Modular Verification of Heap Reachability Properties in Separation Logic

ARSHAVIR TER-GABRIELYAN, ETH Zurich, Switzerland
ALEXANDER J. SUMMERS, ETH Zurich, Switzerland
PETER MÜLLER, ETH Zurich, Switzerland

The correctness of many algorithms and data structures depends on reachability properties, that is, on the existence of chains of references between objects in the heap. Reasoning about reachability is difficult for two main reasons. First, any heap modification may affect an unbounded number of reference chains, which complicates modular verification, in particular, framing. Second, general graph reachability is not supported by first-order SMT solvers, which impedes automatic verification.

In this paper, we present a modular specification and verification technique for reachability properties in separation logic. For each method, we specify reachability only locally within the fragment of the heap on which the method operates. We identify relative convexity, a novel relation between the heap fragments of a client and a callee, which enables (first-order) reachability framing, that is, extending reachability properties from the heap fragment of a callee to the larger fragment of its client, enabling precise procedure-modular reasoning. Our technique supports practically important heap structures, namely acyclic graphs with a bounded outdegree as well as (potentially cyclic) graphs with at most one path (modulo cycles) between each pair of nodes. The integration into separation logic allows us to reason about reachability and other properties in a uniform way, to verify concurrent programs, and to automate our technique via existing separation logic verifiers. We demonstrate that our verification technique is amenable to SMT-based verification by encoding a number of benchmark examples into the Viper verification infrastructure.

CCS Concepts: • **Software and its engineering → Formal software verification**.

Additional Key Words and Phrases: graphs, heap reachability, separation logic, SMT, specification

ACM Reference Format:
Arshavir Ter-Gabrielyan, Alexander J. Summers, and Peter Müller. 2019. Modular Verification of Heap Reachability Properties in Separation Logic. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 121 (October 2019), 28 pages. https://doi.org/10.1145/3360547

1 INTRODUCTION

Separation logic [Reynolds 2002] has greatly simplified the verification of basic heap data structures such as lists and trees by leveraging the disjointness of sub-heaps to reason about the effects of heap modifications. However, verifying data structures that permit unbounded sharing remains challenging. Their correctness often depends on heap reachability properties, that is, the existence of paths of references between objects. For instance, the path compression of union-find needs to preserve the reachability of the root object, the termination of heap traversals might rely on the

Authors' addresses: Arshavir Ter-Gabrielyan, Department of Computer Science, ETH Zurich, Switzerland, ter-gabrielyan@inf.ethz.ch; Alexander J. Summers, Department of Computer Science, ETH Zurich, Switzerland, alexander.summers@inf.ethz.ch; Peter Müller, Department of Computer Science, ETH Zurich, Switzerland, peter.mueller@inf.ethz.ch.

2475-1421/2019/10-ART121
https://doi.org/10.1145/3360547

absence of cyclic paths, and the invariant of a garbage collector may prescribe that each object is reachable from the list of allocated objects or the free-list, but not from both.

Reasoning about reachability properties is difficult for two main reasons. (1) Modularity: reachability is inherently a non-local property. Any heap modification may affect an unbounded number of heap paths, which complicates framing, that is, proving modularly that a heap update or method call does not affect a given reachability property. (2) Automation: general graph reachability is not supported by SMT solvers, which power most automatic verification tools.

Existing work addresses these challenges typically by supporting only certain kinds of reachability properties or certain classes of data structures. For instance, Itzhaky et al. [2014] present a modular verification technique for reachability properties of a broad class of linked-list programs, but do not support structures that can have more than one outgoing reference per object. Such structures may contain an arbitrary number of alternative paths between two objects, and maintaining reachability information via first-order formulas becomes infeasible. Flows [Krishna et al. 2018] is a technique providing local reasoning for updates to subgraphs which *preserve* properties such as reachability, e.g., changes to a subgraph which neither add nor remove paths between nodes in its boundary. However, flows do not provide analogous means for local reasoning about methods which are *intended* to change these paths (e.g., a function which connects two subgraphs).

This paper presents a modular verification technique for general heap reachability properties that supports both acyclic data structures with a bounded number of outgoing references per object (for instance, DAG structures such as BDDs [Akers Jr. 1978]) and (potentially cyclic) 0–1-path graphs, that is, graphs that contain at most one path (modulo cycles) between each pair of objects (such as a ring buffer). Our technique is integrated into separation logic, which allows us to reason about reachability and other properties in a uniform way, to verify concurrent programs, and to automate our technique via existing separation logic verifiers. Our technique enables modular reasoning by specifying reachability properties locally within the memory footprint of a method rather than in the entire heap. A novel form of reachability framing allows one to extend the reachability properties guaranteed by a callee method to the (larger) footprint of its client. As a result, each method can be verified modularly, without considering the heap outside its footprint, the implementations of other methods, or other threads.

Contributions. Our paper makes the following technical contributions:

- *Specification:* We introduce a specification technique for reachability properties in the context of separation logic. It enables modular verification, even for concurrent programs (Sec. 2).
- *Verification:* We present a novel verification technique for reachability properties. In particular, we identify relative convexity of method footprints as a property that enables precise reachability framing and procedure-modular reasoning. Our technique goes beyond prior work [Itzhaky et al. 2014] by supporting all acyclic graphs (Sec. 3).
- *Cyclic graphs:* We extend our verification technique to cyclic 0–1-path graphs. While reachability framing carries over from the acyclic case, cyclic graphs require a more elaborate machinery to handle reference field updates (Sec. 4).
- *Automation:* We demonstrate that our verification technique is amenable to SMT-based verification by encoding a number of benchmark examples into the Viper verification infrastructure [Müller et al. 2016b] (Sec. 5).

2 SPECIFICATION TECHNIQUE

In this section, we illustrate our technique using a DAG data structure with node type ***Node*** and fields `left` and `right`. Method `merge` in Fig. 1 takes as arguments references `l` and `r` to two nodes of disjoint DAGs and attaches `r` as descendant of `l`. It returns `link`, a node of the first DAG, to which `r`

```
method merge(l: Node, r: Node,
             𝔤: Graph, ldag: Graph, rdag: Graph)  // ghost parameters
  returns link: Node                              // updated node
  requires 𝔤 = ldag ⊎ rdag ∧ l ∈ ldag ∧ r ∈ rdag
           ∀x, y ∈ 𝔤 • ¬E(𝔤, x, y) ∨ ¬P(𝔤, y, x)    // acyclic invariant
           ∀n • n ∈ ldag ⇔ P(𝔤, l, n)
           ∀n • n ∈ rdag ⇔ P(𝔤, r, n)
  ensures link ∈ ldag
          ∀x, y ∈ 𝔤 • ¬E(𝔤, x, y) ∨ ¬P(𝔤, y, x)    // acyclic invariant
          ∀x, y • E(𝔤, x, y) ⟺ E₀(𝔤, x, y) ∨ x = link ∧ y = r
          ∀x, y • P(𝔤, x, y) ⟺ P₀(𝔤, x, y) ∨ P₀(𝔤, x, link) ∧ P₀(𝔤, r, y)
{
    if (l.right != null) {
        var nldag := sub(𝔤, ldag, l.right)        // define new ghost parameter
        link := merge(l.right, r, nldag ⊎ rdag, nldag, rdag)
    } else {
        l.right := r
        link := l
    }
}
```

Fig. 1. An example program and specification. Method `merge` attaches the DAG rooted in `r` to a node of the DAG rooted in `l`, and returns that node. We use the edge predicate `E` and the path predicate `P` to specify reachability properties, within a set of objects `𝔤`. Each specification line is a separate conjunct. The footprint `𝔤` is closed due to the equivalences in the last two preconditions.

was attached. The postcondition ensures that exactly one connection was created (via an edge from `link` to the root of the second DAG, `r`), and that heap paths exist in the post-state either if they existed in the pre-state or were connected by the new edge, (`link`, `r`). We explain the specification of `merge` in full detail in the remainder of this section.

2.1 Footprints

Separation logics associate an *access permission* with each memory location. Access permissions are held by method executions and may be transferred between methods upon calls and returns; they can be thought of as additional program state used for reasoning (*ghost* state). A heap location can be accessed only while the corresponding permission is held. The set of locations that a method may access is called its *footprint*. Due to (de)allocation or concurrency, the footprint of a method may change during its execution. A method's precondition specifies which permissions to transfer on calling the method. The initial footprint of a method contains exactly the locations for which its precondition requires permission. Conversely, the method postcondition specifies which permissions to return to the client when the method terminates.

The footprint of any method operating on linked heap structures, e.g., lists and DAGs, contains a statically unknown number of memory locations. To provide a convenient way to refer to a method's footprint, we equip each method with a distinct ghost parameter `𝔤: Graph` to denote its footprint. For simplicity, instead of specifying the footprint as a set of object-field pairs, we let ***Graph*** denote sets of non-null objects and keep the fields implicit when they are clear from the context.

The set stored in $\mathfrak{g}$ is updated whenever the footprint changes, for instance, due to allocation. In order to be able to refer to the final footprint of a method execution in its postcondition, we make $\mathfrak{g}$ an in-out parameter. For simplicity, we assume in the following that the footprint of a method remains unchanged, s.t. the value of $\mathfrak{g}$ is constant; an extension is straightforward.

We equip each method with implicit pre- and postconditions to require and ensure permissions to all locations in the footprint:

```
requires ∀n ∈ g • acc(n.left) ∗ acc(n.right)
ensures  ∀n ∈ g • acc(n.left) ∗ acc(n.right)
```

Here, $\mathtt{acc}(x.f)$ denotes an access permission to the *memory location* for field f of object x (like $x.f \mapsto \_$ in traditional separation logic [Reynolds 2002]), $*$ denotes separating conjunction, and the universal quantifier is an iterated separating conjunction (ISC) [Müller et al. 2016a; Reynolds 2002], which (here) denotes permissions to all field locations of objects in the footprint $\mathfrak{g}$. In contrast to using recursive definitions to specify unbounded heap structures (e.g., separation logic predicates [Parkinson and Bierman 2005; Yang 2001b]), ISC permits arbitrary sharing within the set $\mathfrak{g}$ (many field *values* may alias the same node) and does not prescribe a traversal order within the data structure. We assume for simplicity that a method specification expresses *all* required and returned permissions via these implicit contracts with respect to $\mathfrak{g}$, but it is easy to also support other permission specifications, e.g., points-to predicates and recursive predicates.

In our example, we use two additional ghost parameters `ldag` and `rdag` to allow our specification to simply denote the sets of objects constituting the first and second DAG, respectively. The first precondition expresses that the method footprint is the disjoint union of these two DAGs.

2.2 Local Reachability

Reasoning in a separation logic has the key advantage that one can modularly verify properties of a method, and reuse this verification for all calling contexts (and concurrently-running threads). Enforcing that properties verified for the method depend *only* on its footprint, guarantees that they hold independently of the context; we refer to these as the *local* properties of the footprint. However, classical reachability in the heap is *not* a local property of this form. Hence, combining reachability and separation logic requires us to refine the notion of reachability to one that is local, as we explain next.

