages 213–226. ACM, 2008.
8. S. Heule, I. T. Kassios, P. Müller, and A. J. Summers. Verification condition generation for permission logics with abstract predicates and abstraction functions. In G. Castagna, editor, *ECOOP*, volume 7920 of *LNCS*, pages 451–476. Springer, 2013.
9. O. Lee, H. Yang, and K. Yi. Automatic verification of pointer programs using grammar-based shape analysis. In S. Sagiv, editor, *ESOP*, LNCS, pages 124–140, 2005.
10. K. R. M. Leino and R. Monahan. Reasoning about comprehensions with first-order SMT solvers. In S. Y. Shin and S. Ossowski, editors, *SAC*, pages 615–622. ACM, 2009.
11. K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In G. Castagna, editor, *ESOP*, volume 5502 of *LNCS*, pages 378–393. Springer-Verlag, 2009.
12. M. Moskal. Programming with triggers. In *SMT*, volume 375 of *ACM International Conference Proceeding Series*, pages 20–29. ACM, 2009.
13. P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *VMCAI*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.
14. M. Parkinson and G. Bierman. Separation logic and abstraction. In J. Palsberg and M. Abadi, editors, *POPL*, pages 247–258. ACM, 2005.
15. R. Piskac, T. Wies, and D. Zufferey. Grasshopper—complete heap verification with mixed specifications. In E. Ábrahám and K. Havelund, editors, *TACAS*, volume 8413 of *LNCS*, pages 124–139. Springer, 2014.
16. J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM Annual Conference—Volume 2*, ACM '72, pages 717–740. ACM, 1972.
17. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*. IEEE Computer Society Press, 2002.
18. 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.

19. J. Smans, B. Jacobs, and F. Piessens. Heap-dependent expressions in separation logic. In J. Hatcliff and E. Zucca, editors, *FMOODS*, volume 6117 of *LNCS*, pages 170–185. Springer, 2010.
20. J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.*, 34(1):2:1–2:58, 2012.
21. J. Smans, B. Jacobs, and F. Piessens. Verifast for Java: A tutorial. In D. Clarke, J. Noble, and T. Wrigstad, editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *LNCS*, pages 407–442. Springer, 2013.
22. H. Yang. An example of local reasoning in BI pointer logic: the Schorr-Waite graph marking algorithm. In *Proceedings of the SPACE Workshop*, 2001.

A Additional Definitions and Symbolic Execution Rules

Partial Value Maps. Fig. 10 shows background definitions related to partial value maps (see Sec. 4.2), which are emitted to the SMT solver before the verification starts. The background definitions include a type PVM and, per field declaration, a function $domain_f$ that denotes the domain of a partial value map, a function $apply_f$ that denotes applying a partial value map to a receiver to obtain the value of the corresponding field location, and an extensionality axiom stating that two partial value maps are equal if their domains agree and if they agree on the values in their domain.

1. Let FD be the set of all field declarations $f : T$ of a given program for which ISCs are used
2. Declare a type PVM
3. Declare a function $domain_f\colon\ PVM \to Set[Ref]$ per declaration $f : T \in FD$
4. Declare a function $apply_f\colon\ PVM \times Ref \to T$ per declaration $f : T \in FD$
5. Declare the following extensionality axiom per declaration $f : T \in FD$:

$$\begin{aligned}&\forall\, pvm_1, pvm_2\colon\ PVM \cdot \{\!\!\{toSnap(pvm_1), toSnap(pvm_2)\}\!\!\}\\ &\qquad domain_f(pvm_1) = domain_f(pvm_2) \wedge\\ &\qquad \forall r\colon Ref \cdot r \in domain_f(pvm_1) \Rightarrow apply_f(pvm_1, r) = apply_f(pvm_2, r)\\ &\qquad \Rightarrow pvm_1 = pvm_2\end{aligned}$$

Fig. 10. Background definitions related to partial value maps (see Sec. 4.2). $domain_f$ denotes the domain of a partial value map, $apply_f$ its application to a reference.