Our technique provides two predicates to express reachability properties in specifications. We generalize classical reachability by adding an extra footprint parameter, $\mathfrak{g}$ to make the property local. The *edge predicate* $\mathsf{E}^F(\mathfrak{g}, x, y)$ expresses that object x is in the set $\mathfrak{g}$ and has a field from the set of fields F storing a non-null object y (which need not be in $\mathfrak{g}$). The *path predicate* P denotes, for a fixed $\mathfrak{g}$ and F, the reflexive, transitive closure of E, that is, $\mathsf{P}^F(\mathfrak{g}, x, y)$ expresses that either $x = y$, or there is a path of field references from x to y s.t. all objects on the path (except possibly y) are in $\mathfrak{g}$ and all fields are in F; in particular P may denote reachability via multiple fields. We omit the parameter F when the set of fields is clear from the context; for instance, in our example, F consists of the (only) reference-typed fields `left` and `right`. We say that a path $x \ldots y$ is $\mathfrak{g}$-*local* if $\mathsf{P}(\mathfrak{g}, x, y)$ holds. Both our edge and path predicates are defined over a *mathematical abstraction* of the current heap graph (cf. Sec. 3.1), and are *pure* in the separation logic sense, allowing us to freely repeat them in specifications.

Our edge and path predicates enable rich reachability specifications within a method's footprint. The preconditions of `merge` express that the method footprint is acyclic and closed under the edge relation (due to the second and the last two preconditions), and that `ldag` and `rdag` contain exactly the objects reachable from `l` and `r`, resp. In general, method specifications are checked to only employ edge and path predicates whose first parameter is the method's footprint or a subset thereof.

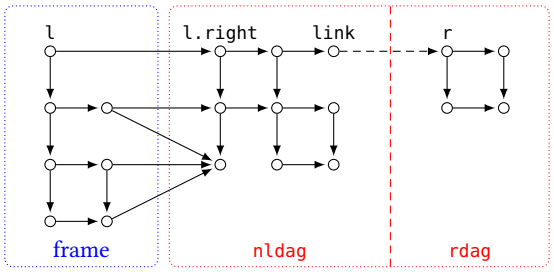

Fig. 2. An example scenario of running `merge` on two DAGs rooted in `l` and `r`. Small circles correspond to heap objects; solid arrows represent fields initialized in the pre-state that are unchanged; the dashed arrow represents the new heap edge (created in the post-state by initializing a field). The frame of the recursive call is surrounded with blue; the footprint is surrounded with red.

Method postconditions typically express how reachability *changes* within this footprint. In our example, the first postcondition specifies that the result `link` is part of the first DAG and its `right`-field was initially null. The `old`-expression allows postconditions to refer to pre-state values; we write $\mathsf{E}_0(\ldots)$ to abbreviate `old(`$\mathsf{E}(\ldots)$`)`, and analogously for P. We can freely mix reachability specifications with specifications in terms of the program heap (e.g., the `link.right` expression). The other postconditions illustrate how we can specify the new edge and path relations in terms of their originals, summarizing the method's effect. In particular, the last postcondition expresses that an object x reaches an object y in the post-state *iff* it reached y already in the pre-state, or if x reaches `link` in the first DAG and y is in the second DAG. Our method specification leaves `link` underspecified, whereas the implementation chooses the rightmost node in the first DAG. We could easily provide a less abstract specification by using path predicates over (only) the `right`-field.

The recursive call in method `merge` needs to supply values for the three ghost parameters. We construct these values using a predefined function `sub(`$\mathfrak{g}$`:`***Graph***`,`$\mathfrak{h}$`:`***Graph***`,root:`***Node***`)`, which yields the subset of $\mathfrak{h}$ reachable from the node `root` via $\mathfrak{g}$-local paths. The properties known for the resulting set are summarized by the following *heap-dependent function* [Müller et al. 2016b] declaration:[1]

```
function sub(g: Graph, h: Graph, root: Node): Graph
  requires root ∈ h ∧ h ⊆ g
  ensures result ⊆ h ∧ root ∈ result ∧ CLOSED_h(result) ∧ ∀n • n ∈ result ⇔ P(g, root, n)
```

where **result** refers to the result value of the function; $\mathsf{CLOSED}_{\mathfrak{h}}(\mathfrak{r})$ denotes that an edge that exits $\mathfrak{r}$ must not end in $\mathfrak{h}$:

$$\mathsf{CLOSED}_{\mathfrak{h}}(\mathfrak{r}) \;:\Longleftrightarrow\; \forall x \in \mathfrak{r}, y \bullet \mathsf{E}(\mathfrak{r}, x, y) \implies y \notin \mathfrak{h} \backslash \mathfrak{r} \tag{1}$$

Note that $\mathsf{CLOSED}_{\mathfrak{h}}(\mathfrak{r})$ is permissive enough to allow selecting new footprints for method calls even if the current footprint is *open*, i.e., if there exist edges that exits the current footprint. To specify that a subheap is closed *in the global heap*, we would use a stronger condition:

$$\mathsf{CLOSED}(\mathfrak{g}) \;:\Longleftrightarrow\; \forall x \in \mathfrak{g}, y \notin \mathfrak{g} \bullet \neg \mathsf{E}(\mathfrak{g}, x, y) \tag{2}$$

2.3 Verification Challenges

The specification ingredients presented above allow us to combine separation logic specification with reachability. However, practical verification of these specifications requires the solution of three challenges. First, we must handle direct updates to the program heap, and model their effects on

[1] Unlike methods, functions in our language are guaranteed to be side-effect free. Hence, we do not distinguish between P and P_0 in the postcondition of `sub` (similar for E and E_0).

our E and P predicates. SMT solvers cannot efficiently automate reasoning about a direct definition of P as transitive closure, but it has been shown that a first-order approximation technique can be efficiently used for this purpose [Dong and Su 1995; Lev-Ami et al. 2009]. Second, and most challenging, we require a technique to deduce reachability for a method call's *client* from what is known about its callee's footprint: a problem we call *reachability framing*. This is necessary, for example, when reasoning about the recursive call to `merge` in our example of Fig. 1; we must relate local reachability information in the client's footprint to that of its callee. Finally, we require a modeling of our verification technique in an automated tool; we aim for proof obligations ultimately amenable to first-order SMT solvers, which necessitates effective quantifier instantiation strategies.

3 REACHABILITY IN ACYCLIC STRUCTURES

In this section, we explain the core ingredients of our verification technique for combining reachability information with separation logic style reasoning. Reasoning about a method starts with assuming its precondition. The precondition provides permissions to access the objects (i.e., nodes) in its footprint and the reachability constraints that guarantee the existence or the absence of heap paths connecting some objects from the footprint. As the program performs modifications to some parts of the heap, our goal is to determine a precise way of checking any (local) reachability query (e.g., in the method's postcondition) after these changes. Hence it is important to identify the paths that were unchanged and those that were created or destroyed by each operation.

Heap modifications are performed either directly by field updates or indirectly through method calls. In the former case, the reachability properties known to hold before the update need to be adjusted to reflect the change of heap references (Sec. 3.2). For a field update, the local reachability properties before and after the update can be expressed within the same (enclosing method's) footprint. The situation is more complex for method calls (Sec. 3.3). To determine the reachability properties after a call (the reachability framing problem), one needs to combine reachability properties before the call that are known to be outside of the call's footprint (hence, unaffected by the call) with reachability properties guaranteed by the callee method (as expressed in the callee's postcondition). These two sets of properties are expressed within the footprints of the client and the callee, respectively. If these footprints are not equal, then the reachability properties guaranteed by the callee need to be re-interpreted in the client's footprint.

We present our techniques for tackling these challenges in the remainder of the paper. We discuss how our reachability reasoning technique is integrated with separation logic in Sec. 3.1. The technique for direct field updates discussed in Sec. 3.2 requires the current method's footprint to be acyclic; note that we generally permit arbitrary structures, including those with heap cycles, *outside* of the footprint. However, our technique for method calls, and all of the formulas that we present in Sec. 3.3, *do not* require acyclicity. Instead, we require and exploit *relative convexity* of method footprints, a novel restriction that is strong enough to reduce the reachability framing problem to first-order formulas tractable for SMT solvers, but permissive enough to embrace a broad spectrum of challenging data structures. Sec. 4 explains how our technique can be extended to potentially *cyclic* 0–1-path graphs.

3.1 Encoding of Edge and Path Predicates

Our specification technique supports reachability via the edge predicate E and the path predicate P. In order to verify such specifications, we encode them into a flavor of separation logic and use an existing verification tool to construct proofs in that logic. We use Implicit Dynamic Frames [Smans et al. 2012] for this purpose, a variation of separation logic [Parkinson and Summers 2011] that separates specifications of access permissions for memory locations from specifications of the values stored in these locations. For instance, separation logic's points-to predicate $x.f \mapsto v$ is

specified in implicit dynamic frames as a conjunction of the access permission and the field content: $\texttt{acc}(x.f) \ * \ x.f = v$. This separation of permissions and value properties allows us to conveniently express additional value properties, e.g., sortedness, in addition to reachability properties, without having to define a new graph-abstraction that exposes the values of interest.

Our edge predicates could be defined directly, e.g., as $(x.f_1 = v \vee x.f_2 = v)$ for two fields f_1 and f_2; conceptually, `E` is a first-order abstraction over this property, which may, in particular, be used in the *syntactic triggering patterns* [Barrett et al. 2017; de Moura and Bjørner 2008; Detlefs et al. 2005; Moskal 2009] that the SMT solver requires to control quantifier instantiations (and which cannot include logical operations such as $\vee$ above).

Unlike the edge predicate `E`, directly defining the path predicate `P` would compromise automation. A definition would involve transitive closure, which is notoriously difficult to handle for SMT solvers. Therefore, we take a different approach here. We leave the path predicate undefined and axiomatize its essential properties, for instance, how it is affected by heap updates. We specify these axioms over mathematical graphs and not directly over the heap-dependent edge and path predicates. Therefore, our encoding first abstracts the heap within a footprint to a set of edges (ordered pairs of nodes) and then expresses reachability over those. This abstraction is defined by a predefined function called `snapshot`. For simplicity, we define `snapshot` using the notation of our source language, but it is only used internally by our encoding. In particular, the function implicitly depends on the heap and requires permissions to all objects in its footprint $\mathfrak{g}$:

function $\texttt{snapshot}^F(\mathfrak{g}\texttt{:}\ \textbf{\textit{Graph}})\texttt{:}\ \textbf{\textit{Edgeset}}$

ensures $\forall x, y \bullet x \in \mathfrak{g} \wedge y \neq \texttt{null} \wedge (x.f_1 = y \vee \ldots \vee x.f_n = y) \iff (x, y) \in \textbf{result}$

Here, ***Edgeset*** is the type of sets of pairs of nodes, and $F = \{f_1, \ldots, f_n\}$; we omit this parameter when it is clear from the context. The postcondition can be thought of as an axiom over an uninterpreted function that defines its semantics. Note that `snapshot` also collapses edges between two objects for different field names (duplicate edges are not needed to keep track of reachability).

This abstraction function lets us define the edge predicate in a straightforward way:

$$\texttt{E}^F(\mathfrak{g}, x, y) \ :\iff \ (x, y) \in \texttt{snapshot}^F(\mathfrak{g}) \tag{3}$$

To avoid the issues with transitive closure mentioned above, we do not define the path predicate directly, but axiomatize the properties we need for verification. In fact, we define the path relation in terms of a function $\hat{P}$ over graphs and then axiomatize the latter, state-independent function:

$$\texttt{P}^F(\mathfrak{g}, x, y) \ :\iff \ \hat{P}(\texttt{snapshot}^F(\mathfrak{g}), x, y) \tag{4}$$

For this axiomatization, we carefully control the quantifier instantiation performed by SMT solvers to avoid diverging proof search. For instance, we include the axiom below, but let the solver instantiate it only to a fixed depth of unrolling $\hat{P}$ [Leino and Monahan 2009].

$$\hat{P}(G, x, y) \ :\iff \ x = y \vee \exists z \bullet (x, z) \in G \wedge \hat{P}(G, z, y) \tag{5}$$

3.2 Field Updates

A field update `x.f := v` may affect reachability properties in the heap and, thus, both edge and path predicates. Since our encoding contains a precise definition of the edge predicate in terms of the underlying heap (via (3) and the definition of `snapshot`), the verifier can determine which edge predicates hold after a field update.

However, determining the effect of a single field update on the *path* relation is more intricate as its the axiomatization is not sufficient to determine which predicates hold after a field assignment (e.g., because this reasoning step would require induction proofs, which SMT solvers cannot find automatically). We solve this problem by adapting an existing approach: for *acyclic* graphs (which

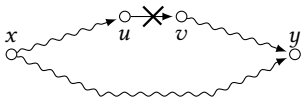

Fig. 3. The reachability update problem in presence of alternative paths. Concrete heap edges are represented by straight arrows, while (possibly, zero-length) heap paths are represented by wavy arrows. The upper path $x \ldots y$ *depends* on the edge (u, v); removing this edge would destroy the path, but x may still reach y after deleting (u, v), here, via the lower path. Alternative paths may occur in our setting because we permit multiple reference fields per object.

we focus on in this section), one can provide first-order *update formulas* that express precisely how adding or deleting a single edge affects reachability [Dong and Su 1995; Lev-Ami et al. 2009]. For example, the following update formula characterizes the effect of adding an edge between nodes a and b (e.g., by initializing a field of a):

$$\forall x, y \bullet \mathsf{P}^F(\mathfrak{g}, x, y) \iff \mathsf{P}_0^F(\mathfrak{g}, x, y) \vee \mathsf{P}_0^F(\mathfrak{g}, x, a) \wedge \mathsf{P}_0^F(\mathfrak{g}, b, y) \tag{6}$$

where P and P_0 denote the path predicate in the states before and after the update.

The update formula for removing an edge is more complex. Since we allow for an arbitrary out-degree of nodes (via multiple reference fields), it is possible for there to exist multiple paths between two different nodes (Fig. 3). When adding an edge between two nodes, the new P relation can be updated relatively simply, e.g., via (6); no paths have been lost, and only paths connected by this new edge are created. On *removal* of an edge, no paths are created, but, for node pairs previously connected by a path using this edge, it is unclear whether or not they belong to the new P relation, due to the possibility of *alternative paths*. This entails a more-complex update formula for the edge-removal case (due to Dong and Su [1995]); see [Ter-Gabrielyan et al. 2019a, App. A] for details. A general field update entails removing and then adding an edge, as we demonstrate for our `merge` example in Fig. 7.

Our verification technique rewrites each field update $\mathtt{x}.f := \mathtt{v}$ with a method call to an internal `update` method with the same footprint $\mathfrak{g}$ as for the current method. The postconditions of `update` make the reachability update formulas available to the SMT solver. This way, we assume the update formulas for each field set F that is used in the current method specification and that contains the updated field f (reachability for other field sets is not affected by the update).

The else-branch in the example from Fig. 1 modifies the heap through a single field update. The second postcondition describes the effect on the edge relation; it follows directly from the definition of the edge predicate. The third postcondition, about the path relation, is exactly the update formula (6), with `link` and `r` for a and b, resp.

3.3 Method Calls and Relatively Convex Footprints

Update formulas allow us to precisely capture the effect of adding or removing *individual* edges, which is sufficient to reason about field updates. However, reasoning modularly about method calls requires us to determine the effect of *multiple* heap updates. According to the callee's specification, we can partition the footprint $\mathfrak{g}$ of the client into the footprint $\mathfrak{h}$ of the callee and the remainder $\mathfrak{f}$ ($\mathfrak{g} = \mathfrak{f} \uplus \mathfrak{h}$). This remainder is the *frame* of the call and cannot be modified by the callee method. The postcondition of a callee method provides a specification of reachability information *within* its footprint $\mathfrak{h}$. The challenge is to determine the effect of the call on reachability within the (generally larger) footprint $\mathfrak{g}$ of its client. For the edge relation, this extrapolation is straightforward:

$$\forall x \in \mathfrak{f} \uplus \mathfrak{h}, y \bullet \mathsf{E}(\mathfrak{f} \uplus \mathfrak{h}, x, y) \iff \mathsf{E}(\mathfrak{f}, x, y) \vee \mathsf{E}(\mathfrak{h}, x, y) \tag{7}$$

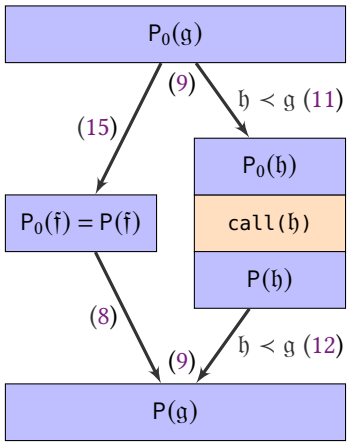

Fig. 4. The flow of reachability information in the presence of method calls. Reachability information in the client's footprint $\mathfrak{g} = \mathfrak{f} \uplus \mathfrak{h}$ is split into reachability information within $\mathfrak{h}$ and $\mathfrak{f}$ before the call, the effect of the call on $\mathfrak{h}$ is accounted for, and then the information is recombined to paths in $\mathfrak{g}$. The numbers in parentheses indicate ingredients of our technique explained in this section.

In separation logic, a method may modify any heap edges that *originate* in its footprint; hence, the predicate $\mathsf{E}(\mathfrak{h}, x, y)$ implies that x is in the footprint $\mathfrak{h}$ and $\mathsf{E}(\mathfrak{f}, x, y)$ implies that x is in the frame $\mathfrak{f}$. We refer to edges that cross the boundary of the footprint as *cut points*: if $x \in \mathfrak{h}, y \notin \mathfrak{h}$, then (x, y) is an *exit point* of the footprint, and if $x \notin \mathfrak{h}, y \in \mathfrak{h}$, then (x, y) is an *entry point* into the footprint.

Unfortunately, a simple rule such as (7) does not exist for relating *paths* in $\mathfrak{f} \uplus \mathfrak{h}$ to those in $\mathfrak{f}$ and $\mathfrak{h}$. A path can span fields from both heap partitions, and, in general, could cross the boundary between the two unboundedly many times. It is known that, in full generality, a first-order reachability framing formula for our path predicate cannot exist (see e.g., [Itzhaky et al. 2014]). The key insight behind our technique for handling method calls is that this intractable situation becomes tractable if the footprint of the callee is *relatively convex* in the composed heap.

Definition 1 (Relatively Convex Footprints). *In a given program state and for a given set of reference fields F, footprint $\mathfrak{h}$ defines a* relatively convex sub-footprint *of footprint $\mathfrak{g}$ (written $\mathfrak{h} \prec \mathfrak{g}$) iff $\mathfrak{g} = \mathfrak{f} \uplus \mathfrak{h}$ for some footprint $\mathfrak{f}$, and no paths within $\mathfrak{g}$ leave $\mathfrak{h}$ and then return:*

$$\forall x, y \in \mathfrak{h}, u \in \mathfrak{f} \bullet \neg \mathsf{P}^F(\mathfrak{g}, x, u) \vee \neg \mathsf{P}^F(\mathfrak{g}, u, y)$$

We show, in the remainder of this section, how we exploit this property to enable precise, first-order, and modular reasoning about reachability in presence of method calls. In particular, we are able to make tractable the problem of framing reachability information when a method footprint $\mathfrak{h}$ is relatively convex in its client's footprint $\mathfrak{g}$. This requirement is checked by our technique at the call site, but is typically naturally the case. For example, any method operating on a recursively-defined data type, its sub-structures or portions thereof (such as linked list segments), a strongly connected component of a potentially-cyclic structure, or combinations of these will have a relatively convex footprint. DAG traversals also have relatively convex footprints. For instance, the recursive call to `merge` in our running example of Fig. 1 and the corresponding illustration in Fig. 2 demonstrate a method call with a relatively convex footprint. Note that both acyclicity and relative convexity are defined relatively to a field set F. Therefore, even operations on data structures with back-pointers (such as parent-pointers in a tree) typically have relatively convex footprints as long as the path predicates are defined in terms of only forward-references or only back-pointers.

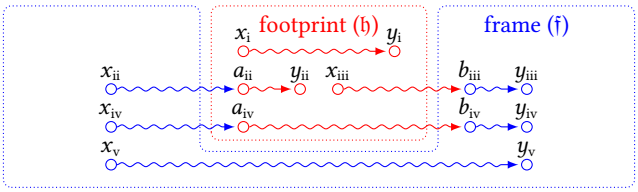

Fig. 5. No paths originating and ending inside a relatively convex footprint may go though nodes of its frame. Therefore, the paths that originate in the frame may enter and exit the footprint at most once. This gives five possibilities for a path to interact with a relatively convex footprint.