The trigger of the extensionality axiom $\{\!\!\{toSnap(pvm_1), toSnap(pvm_2)\}\!\!\}$ ensures that the extensionality axiom is instantiated whenever it is necessary to reason about the equality of partial value maps that are used as snapshots. Wrapping partial value maps by *toSnap* is necessary because Viper requires snapshots to uniformly be of type *Snap*; function *toSnap* embeds values into the *Snap* type (a corresponding inverse function exists as well). This external requirement (of Viper, not of our technique) turned out to be beneficial for us, since it allows choosing triggers that are permissive, yet yield good performance.

Inhaling and Exhaling Pure Assertions. Fig. 11 shows the symbolic execution rules for inhaling and exhaling potentially heap-dependent (but pure) assertions such as pure quantifiers. Both rules use `eval` to evaluate the assertion, the result is then added to the path conditions or asserted to hold in the current state, respectively.

```
inhale(h₀, π₀, e) ⤳
    var (π₁, e̲) := eval(h₀, π₀, e)
    return (h₀, π₁ ∪ {e̲})

exhale(h₀, π₀, e) ⤳
    var (π₁, e̲) := eval(h₀, π₀, e)
    check π₁ ⊨ e̲
    return (h₀, π₁)
```

Fig. 11. Symbolic execution rules for inhaling and exhaling pure assertions.

Symbolic Evaluation of Expressions. Fig. 12 shows selected symbolic execution rules for evaluating expressions. Evaluating an implication $e_1 \Rightarrow e_2$ starts by evaluating e_1, and temporarily assuming $\underline{e_1}$ while evaluating e_2 (see also the discussion of Fig. 2 in Sec. 3.1). From the path conditions obtained from evaluating e_1 (π_δ), all instances of VmDefEq are extracted (π_v). The final set of path conditions, with which the verification proceeds (π_3), includes the path conditions obtained from the evaluation of e_1, all instances of VmDefEq that were obtained from evaluating e_2 (this allows memoizing `summarise` because value map definitions are always in scope, that is, are not nested under implications), and—conditionally on $\underline{e_1}$—the remaining path conditions from evaluating e_2.

Viper's remaining symbolic execution rules for evaluating expressions did not need to be changed when we implemented our technique. For illustrative purposes, we show the rule for evaluating heap-*in*dependent functions (including arithmetic and other operators), and for evaluating short-circuiting conjunction.

```
eval(h_0, π_0, e_1 ⇒ e_2) ⇝
    var (π_1, e_1) := eval(h_0, π_0, e_1)
    var (π_2, e_2) := eval(h_0, π_1 ∪ {e_1}, e_2)
    var π_δ := π_2\(π_1 ∪ {e_1})
    var π_v := {b ∈ π_δ | b is instance of VmDefEq}
    var π_3 := π_1 ∪ π_v ∪ {e_1 ⇒ ⋀(π_δ\π_v)}
    return (π_3, e_1 ⇒ e_2)

eval(h_0, π_0, fun(e_1, ..., e_n)) ⇝                    /* fun is heap-independent */
    var (π_1, e_1) := eval(h_0, π_0, e_1)
    ...
    var (π_n, e_n) := eval(h_0, π_{n-1}, e_n)
    return (π_n, fun(e_1, ..., e_n))

eval(h_0, π_0, e_1 ∧ e_2) ⇝
    var (π_1, e_1) := eval(h_0, π_0, e_1)
    var (π_2, e_⇒) := eval(h_0, π_1, e_1 ⇒ e_2)
    return (π_2, e_1 ∧ e_⇒)
```

Fig. 12. Additional symbolic execution rules for evaluating expressions.