Method call overview. A high-level overview of our solution is illustrated in Fig. 4. We use P_0 to represent the paths before the call, and P for those afterwards. According to standard separation logic reasoning, method calls are only allowed if the callee's footprint $\mathfrak{h}$ is a subset of the client's ($\mathfrak{g} = \mathfrak{f} \uplus \mathfrak{h}$, for some frame $\mathfrak{f}$). Under the additional requirement that $\mathfrak{h} \prec \mathfrak{g}$, the technique we present in this section shows how to decompose reachability information before the call (i.e., expressed in terms of $\mathsf{P}_0(\mathfrak{g}, \ldots)$) into paths in the callee's footprint ($\mathsf{P}_0(\mathfrak{h}, \ldots)$) and paths in the frame ($\mathsf{P}_0(\mathfrak{f}, \ldots)$). The callee's specification is responsible for relating $\mathsf{P}(\mathfrak{h}, \ldots)$ information to $\mathsf{P}_0(\mathfrak{h}, \ldots)$ information, i.e., specifying how reachability changes *within* the callee's footprint. Conversely, reachability purely in the *frame* $\mathfrak{f}$ cannot be changed by a call, since it does not have the permissions to do so. Indeed, based on consideration of the permissions *not passed* to the method call, we know that the following formula holds (which we call *separation-logic framing*):

$$\forall x \in \mathfrak{f}, y \bullet \mathsf{P}(\mathfrak{f}, x, y) \iff \mathsf{P}_0(\mathfrak{f}, x, y) \tag{8}$$

Our technique then provides means of reconstructing reachability in the client's footprint ($\mathsf{P}(\mathfrak{g}, \ldots)$) from the information we have after the call in terms of $\mathsf{P}(\mathfrak{h}, \ldots)$ and $\mathsf{P}(\mathfrak{f}, \ldots)$.

Path partitioning. The first key step of our solution is *path partitioning*. We exploit relative convexity of the callee's footprint to define formulas for soundly and precisely relating reachability in a client's footprint to reachability in the callee and its frame, and vice versa. Fig. 5 illustrates the possibilities for a path in the client's footprint $\mathfrak{g}$ to interact with a relatively convex footprint $\mathfrak{h}$. We proceed by analyzing Fig. 5 by cases, deriving formulas one of which must hold in each possible case.

Crucially, our relative convexity assumption $\mathfrak{h} \prec \mathfrak{g}$ guarantees that no paths $x \ldots y$ in $\mathfrak{g} = \mathfrak{f} \uplus \mathfrak{h}$ enter or leave $\mathfrak{h}$ more than once. We summarize the five cases for paths from x to y based on the distribution of these nodes between the footprint, $\mathfrak{h}$, and the frame of the call, $\mathfrak{f}$: (i) $x, y \in \mathfrak{h}$ as is the whole path, (ii) $x \in \mathfrak{f}, y \in \mathfrak{h}$; the path crosses the boundary *once*, (iii) $x \in \mathfrak{h}, y \in \mathfrak{f}$ again crossing *once*, (iv) $x, y \in \mathfrak{f}$ with a path entering and leaving $\mathfrak{h}$ *once*, and (v) $x, y \in \mathfrak{f}$ with a path entirely in $\mathfrak{f}$. Note that these cases are exhaustive for a path between $x, y \in \mathfrak{g}$, due to our convexity restriction.

These five cases translate to the following formulas, allowing us to relate reachability in $\mathfrak{f} \uplus \mathfrak{h}$ and reachability in the two subheaps $\mathfrak{f}$ and $\mathfrak{h}$ individually, which we call *path partitioning formulas*:

$$\begin{aligned}
&\text{(i)} && \forall x \in \mathfrak{h}, y \in \mathfrak{h} \bullet \mathsf{P}(\mathfrak{f} \uplus \mathfrak{h}, x, y) \iff \mathsf{P}(\mathfrak{h}, x, y) \\
&\text{(ii)} && \forall x \in \mathfrak{f}, y \in \mathfrak{h} \bullet \mathsf{P}(\mathfrak{f} \uplus \mathfrak{h}, x, y) \iff \exists a \in \mathfrak{h} \bullet \mathsf{P}(\mathfrak{f}, x, a) \wedge \mathsf{P}(\mathfrak{h}, a, y) \\
&\text{(iii)} && \forall x \in \mathfrak{h}, y \in \mathfrak{f} \bullet \mathsf{P}(\mathfrak{f} \uplus \mathfrak{h}, x, y) \iff \exists b \in \mathfrak{f} \bullet \mathsf{P}(\mathfrak{h}, x, b) \wedge \mathsf{P}(\mathfrak{f}, b, y) \\
&\text{(iv)–(v)} && \forall x \in \mathfrak{f}, y \in \mathfrak{f} \bullet \mathsf{P}(\mathfrak{f} \uplus \mathfrak{h}, x, y) \iff \mathsf{P}(\mathfrak{f}, x, y) \vee \\
& && \qquad \exists a \in \mathfrak{h}, b \in \mathfrak{f} \bullet \mathsf{P}(\mathfrak{f}, x, a) \wedge \mathsf{P}(\mathfrak{h}, a, b) \wedge \mathsf{P}(\mathfrak{f}, b, y)
\end{aligned} \tag{9}$$

These formulas can be used left-to-right or right-to-left. In the former case, we obtain a canonical means of *decomposing* information about paths in a composed footprint of the client into information

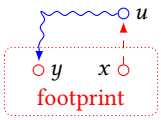

Fig. 6. The footprint (surrounded with red) is relatively convex *before* a method call, satisfying (11). The heap edge created by the call is represented by a dashed arrow. The new edge (x, u) is an *exit point* of the footprint into the frame; since its end node u reaches the footprint via some path $u \ldots y$, adding (x, u) violates the relative convexity property of the footprint (12).

about paths in the callee's footprint and paths in the frame. In the latter case, we obtain means of *reassembling* reachability information in the composed footprint from that in the constituent parts. In practice, we add separate **assume** statements for both directions of each formula, so that we can clearly specify to the underlying SMT solver when to instantiate the formula in which direction.

It is due to our relative convexity assumption that there exist simple first-order path partitioning formulas (9). Without this assumption, the number of cut points of $\mathfrak{h}$ could be *arbitrary*, and localization of reachability information would require either considering an unbounded set of cases or higher-order reasoning, preventing automatic verification.

For simplicity, formulas (9) cover only the cases $x, y \in \mathfrak{g}$: however, paths that are local to a particular footprint may leave that footprint by a single edge (and so case (i) above, for example, does not provide information about such paths in $\mathfrak{h}$). The following formulas reduce the case $y \notin \mathfrak{g}$ to the cases already covered by introducing a node $u \in \mathfrak{g}$ with an edge to y:

$$\begin{aligned} \forall x \in \mathfrak{h}, y \notin \mathfrak{h} \ \bullet \ \mathsf{P}(\mathfrak{h}, x, y) &\iff \exists u \in \mathfrak{h} \ \bullet \ \mathsf{P}(\mathfrak{h}, x, u) \wedge \mathsf{E}(\mathfrak{h}, u, y) \\ \forall x \in \mathfrak{f}, y \notin \mathfrak{f} \ \bullet \ \mathsf{P}(\mathfrak{f}, x, y) &\iff \exists u \in \mathfrak{f} \ \bullet \ \mathsf{P}(\mathfrak{f}, x, u) \wedge \mathsf{E}(\mathfrak{f}, u, y) \end{aligned} \tag{10}$$

Checking relative convexity of footprints. In terms of reasoning about calls, we emit **assume** statements for our path partitioning formulas *both* before and after a method call (to decompose paths into those matching the callee's footprint and frame before the call, and to reconstruct information from these sources back to the client's footprint, afterwards; cf. Fig. 4). In both cases, before assuming our path-partitioning formulas, we first check that the footprint is relatively convex (since this property justifies their soundness); as we show in Fig. 6, a method's footprint could be relatively convex before the call but non-convex in the client's footprint afterwards. The two checks employed by our technique must be expressed in slightly different terms. Before the call (and without yet emitting our path-partitioning formulas) the $\mathfrak{g}$-local reachability information is available, and we directly use the formula from Def. 1:

$$\forall x, y \in \mathfrak{h}, u \in \mathfrak{f} \ \bullet \ \neg \mathsf{P}(\mathfrak{g}, x, u) \vee \neg \mathsf{P}(\mathfrak{g}, u, y) \tag{11}$$

However, after the call we obtain the $\mathfrak{h}$-local reachability from the postcondition of the callee, while the $\mathfrak{f}$-local reachability is preserved. We cannot use $\mathfrak{g}$-local reachability in the post-state of the call; the aim of our path-partitioning formulas is to *deduce* information in this form, and these are only justified *after* making the convexity check. Therefore, after the method call, we use the following alternative formulation:

$$\forall x, y \in \mathfrak{h}, u \in \mathfrak{f} \ \bullet \ \neg \mathsf{P}(\mathfrak{h}, x, u) \vee \neg \mathsf{P}(\mathfrak{f}, u, y) \tag{12}$$

Fig. 7. Encoding `merge` in Viper. Types are translated directly. The specifications (Sec. 2.1) are omitted for brevity. The reference field update is translated via `unlinkDAG`, `linkDAG` (Sec. 3.2). The method call is augmented with *local assumptions* in the form of macros (lines starting with capital letters), constraining the states `l1`, `l2`. The macros with infix `Convex` also check relative convexity of corresponding footprints (Sec. 3.3). The complete encoding is part of the publicly available artifact [Ter-Gabrielyan et al. 2019b].

Relating reachability information before and after a method call. Based solely on our assumption of relatively convex footprints, we now have a rich set of formulas available for precisely relating reachability information before and after a method call. To illustrate how our formulas can be used in practice, we consider one of the verification conditions needed for verifying the postcondition in Fig. 1 after the recursive call to `merge`. Concretely, we consider the following Hoare triple:

$$\{\ \texttt{l.right} \neq \texttt{null} \wedge \mathsf{P}_0(\mathfrak{g}, \texttt{r}, n)\ \}\ \texttt{link := merge(l.right, r, } \mathfrak{h}\texttt{, ...)}\ \{\ \mathsf{P}(\mathfrak{g}, \texttt{l}, n)\ \}$$

Here, $\mathfrak{g}$ and $\mathfrak{h}$ are the footprints of the client and the callee, resp., `l` and `r` are the roots of the left and right DAGs, resp. (Fig. 2), and n is *some* node reachable from `r` in the pre-state; we omit the rest of the arguments of `merge` for brevity. The condition `l.right` $\neq$ `null` comes from the **`if`** statement in Fig. 1 and holds before (and after) the recursive call; we enter this branch *iff* we have not yet found the `link` node and must keep recursively traversing the current structure. We proceed with a proof sketch for the postcondition of this Hoare triple. Other checks needed to verify the Hoare triple include relative convexity checks (11) and (12), and the precondition check before the recursive call; these require similar reasoning steps and are omitted for brevity.

The postcondition $\mathsf{P}(\mathfrak{g}, \texttt{l}, n)$ expresses the existence of a path $\texttt{l} \ldots n$. We justify this postcondition by showing the existence of a (single-edge) frame-local path $\texttt{l} \ldots \texttt{l.right}$ and an $\mathfrak{h}$-local path $\texttt{l.right} \ldots n$ (where $\mathfrak{h}$ is the footprint of the call). The former sub-path starts in $\texttt{l} \notin \mathfrak{h}$ and ends in $\texttt{l.right} \in \mathfrak{h}$ (since $\mathfrak{h} = \texttt{nldag} \uplus \texttt{rdag}$, where `nldag` was constructed via `sub`), and the latter sub-path starts in $\texttt{l.right} \in \mathfrak{h}$ and ends in $n \in \mathfrak{h}$ ($n \in \texttt{rdag}$ follows from the precondition $\mathsf{P}_0(\mathfrak{g}, \texttt{r}, n)$ of the Hoare triple and the last precondition of `merge`, while $\texttt{rdag} \subseteq \mathfrak{h}$ because $\mathfrak{h} = \texttt{nldag} \uplus \texttt{rdag}$). The distribution of the starting and ending nodes of these sub-paths enables an instantiation of (ii) from (9) with `l`, `l.right`, n for x, a, y, resp., reducing the overall proof goal to the two predicates $\mathsf{P}(\mathfrak{f}, \texttt{l}, \texttt{l.right})$ and $\mathsf{P}(\mathfrak{h}, \texttt{l.right}, n)$. First, since $\texttt{l.right} \neq \texttt{null}$, the former predicate can be justified by (5) and the postcondition of `snapshot`. Second, we instantiate the last postcondition of `merge` with `l.right`, n for x, y, resp. in order to reduce the latter predicate to $\mathsf{P}_0(\mathfrak{h}, \texttt{l.right}, \texttt{link})$ and $\mathsf{P}_0(\mathfrak{h}, \texttt{r}, n)$. Note that, since the path $\texttt{l.right} \ldots \texttt{link}$ starts in the root of `nldag`, the former predicate is implied by the last postcondition of `merge`. We can justify the latter predicate, $\mathsf{P}_0(\mathfrak{h}, \texttt{r}, n)$, with an instantiation of (i) from (9) with `r`, n for x, y, resp., since (as we argued above) $\texttt{r}, n \in \mathfrak{h}$. □

3.4 Frame-Localized Reachability

The ingredients presented thus far form the core of our solution for handling method calls, but are not yet sufficient to preserve reachability information in all cases, as we explain next. In some cases, we need to be able to localize reachability information in the *frame* of the call (cf. the left branch in Fig. 4). But precise frame-local reachability information cannot be obtained the same way as footprint-local reachability because, unlike method footprints, our technique permits the frame to be *non-convex* in the client's footprint. For example, consider a call to a method that operates on an acyclic list segment; the footprint of this call must be convex, while the frame would generally be non-convex in the entire list. Since the issue is subtle, we illustrate how information can be lost with a concrete example, and then show how to plug the gap.

The problematic scenario. The program in Fig. 8 consists of two methods: the client, `joinAndModify`, and the callee, `disconnectAll`. This program[2] concerns a particular shape of DAG structure, which we call a *hammock* between two nodes. We say that a (closed) DAG $\mathfrak{h}$ is a hammock between two (distinct) nodes s and t *iff* it consists of all nodes reachable from its node s (called the *source*) that reach its distinct node t (called the *sink*):

[2]The essential property of the example in Fig. 8 is that the client method *creates* heap edges inside the footprint of the callee (and not just in the frame of the call), whereas the callee *destroys* some of the paths in its footprint.

```
method joinAndModify(𝔤: Graph,
      𝔣: Graph, s1: Node, t1: Node,
      𝔥: Graph, s2: Node, t2: Node)
  requires 𝔤 = 𝔣 ⊎ 𝔥 ∧
    HAMMOCK_𝔤(𝔣, s1, t1) ∧ HAMMOCK_𝔤(𝔥, s2, t2)
    ∧ s1.left = null ∧ t2.right = null
  ensures P(𝔤, s1, t1)
{ /* state 0 */ s1.left := s2; t2.right := t1
  /* state 1 */ disconnectAll(𝔥)
  /* state 2 */ }
```

```
method disconnectAll(𝔤: Graph)
  ensures (∀x, y ∈ 𝔤 • P(𝔤, x, y) ⇔ x = y) ∧
    ∀x ∈ 𝔤, y ∉ 𝔤 • E(𝔤, x, y) ⇔ E0(𝔤, x, y)
```

The method `disconnectAll` destroys all non-trivial paths inside $\mathfrak{h}$ (exemplifying a possible destructive update to the heap structure).

Fig. 8. The method `joinAndModify` first attaches the hammock $\mathfrak{h}$ to the hammock $\mathfrak{f}$, creating a larger hammock, and then calls the method `disconnectAll`, creating a frame that is non-convex in $\mathfrak{g}$. Verification of the postcondition is challenging, as it requires localizing reachability in the frame, $\mathfrak{f}$, of the call to `disconnectAll`. Fig. 10 illustrates a typical run of `joinAndModify`.

$$\begin{aligned}\mathsf{HAMMOCK}_{\mathfrak{g}}(\mathfrak{h}, s, t) \;:\Longleftrightarrow\; & s \in \mathfrak{h} \wedge t \in \mathfrak{h} \wedge \mathsf{CLOSED}(\mathfrak{h}) \wedge \mathsf{ACYCLIC}_{\mathfrak{g}}(\mathfrak{h}) \wedge s \neq t \wedge \\ & \forall n \in \mathfrak{h} \bullet \mathsf{P}(\mathfrak{g}, s, n) \wedge \mathsf{P}(\mathfrak{g}, n, t)\end{aligned} \tag{13}$$

$$\mathsf{ACYCLIC}_{\mathfrak{g}}(\mathfrak{h}) \;:\Longleftrightarrow\; \mathfrak{h} \subseteq \mathfrak{g} \wedge \forall x, y \in \mathfrak{h} \bullet \neg\mathsf{E}(\mathfrak{g}, x, y) \vee \neg\mathsf{P}(\mathfrak{g}, y, x) \tag{14}$$

We start reasoning about `joinAndModify` in state 0, with the footprint being comprised of two (disjoint) hammocks, $\mathfrak{f}$ and $\mathfrak{h}$, where s_1 and s_2 are their sources and t_1 and t_2 are their sinks, resp. The first two operations are field updates, resulting in state 1. They join the two hammocks into one by creating exactly two edges: (s_1, s_2) and (t_2, t_1). Hence, there must exist at least two distinct paths from s_1 to t_1 in state 1: one path through the nested hammock, $\mathfrak{h}$, and one inside $\mathfrak{f}$. Note that this makes the subheap $\mathfrak{f}$ *non-convex* in $\mathfrak{g}$, even though $\mathfrak{h}$ is still relatively convex in $\mathfrak{g}$. The last operation in `joinAndModify` is a method call with a (relatively convex) footprint, $\mathfrak{h}$, which results in state 2. The callee method, `disconnectAll`, destroys all heap paths *inside* its footprint (first conjunct of the postcondition), while preserving all of its exit points (second conjunct of the postcondition). We omit the callee's implementation because the problem that we are about to explain occurs exclusively at the call site.

The postcondition of `joinAndModify` says that there *still* exists a $\mathfrak{g}$-local path $s_1 \ldots t_1$ in state 2. Intuitively, this claim should hold, as these two nodes *were*, before the call to `disconnectAll`, reachable via at least one $\mathfrak{f}$-local path that could not have been destroyed as a result of the method call (because $\mathfrak{f}$ is the frame of that call). However, our path-partitioning formulas (9) do not capture that such a frame-local path definitely existed; we learn from cases (iv)–(v) of (9) only the *disjunction* describing that at least one of the two paths from s_1 to t_1, labeled "iv" and "v" in Fig. 5, must have existed before the call, but we do not know which. Since the call is known to destroy the paths corresponding to one disjunct, we cannot deduce $\mathsf{P}_2(\mathfrak{g}, s_1, t_1)$ after the call unless we can precisely derive frame-local reachability.

Localizing reachability in the frame of a relatively convex footprint. Fig. 9 demonstrates the general problem of localizing reachability information in the frame of a method call. Consider a method call with a relatively convex footprint $\mathfrak{h}$ and the frame $\mathfrak{f}$; the client's footprint $\mathfrak{g}$ is their disjoint union $\mathfrak{g} = \mathfrak{f} \uplus \mathfrak{h}$. Our path partitioning formulas (9) allow us to precisely define $\mathfrak{h}$-*local* reachability based solely on $\mathfrak{g}$-local reachability. As demonstrated by our `joinAndModify` example of Fig. 8, we

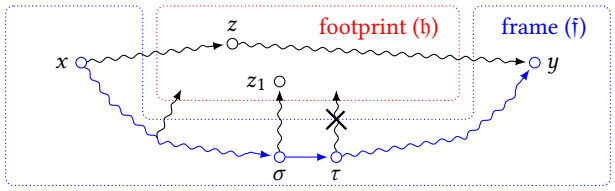

Fig. 9. Paths starting and ending in the frame of a relatively convex footprint may either be local to the frame or go through the footprint. Relative convexity of the footprint guarantees that there must be at most one entry and one exit point *per path*. Note that x and σ, as well as τ and y may possibly alias each other, but $x \neq y$ and $\sigma \neq \tau$ are guaranteed.

additionally need a complementary formula that would precisely define $\mathfrak{f}$-*local* reachability (again, based solely on $\mathfrak{g}$-local reachability). In other words, we are looking for a first-order formula over the relation P (with the first parameter fixed to $\mathfrak{g}$) that, for a given pair of nodes $x, y \in \mathfrak{f}$, precisely defines the existence of an $\mathfrak{f}$-local path $x \ldots y$. Fortunately, such an *in-frame reachability localization formula* exists if $\mathfrak{f}$ is a *frame of a relatively convex footprint* $\mathfrak{h}$ (even if $\mathfrak{f}$ itself is non-convex in $\mathfrak{g}$):

$$\begin{aligned}
&\forall x, y \in \mathfrak{f} \bullet \big(\forall z \in \mathfrak{h} \bullet \neg\mathsf{P}(\mathfrak{g}, x, z) \vee \neg\mathsf{P}(\mathfrak{g}, z, y)\big) \implies \big(\mathsf{P}(\mathfrak{f}, x, y) \Leftrightarrow \mathsf{P}(\mathfrak{g}, x, y)\big) \\
&\forall x, y \in \mathfrak{f} \bullet \quad \big(\exists z \in \mathfrak{h} \bullet \mathsf{P}(\mathfrak{g}, x, z) \wedge \mathsf{P}(\mathfrak{g}, z, y)\big) \implies \Big(\mathsf{P}(\mathfrak{f}, x, y) \Leftrightarrow \exists \sigma, \tau \in \mathfrak{f} \bullet \\
&\qquad\qquad \mathsf{P}(\mathfrak{g}, x, \sigma) \wedge \mathsf{E}(\mathfrak{g}, \sigma, \tau) \wedge \mathsf{P}(\mathfrak{g}, \tau, y) \wedge \\
&\qquad\qquad \big(\exists z_1 \in \mathfrak{h}.\, \mathsf{P}(\mathfrak{g}, \sigma, z_1)\big) \wedge \neg\big(\exists z_2 \in \mathfrak{h}.\, \mathsf{P}(\mathfrak{g}, \tau, z_2)\big)\Big)
\end{aligned} \tag{15}$$