B Descriptions of Examples

- `arraylist` is an encoding of a list implemented on top of an array, with operations to append an element to the list, and to insert an element into the list such that the list, if it was sorted before, remains sorted afterwards.
- `array-quickselect` is an encoding of a (recursive) quickselect implementation over an array, with strong specifications such as "the array has been permuted", and "the n-th smallest element has been selected".
- `binary-search-array` is an encoding of an (iterative) binary search performed over a sorted array.
- `graph-copy` is the encoding of an algorithm that copies a graph. Its specifications make use of a custom axiomatisation of maps to record relations between original and copied nodes.
- `graph-marking` is the encoding of a graph marking algorithm, in the spirit of mark-and-sweep garbage collectors, with strong specifications such as "nodes reachable from marked nodes are marked themselves".
- `longest-common-prefix` is a challenge from the VerifyThis Verification Competition 2012: finding the longest common prefix of two arrays.
- `max-array-elimination` is a challenge from the COST Verification Competition 2011: finding the maximum in an array by elimination.
- `max-array-st&&ard` is an encoding of the straightforward way of finding the maximum in an array; it uses the same interface specifications and the same client as the previous example.
- `parallel-array-replace` is the running example from this paper: replace each occurrence of an element in an array segment by recursing over the two half-segments in parallel.

C Examples

C.1 Running Example: Parallel Array-Replace

Fig. 13 shows the encoding of our running example (`parallel-replace` from our test set) in Viper. Here, `loc(a,i)` is the injective function mapping an array `a` to the ghost objects modelling its array slots. So, a source-level expression `a[i]` is translated to `loc(a,i).val` (see also Sec. 3.2). Our code defines the pre- and postconditions of the `Replace` method as parameterised macros (occurrences of which are inlined, similar to C-style macros), for reuse when encoding the recursive parallel calls. Viper does not support parallel composition, but fork-join-style concurrency can be modelled by appropriate **`exhale`** (fork) and **`inhale`** (join) statements.

Fig. 14 shows the background definitions for the array encoding that is used in Fig. 13 (as well as in other array-related examples from our test suite). Axiom `all_diff` constrains function `loc` to be injective in both arguments by axiomatising `first` and `second` to be the inverse functions for the first and second parameter of `loc`, respectively.

Fig. 15 shows a client that uses `Replace`, and a heap-dependent boolean function `Contains` that yields true if an array contains a given value in the array prefix `[0..before)`. `Contains` is intentionally left abstract (i.e., it has no body) to demonstrate that the only way of reasoning about the function is via function framing, which indeed allows us to prove the final assertion.

C.2 Graph-Marking

Fig. 16 shows an encoding of a graph-marking algorithm (`graph-marking` from our test set) in Viper. In Viper, the double ampersand (`&&`) is overloaded: it denotes the separating conjunction ($*$) as well as the usual boolean conjunction ($\wedge$); in the conjunction of two impure assertions, it always denotes the separating conjunction.

The macro `INV` describes a graph in terms of accessibility predicates and closure properties over a given set of nodes (of the graph): the first three **`forall`**s are ISCs, denoting permissions to the fields of each node in the set of nodes. The remaining two **`forall`**s are pure quantifiers; they express that the set of nodes is closed under following the `left` and `right` fields. The two quantifiers have been annotated with triggers to improve performance, as is common for Viper encodings.

```
define pre1(a, l, r) 0 <= l && l < r && r <= len(a)
define pre2(a, l, r)  forall i: Int :: l <= i && i < r ==>
                                       acc(loc(a, i).val)
define post1(a, l, r) forall i: Int :: l <= i && i < r ==>
                                       acc(loc(a, i).val)
define post2(a, l, r) forall i: Int :: l <= i && i < r ==>
                      (old(loc(a, i).val == from)
                         ? loc(a, i).val == to
                         : loc(a, i).val == old(loc(a, i).val))

method Replace(a: Array, left: Int, right: Int, from: Int, to: Int)
  requires pre1(a, left, right)
  requires pre2(a, left, right)
  ensures  post1(a, left, right)
  ensures  post2(a, left, right)
{
  if (right - left <= 1) {
    if(loc(a, left).val == from) {
      loc(a, left).val := to
    }
  } else {
    var mid: Int := left + (right - left)  /  2

//fork-left
    exhale pre1(a, left, mid)
    exhale pre2(a, left, mid)

//fork-right
    exhale pre1(a, mid, right)
    exhale pre2(a, mid, right)

//join-left
    inhale post1(a, left, mid)
    inhale post2(a, left, mid)

//join-right
    inhale post1(a, mid, right)
    inhale post2(a, mid, right)
  }
}
```