We explain and justify (15) using the diagram of Fig. 9. Generally, since $\mathfrak{g} = \mathfrak{f} \uplus \mathfrak{h}$, we can case split on whether there exists a path $x \ldots y$ that goes through $\mathfrak{h}$, allowing us to obtain the required $\mathfrak{f}$-local reachability formula. The first formula above covers the case in which such a path *does not* exist; thus, the following must hold: $\forall z \in \mathfrak{h}.\ \neg\mathsf{P}(\mathfrak{g}, x, z) \vee \neg\mathsf{P}(\mathfrak{g}, z, y)$ (which trivially holds for all $x, y \in \mathfrak{f}$ in the special case when $\mathfrak{f}$ *is* convex in $\mathfrak{g}$). The second formula above says that, if there exists a path through $\mathfrak{h}$ (the upper kind of path in Fig. 9), it must pass through some node $z \in \mathfrak{h}$; hence, the following condition must hold: $\exists z \in \mathfrak{h}.\ \mathsf{P}(\mathfrak{g}, x, z) \wedge \mathsf{P}(\mathfrak{g}, z, y)$. Under this condition, we must define the existence of an $\mathfrak{f}$-local path that *also* connects $x \ldots y$. The key idea that we exploit to justify the second formula in (15) is to use our relative convexity assumption to justify a *three-way* split of the (hypothetical) $\mathfrak{f}$-local path $x \ldots y$ into three segments: a path $\mathsf{P}(\mathfrak{f}, x, \sigma)$, an edge $\mathsf{E}(\mathfrak{f}, \sigma, \tau)$, and a path $\mathsf{P}(\mathfrak{f}, \tau, y)$ (the lower kind of path in Fig. 9). Furthermore, we choose (σ, τ) such that σ is *the last node that reaches* $\mathfrak{h}$ $(\exists z_1 \in \mathfrak{h}.\ \mathsf{P}(\mathfrak{g}, \sigma, z_1))$ and τ is *the first node that does not reach* $\mathfrak{h}$ $(\neg\exists z_2 \in \mathfrak{h}.\ \mathsf{P}(\mathfrak{g}, \tau, z_2))$. Under our assumptions about x and y, this requirement can always be satisfied because the footprint of the callee, $\mathfrak{h}$, is reachable from (at least) the node x and is unreachable from (at least) the node y.

We summarize the conditions under which the predicates defining the three-way split of our hypothetical path $x \ldots (\sigma, \tau) \ldots y$ can be rewritten with $\mathfrak{g}$ instead of $\mathfrak{f}$, without losing precision:

- The footprint of the call, $\mathfrak{h}$, is convex in the client's footprint, $\mathfrak{g}$.
- Both nodes x and y are in the frame of the call, $\mathfrak{f}$.
- We picked σ that reaches $\mathfrak{h}$ and τ that does not reach $\mathfrak{h}$ s.t. (σ, τ) is on the path $x \ldots y$.

For the first predicate, we need to argue by contradiction: suppose that $\mathsf{P}(\mathfrak{g}, x, \sigma)$ were the case, but $\mathsf{P}(\mathfrak{f}, x, \sigma)$ were not (the opposite implication is direct, since $\mathfrak{f} \subseteq \mathfrak{g}$). Then, the path from x to σ must visit the callee's footprint, $\mathfrak{h}$. However, by construction, σ is known to have a path to some node

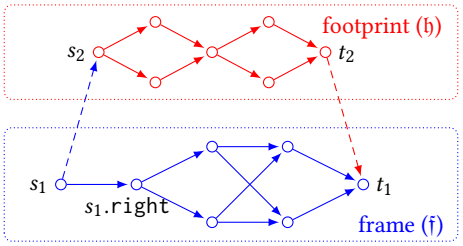

Fig. 10. An example scenario of running the method `joinAndModify`. In state 0, only the solid edges exist. In state 1, the client has created two new edges: (s_1, s_2) and (t_2, t_1). In state 2, the callee has destroyed all solid-red edges, but there still exists a path $s_1 \ldots t_1$ via solid-blue edges. However, we cannot deduce the existence of this path using just the path partitioning formulas alone due to the disjunction in case (iv)–(v) of (9) that does not allow us to distinguish whether *all* paths between s_1 and t_1 were passing through the footprint in state 1. Therefore, recovering this bit after the call to `disconnectAll` requires precise localization of reachability information in the frame (15).

in the callee's footprint, and this violates the assumption that this footprint is relatively convex. Hence, we get $\mathsf{P}(\mathfrak{f}, x, \sigma) = \mathsf{P}(\mathfrak{g}, x, \sigma)$. The second predicate is easiest: a single edge between two nodes in the frame ($\mathfrak{f}$) can only depend on the frame itself; therefore, we get $\mathsf{E}(\mathfrak{f}, \sigma, \tau) = \mathsf{E}(\mathfrak{g}, \sigma, \tau)$. The third predicate expresses the existence of a path $\tau \ldots y$; since we have picked τ s.t. it does not reach the footprint, such a path exists in this case exactly when it exists in the frame, giving $\mathsf{P}(\mathfrak{f}, \tau, y) = \mathsf{P}(\mathfrak{g}, \tau, y)$. Thus, our construction of σ and τ, along with our relative convexity property for method footprints, allows us to justify the formulation in (15). These formulas now provide the missing ingredient for our technique that complements our path-partitioning formulas of (9). □

Revisiting the problematic scenario. We return to our `joinAndModify` method, and show that we can now verify the last conjunct of its postcondition. Previously, we were unable to verify $\mathsf{P}_2(\mathfrak{g}, s_1, t_1)$ after the call to `disconnectAll`, while intuitively, a $\mathfrak{g}$-local path $s_1 \ldots t_1$ exists in state 2, because the method call could not have destroyed the existing *frame*-local path $s_1 \ldots t_1$ that existed in state 1 (Fig. 10). Thus, if we could deduce $\mathsf{P}_1(\mathfrak{f}, s_1, t_1)$ (before the call to `disconnectAll`), we would obtain our proof goal. This is now possible using the second equation from (15): instantiating s_1 for x and t_1 for y, we deduce the hypothesis of the implication, since before the call to `disconnectAll` we can deduce that paths from s_1 to t_1 exist passing through the footprint. To deduce $\mathsf{P}_1(\mathfrak{f}, s_1, t_1)$ from our formula, we need to obtain the following property (recall that $\mathfrak{g} = \mathfrak{f} \uplus \mathfrak{h}$):

$$\exists \sigma, \tau \in \mathfrak{f}.\ \mathsf{P}(\mathfrak{g}, s_1, \sigma) \wedge \mathsf{E}(\mathfrak{g}, \sigma, \tau) \wedge \mathsf{P}(\mathfrak{g}, \tau, t_1) \wedge \big(\exists z_1 \in \mathfrak{h}.\ \mathsf{P}(\mathfrak{g}, \sigma, z_1)\big) \wedge \neg\big(\exists z_2 \in \mathfrak{h}.\ \mathsf{P}(\mathfrak{g}, \tau, z_2)\big)$$

The existentially-quantified pair (σ, τ) can be witnessed by $(s_1, s_1.\texttt{right})$. From this, and our hammock properties (13), all conditions above follow directly, allowing us to deduce our intermediate proof goals, $\mathsf{P}_{1,2}(\mathfrak{f}, s_1, t_1)$, and use the last case of (9) to deduce the ultimate proof goal, $\mathsf{P}_2(\mathfrak{g}, s_1, t_1)$. □

Together with (15), the ingredients of our technique presented in Fig. 4 empower *completely general, precise* reasoning about reachability in the presence of method calls with relatively convex footprints. Note that, in examples where stronger properties are known about a method's footprint, our formulas from (15) reduce to much simpler criteria. In particular, if a method operates on a *closed* data structure (no paths leave the footprint), we can always apply the first of our formulas; the full expressiveness of our conditions is required only in the presence of potential paths crossing the callee's footprint (e.g., Fig. 10). Our technique is complete, provided that callee postconditions specify sufficient information about reachability within their footprint. However, even in examples

where this information is incomplete, our technique is applicable and provides useful information at the call site, for instance, by deducing which frame-local paths will be preserved across a method call. It is the restriction to method calls with relatively convex footprints which enables us to express appropriate formulas to preserve this information; without this restriction, we would not be able to precisely define the existence of frame-local paths exclusively via coarser reachability information. Finally, we note on the efficiency of formulas (15): in our encoding (demonstrated in Fig. 7 and validated in Sec. 5), we supply appropriate triggers for the universal quantifiers and Skolemize the existential quantifiers.

This completes our treatment of acyclic graphs and method calls; the latter is the most complex part of our technique, and applies equally to the cyclic case, which we tackle in the next section.

4 REACHABILITY IN CYCLIC STRUCTURES

In the previous section, we presented our technique for enabling modular reasoning about heap reachability in combination with first-order separation logic. The presented technique operates under two key restrictions: (1) that method footprints are always relatively convex in their client's footprint and (2) that all footprints used contain acyclic graphs. These are two *independent* criteria, which our technique checks where necessary. Restriction (1) alone enables our handling of method calls. In this section, we show that we can adapt our technique to a particular setting in which restriction (2) is dropped: that of general *0–1-path graphs*. A graph is called a 0–1-path graph (hereafter, ZOPG) if there exists *at most one* (non-trivial) path (modulo cycles) between all pairs of nodes in the graph; for instance, $\{(a,b),(b,c),(c,a),(c,d)\}$ is a ZOPG, but $\{(a,b),(b,a),(a,c),(c,a)\}$ is not since there are *two* distinct (non-trivial) paths from a to itself [Dong and Su 1995]. Although this notion does not permit *arbitrary* cyclic graphs, the technique presented in this section allows us to adapt our work to reason about reachability in the presence of potentially-cyclic lists in the heap or more-complex data structures consisting of these, including, for example: trees where the children of each node are stored in a cyclic list (e.g., using Java LinkedList), generalized tree-like structures in which some nodes consist of rings, and the ring representation of heap-ordered trees [Fredman et al. 1986]. Therefore, the ZOPG class is an important generalization of (potentially-cyclic) singly-linked lists, which is the class handled in the closest prior work [Itzhaky et al. 2014].

Extending our technique to ZOPGs requires a new way of handling direct field updates (Sec. 4.1); our handling in Sec. 3 depended on acyclicity, and a way to retain that certain graphs in the program *are* ZOPGs; modifying a ZOPG by adding an edge could violate the ZOPG invariant (Sec. 4.2). Note that our requirement of relatively convex footprints (Def. 1) is again crucial, enabling an efficient solution of the latter problem.