Fig. 13. Our running example, encoded in Viper. Inlined macros are used to reuse the pre- and postcondition of `Replace` when encoding the parallel recursive calls.

```
field val: Int

domain Array {
  function loc(a: Array, i: Int): Ref
  function len(a: Array): Int
  function first(r: Ref): Array
  function second(r: Ref): Int

  axiom all_diff {
    forall a: Array, i: Int :: {loc(a, i)}
      first(loc(a, i)) == a && second(loc(a, i)) == i
  }

  axiom length_nonneg {
    forall a: Array :: len(a) >= 0
  }
}
```

Fig. 14. Background definitions for our array encoding. It declares a type `Array`, an injective function `loc` denoting the ghost object representing the array slot at a given index, and a function `len` that denotes the length of an array.

```
method Client(a: Array)
  requires 1 < len(a)
  requires forall i: Int ::
              0 <= i && i < len(a) ==> acc(loc(a, i).val)
  requires Contains(a, 5, 1)
{
  Replace(a, 1, len(a), 5, 7)
  assert Contains(a, 5, 1) // Requires function framing
}

function Contains(a: Array, v: Int, before: Int): Bool
  requires 0 <= before && before <= len(a)
  requires forall i: Int ::
              0 <= i && i < before ==> acc(loc(a, i).val)
```

Fig. 15. Client of the `Replace` method from Fig. 13. Function framing allows us to prove the assertion in method `Client`.

```
field left: Ref;  field right: Ref;  field marked: Bool

define INV(nodes)
     !(null in nodes)
  && (forall n: Ref :: n in nodes ==> acc(n.left))
  && (forall n: Ref :: n in nodes ==> acc(n.right))
  && (forall n: Ref :: n in nodes ==> acc(n.marked))
  && (forall n: Ref :: {n.left in nodes}{n in nodes, n.left}
        n in nodes && n.left  != null ==> n.left  in nodes)
  && (forall n: Ref :: {n.right in nodes}{n in nodes, n.right}
        n in nodes && n.right != null ==> n.right in nodes)

method trav_rec(nodes: Set[Ref], node: Ref)
  requires node in nodes && INV(nodes) && !node.marked
  ensures node in nodes && INV(nodes)
/* Marked nodes are not unmarked */
  ensures forall n: Ref :: {n in nodes, n.marked}
            n in nodes ==> (old(n.marked) ==> n.marked)
  ensures node.marked
/* The graph structure is not modified. */
  ensures forall n: Ref :: {n in nodes, n.left}
            n in nodes ==> (n.left == old(n.left))
  ensures forall n: Ref :: {n in nodes, n.right}
            n in nodes ==> (n.right == old(n.right))
/* Propagation of the marker */
  ensures forall n: Ref :: {n in nodes, n.marked}
                           {n in nodes, n.left.marked}
            n in nodes ==>
              (  old(!n.marked)
               && n.marked ==> (n.left == null || n.left.marked))
  ensures forall n: Ref :: {n in nodes, n.marked}
                           {n in nodes, n.right.marked}
            n in nodes ==>
              (   old(!n.marked)
               && n.marked ==> (n.right == null || n.right.marked))
{
  node.marked := true

  if (node.left != null && !node.left.marked) {
    trav_rec(nodes, node.left)
  }

  if (node.right != null && !node.right.marked) {
    trav_rec(nodes, node.right)
  }
}
```

Fig. 16. An encoding of a simple graph-marking algorithm in Viper.