4.1 Field Updates in ZOPGs

To support direct field updates, we adapt prior work [Dong and Su 1995] that shows how to precisely update a more-refined reachability relation called $\mathtt{DEP}$ for ZOPGs. There are few changes in our adaptation: our $\mathtt{DEP}$ relation is compatible with the *reflexive* reachability relation ($\mathtt{P}$), whereas Dong and Su work with irreflexive reachability, and we parameterize our $\mathtt{DEP}$ relation with two extra parameters, F and $\mathfrak{g}$, supporting separation-logic reasoning, as we did for $\mathtt{P}$ in Sec. 2.2. The predicate $\mathtt{DEP}^F(\mathfrak{g},x,y,u,v)$ expresses the existence of a (non-trivial) path of field references from x to y such that all objects on the path (except possibly y) are in $\mathfrak{g}$, all fields are in F, and the path *depends on* the edge (u,v). Intuitively, this means that removing (u,v) from the graph will destroy the path $x \ldots y$ (which is the unique path from x to y in a ZOPG). We will omit the parameter F when it is clear from the context. Note that $\mathtt{DEP}(\mathfrak{g},x,y,u,v) \Rightarrow u \neq v$ since an edge (v,v) cannot be a dependency of any path: deleting such an edge would not affect reachability. Note also that $\mathtt{DEP}(\mathfrak{g},x,y,u,v) \Rightarrow x \neq y$ since a trivial path $x \ldots x$ does not depend on any edges.

Although precisely updating the classical reachability relation P in potentially-cyclic graphs after destructive heap operations is beyond first-order logic (and cannot be efficiently automated), the information about the DEP relation *can* be updated precisely and efficiently after such destructive operations [Dong and Su 1995]. For example, if the edge (s, t) is deleted by executing the statement `s.adj := null` in a method with footprint $\mathfrak{g}$, then the new relation, DEP, can be simply expressed via the old relation DEP_0 as follows:

$$\forall x, y, u, v \bullet \mathsf{DEP}^F(\mathfrak{g}, x, y, u, v) \iff \mathsf{DEP}_0^F(\mathfrak{g}, x, y, u, v) \wedge \neg \mathsf{DEP}_0^F(\mathfrak{g}, x, y, \mathtt{s}, \mathtt{t}) \tag{16}$$

For fixed F and $\mathfrak{g}$, the intuition for (16) is this: (x, y, u, v) is in the new relation *iff* it was in the old relation and the deleted edge (s, t) was not a dependency of the path $x \ldots y$ before the update.

Precisely updating DEP after an operation that only creates an edge (e.g., by executing the statement `s.adj := t`) is also possible, provided one additionally checks that the newly-created edge does not violate the ZOPG invariant; we describe how this check is enforced in Sec. 4.2. As before, a general field update entails removing and then adding an edge (see [Ter-Gabrielyan et al. 2019a, App. B]). Our treatment of the DEP relation is similar to the treatment of the P relation described in Sec. 3.2: since the mathematical definitions of these relations are beyond first-order logic, we provide the verifier with a *partial* axiomatization (see [Ter-Gabrielyan et al. 2019a, App. C]). We rewrite each field update with a call to an internal `updateZOPG` method with the same footprint $\mathfrak{g}$ as for the current method; the postconditions of `updateZOPG` make the DEP update formulas (e.g., (16)) available to the SMT solver.

A technical difference between our reachability relation P and the DEP relation is that the latter carries richer information (in particular, knowledge of every edge on which each path depends). Conversely, it seems unlikely that having to enumerate all edge facts in a graph would be suitable for a method specification; the abstraction provided by P is typically desirable. Thus, we do not provide DEP as a primitive in our specifications, and instead provide a means of converting between information in one relation and the other, while losing as little information as possible. Our conversion rules are based on the following main axiom:

$$\forall \mathfrak{h}, x, y \bullet \mathsf{P}^F(\mathfrak{h}, x, y) \wedge x \neq y \iff \exists u, v \bullet \mathsf{DEP}^F(\mathfrak{h}, x, y, u, v) \tag{17}$$

Unlike the update formulas that are emitted for concrete method footprints, our conversion axioms (e.g., (17)) *quantify* over the footprint ($\mathfrak{h}$); as before, we carefully select the triggers for these axioms to guide the SMT solver's quantifier instantiation procedure.

In general, formula (17) does not capture full information in principle expressible with the DEP relation; intuitively, this is because a single path $x \ldots y$ (described by the LHS) may depend on multiple edges, all of which match the RHS existential quantifier. To partially mitigate this fact, we augment our axiomatization with a number of additional properties. For instance, one can easily prove the following formula (an axiom in our technique) about ZOPGs, providing (for fixed F and $\mathfrak{g}$) some quadruples which *do not* belong to the DEP relation:

$$\forall \mathfrak{h}, u, v, w \bullet \neg \mathsf{DEP}^F(\mathfrak{h}, v, w, u, v) \tag{18}$$

Note that if $v = w$, $\neg\mathsf{DEP}(\mathfrak{h}, v, v, u, v)$ holds because a trivial path $v \ldots v$ does not depend on any edges. Assume $v \neq w$. There can be at most one (cycle-free) path from v to w in a ZOPG. If there are no paths from v to w, then we get $\neg\mathsf{DEP}(\mathfrak{h}, v, w, u, v)$ from (17). Otherwise, the edge (u, v) is not part of the (cycle-free) path $v \ldots w$ and cannot be one of its dependencies. □

Our ZOPG axiomatization is based on a set of formulas like (18) that, together with (17), help reasoning about the DEP relation (we provide the full axiomatization in [Ter-Gabrielyan et al. 2019a, App. C]). Equipped with this conversion between relations P and DEP, precise reachability information is preserved in all cases that we have observed. This is interesting because the DEP

relation carries more information than the transitive relation P, so (for fixed F and $\mathfrak{g}$) not all quadruples (x, y, u, v) in DEP can be extracted precisely from P, even if all pairs (x, y) in P are known. Intuitively, these missing quadruples appear not to be needed in practice because the overall proof goals are phrased in terms of just P (and not DEP).

We illustrate how reachability information is preserved throughout the transformations between P and DEP with a concrete example. Consider the following Hoare triple that describes a heap update in a ZOPG with footprint $\mathfrak{g}$ and a single reference field `next`:

$$\left\{ \begin{array}{l} \mathsf{x}, \mathsf{y} \in \mathfrak{g} \wedge \mathsf{x.next} = \mathsf{y} \wedge \\ \forall n, m \in \mathfrak{g} \bullet \mathsf{P}_0(\mathfrak{g}, n, m) \end{array} \right\} \ \mathtt{x.next := null} \ \left\{ \ \forall m \in \mathfrak{g} \bullet \mathsf{P}(\mathfrak{g}, \mathsf{y}, m) \ \right\}$$

We can justify the postcondition assertion as follows. Consider an arbitrary node $m \in \mathfrak{g}$. If $m = \mathsf{y}$, then we trivially get $\mathsf{P}(\mathfrak{g}, \mathsf{y}, \mathsf{y})$. Otherwise, we assume $m \neq \mathsf{y}$, and the remaining proof obligation is $\mathsf{P}(\mathfrak{g}, \mathsf{y}, m)$; to justify this, we need to exploit information from the pre-state. Since by (17), we can reduce the current proof obligation to $\mathsf{DEP}(\mathfrak{g}, \mathsf{y}, m, \mathsf{y}, \mathsf{y.next})$, we can instantiate the DEP update formula (16), obtaining two pre-state conditions: $\mathsf{DEP}_0(\mathfrak{g}, \mathsf{y}, m, \mathsf{y}, \mathsf{y.next})$ and $\neg\mathsf{DEP}_0(\mathfrak{g}, \mathsf{y}, m, \mathsf{x}, \mathsf{y})$. The former is justified by the precondition quantifier (providing $\mathsf{P}_0(\mathfrak{g}, \mathsf{y}, m)$) and the main conversion axiom (17), whereas the latter can be obtained directly from the additional conversion axiom (18).

This example, as well as our evaluation (Sec. 5), show that necessary reachability information can be fully recovered after the following steps: first, conversion from P to DEP, second, application of update formulas for the DEP relation, and third, conversion from DEP to P. We plan to investigate as future work the extent to which this approach is always precise for preserving reachability information of this kind.

4.2 Preservation of the ZOPG Invariant

To justify the handling of field updates from the previous subsection, we require knowledge that the graph being updated is a ZOPG. Since this fact can be violated by changes to the heap, an important question is how we can know if the ZOPG invariant holds. It can be expressed in first-order logic with the combination of edge and path predicates as follows:

$$\begin{aligned} \mathsf{ZOPG}(\mathfrak{h}) \ :\Longleftrightarrow \ & \big(\forall x_1, x_2, a, b \in \mathfrak{h}, y \ \bullet \ (x_1 \neq x_2 \vee a \neq b) \wedge \mathsf{P}(\mathfrak{h}, x_1, x_2) \wedge \mathsf{P}(\mathfrak{h}, x_2, x_1) \wedge \\ & \quad \mathsf{E}(\mathfrak{h}, x_1, a) \wedge \neg\mathsf{P}(\mathfrak{h}, a, x_1) \wedge \\ & \quad \mathsf{E}(\mathfrak{h}, x_2, b) \wedge \neg\mathsf{P}(\mathfrak{h}, b, x_2) \implies \neg\mathsf{P}(\mathfrak{h}, a, y) \vee \neg\mathsf{P}(\mathfrak{h}, b, y)\big) \ \wedge \\ & \forall x, a, b \in \mathfrak{h} \ \bullet \ a \neq x \wedge b \neq x \wedge \mathsf{E}(\mathfrak{h}, x, a) \wedge \mathsf{P}(\mathfrak{h}, a, x) \wedge \\ & \quad \mathsf{E}(\mathfrak{h}, x, b) \wedge \mathsf{P}(\mathfrak{h}, b, x) \implies a = b \end{aligned} \tag{19}$$

The first conjunct of the formula expresses a situation in which two (potentially aliasing) nodes x_1 and x_2 are on the same strongly-connected component (SCC), and two edges (starting in x_1 and x_2) that are *different*—at least by source or target—end in nodes a and b, resp., outside of the SCC (a and b may alias unless $x_1 = x_2$). In such a case, it is forbidden that any node y is reachable from both a and b (this would form two different paths from the SCC to y). The second conjunct restricts the structure of SCCs themselves: no two *different* edges may start in x and stay within the same SCC.

Intuitively, formula (19) is hard to automate because it uses a non-trivial combination of edge and reachability predicates. Establishing $\mathsf{ZOPG}(\mathfrak{h})$ would require, for example, the information about all path splits in $\mathfrak{h}$, i.e., all nodes $x \in \mathfrak{h}$ s.t. $\exists a, b \in \mathfrak{h} \ \bullet \ a \neq b \wedge \mathsf{E}(\mathfrak{h}, x, a) \wedge \mathsf{E}(\mathfrak{h}, x, b)$. Such details ultimately require specifications to enumerate edges in the graph, which is impractical, and breaks the abstraction that reachability specifications grant. Even if the full information about the edge relation were present, establishing $\mathsf{ZOPG}(\mathfrak{h})$ would require an induction proof that is beyond the power of modern SMT solvers. Instead of checking this invariant from scratch, we design a mechanism for checking that the ZOPG invariant is *preserved* across changes to the heap.

```
method testZopgObligations(g: Zopg, R: Graph, r: Node, u: Node)
  requires {u} ⊎ R ⊆ g ∧ CLOSED({u}) ∧ RING_g(R) ∧ r ∈ R ∧
           ∀x ∈ g, y • P(g, x, y) ∧ P(g, x, u) ⟹ ¬P(g, r, y)  // (Pre)
{
    var h: Zopg := {u} ⊎ R
    ringInsert(h, R, r, u)
}

method ringInsert(g: Zopg, R: Graph, r: Node, u: Node)
  requires g = {u} ⊎ R ∧ CLOSED({u}) ∧ RING_g(R) ∧ r ∈ R
  ensures RING_g(g) ∧ (∀n ∉ g • P(g, u, n) ⇔ P_0(g, r, n)) ∧
          ∀x ∈ g, y • x ≠ u ∧ y ≠ u ⟹ (P(g, x, y) ⇔ P_0(g, x, y))
{
    u.next := r.next
    r.next := u
}
```

Fig. 11. An example client with a ZOPG footprint. For simplicity, the method `testZopgObligations` has no postconditions. In order to verify it, one must nonetheless prove that the ZOPG invariant is maintained after the call to `ringInsert`. The definition of RING is given in (20). Note the different meaning of footprint parameters written in subscripts vs. those written in parentheses (e.g., $\mathfrak{g}$ and $\mathfrak{h}$, resp., in the definition of $\mathsf{SCC}_{\mathfrak{g}}(\mathfrak{h})$): the former are used as arguments for the E and P predicates, whereas the latter are used for restricting the domain of quantification.

$$
\begin{aligned}
\mathsf{RING}_{\mathfrak{g}}(\mathfrak{h}) &:\Longleftrightarrow \mathsf{FUNCTIONAL}(\mathfrak{h}) \wedge \mathsf{UNSHARED}(\mathfrak{h}) \wedge \mathsf{SCC}_{\mathfrak{g}}(\mathfrak{h}) \\
\mathsf{SCC}_{\mathfrak{g}}(\mathfrak{h}) &:\Longleftrightarrow \forall x, y \in \mathfrak{h} \bullet \mathsf{P}(\mathfrak{g}, x, y) \\
\mathsf{FUNCTIONAL}(\mathfrak{h}) &:\Longleftrightarrow \forall a, b, c \in \mathfrak{h} \bullet \mathsf{E}(\mathfrak{h}, a, b) \wedge \mathsf{E}(\mathfrak{h}, a, c) \implies b = c \\
\mathsf{UNSHARED}(\mathfrak{h}) &:\Longleftrightarrow \forall a, b, c \in \mathfrak{h} \bullet \mathsf{E}(\mathfrak{h}, a, c) \wedge \mathsf{E}(\mathfrak{h}, b, c) \implies a = b
\end{aligned}
\tag{20}
$$

Extending the specification language for potentially-cyclic footprints. As a first step, we introduce an additional annotation in our specification language, so that we can label certain method footprints as ZOPGs. In addition to general graphs (whose structure is only constrained by other specifications), such as $\mathfrak{h}$: ***Graph***, we allow the footprints of some methods to be more specifically marked as ZOPGs, using the syntax $\mathfrak{g}$: ***Zopg***. For method footprints declared this way, we will explain the additional proof obligations necessary to check that we *maintain* the ZOPG invariant. In particular, a method with footprint $\mathfrak{g}$: ***Zopg*** can be translated to a method with footprint $\mathfrak{g}$: ***Graph*** with additional *ZOPG proof obligations*.

We illustrate the generation of ZOPG proof obligations based on the example in Fig. 11. The client, `testZopgObligations`, operates on a ZOPG $\mathfrak{g}$ that includes two disjoint parts: a ring R and a (closed) singleton graph consisting of just one node, u. The extra node r denotes an arbitrary node of the ring. The only operation performed by the client is a call to `ringInsert`. To verify that $\mathfrak{g}$ remains a ZOPG by the end of `testZopgObligations`, we need to check that the callee does not create alternative paths—not just in its footprint, $\mathfrak{h}$ (which is guaranteed to remain a ZOPG, as the methods with footprints marked by ***Zopg*** are locally checked to preserve this property), but also in the larger subheap, $\mathfrak{g}$. The callee `ringInsert`, operates on a ZOPG that *equals* the union of two

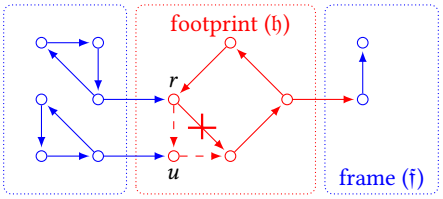

Fig. 12. A typical scenario of running `testZopgObligations`. r is an arbitrary node of the ring, and u is added to the ring after the call to `ringInsert`. The diagram demonstrates a data structure with nodes that can have two reference fields (for simplicity, the implementation of `ringInsert` shows a single field, `next`). The second conjunct in the client's precondition says that no (non-trivial) paths may originate from u (there may be paths ending in u). The footprint may have both entry and exit points, but (as required by the last conjunct in the client's precondition) each connected component of the frame may have *at most one* entry or exit point into the footprint; otherwise, the ZOPG invariant would be violated by the call.

disjoint parts: a closed singleton graph u and a ring R (these two parts must be mutually-unreachable in the pre-state). The callee attaches u to the ring R, resulting in a larger ring, $u \uplus R$. The callee's postcondition says that (in the post-state) its entire footprint is a ring (thus, all pairs of footprint nodes are mutually reachable), and precisely defines its local reachability. Local paths that end *outside* of the callee's footprint (i.e., *outgoing paths*) are defined by the last two conjuncts: the former says that all exit points reachable from the ring in the pre-state are exactly the exit points reachable from u in the post-state, whereas the latter preserves all outgoing paths of the initial ring, i.e., *all* exit points of the footprint in the pre-state where $\{u\}$ was closed. Fig. 12 illustrates the client's footprint in a state after `ringInsert` has executed.

Maintaining the ZOPG invariant after a field update. The knowledge that a subheap was a ZOPG in the pre-state of an operation helps checking that that subheap is still a ZOPG in the post-state, as we show next. We translate a general field update `u.next := v` in a method with the footprint `𝔤: Zopg` to `u.next := null; u.next := v`, where (assuming `v` is not null) the first update deletes an edge and the second one creates a new edge. Deleting edges does not alter the graph class of $\mathfrak{g}$. However, a newly added edge may create an alternative path between some nodes of the graph. Concretely, new paths will be created between all pairs of nodes (x, y) s.t. there exist two paths: $x \ldots \mathsf{u}$ and $\mathsf{v} \ldots y$. Therefore, we get the following soundness criterion (emitted as a proof obligation before the second update) for a field update in a ZOPG:

$$\mathsf{u} \neq \mathsf{v} \implies \forall x \in \mathfrak{g}, y \bullet \mathsf{P}(\mathfrak{g}, x, \mathsf{u}) \wedge \mathsf{P}(\mathfrak{g}, \mathsf{v}, y) \implies \neg \mathsf{P}(\mathfrak{g}, x, y) \tag{21}$$

Note that y may be outside of the current method's footprint because a $\mathfrak{g}$-local path may leave $\mathfrak{g}$ (*iff* its last edge leaves that subheap). The formula (21) is much simpler than, e.g., (19) because it is (a) an *incremental* condition (we used the knowledge that the subheap *was* a ZOPG before the update; otherwise, we would need to consider alternative paths other than those introduced by the new creation) and (b) the operation is a field update (hence, only one edge has been added to the graph). Keeping track of graph classes in the presence of method calls is more involved, but the idea (a) is again helpful for tackling this problem.

Maintaining the ZOPG invariant after a method call. A method call may violate the ZOPG invariant at call site even if the footprint of the call remains a ZOPG (a condition which is checked locally for the callee). The condition that a method call *does not* violate the ZOPG invariant at call site is generally as hard to check as the formula (19) itself. Fortunately, this condition can be drastically simplified if the footprints or the callee and the client are *relatively convex*.

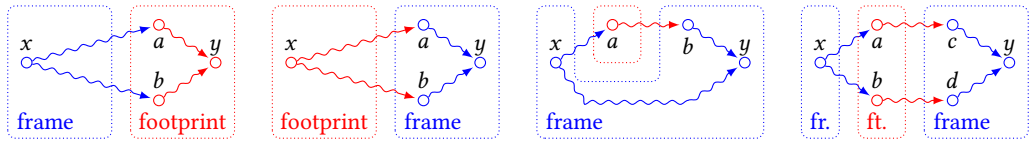

Fig. 13. The four configurations that violate the ZOPG invariant after a method call with a relatively convex ZOPG footprint (in red) and a ZOPG frame (in blue).

We proceed as follows. First, we enumerate the ways in which a method call that preserves the ZOPG invariant on its own footprint, could potentially *violate* that invariant for its client's footprint. In particular, this must be by the creation of at least one new path. A call to a method with a *convex* footprint may result in one of the four *bad heap configurations* (violating the ZOPG invariant) depicted in Fig. 13. Second, we conjoin the *negated* formulas (22), (23), (24), (25) characterizing these four bad configurations, comprising an efficient criterion for preserving the ZOPG invariant. Checking this criterion can be easily automated: unlike formula (19), our criterion requires no information about the edge relation whatsoever. Our technique encodes this criterion as a proof obligation for the client. Finally, we sketch a proof of completeness for the four cases in Fig. 13.

The bad configuration in Fig. 13 (α) corresponds to a scenario in which the method call has created an alternative path from x to y, where the former node does not belong to the callee's footprint. We can describe this configuration via the following formula:

$$\begin{aligned}\exists x \in \mathfrak{f}, a, b \in \mathfrak{h}, y \notin \mathfrak{f} \bullet\ & a \neq b \wedge \mathsf{P}_0(\mathfrak{f}, x, a) \wedge \mathsf{P}_0(\mathfrak{f}, x, b) \\ & \wedge \mathsf{P}(\mathfrak{h}, a, y) \wedge \neg \mathsf{P}_0(\mathfrak{h}, a, y) \wedge \mathsf{P}(\mathfrak{h}, b, y)\end{aligned} \tag{22}$$

The symbols P_0 and P denote the reachability relation before and after the method call; $\mathfrak{h}$ is the callee's footprint; $\mathfrak{f}$ is the frame of the call. We evaluate the first two reachability predicates in the old state because frame-local reachability is not affected by the call. The information about the last three predicates comes from the postcondition of the callee[3]. We assume w.l.o.g. that $a \ldots y$ has been *newly created* by the call (whereas $b \ldots y$ may have existed before the call). Both paths could not have existed before the call, as that would contradict our assumption that $\mathfrak{g}$ was a ZOPG.

Returning to our example of Fig. 11, we observe that the precondition of `testZopgObligations` is strong enough to prevent the bad configuration (α) after the call to `ringInsert`; we prove this by contradiction. Assume that, while preserving its local ZOPG invariant, the call results in the bad configuration (α) for some $x \in \mathfrak{f}, a, b \in \mathfrak{h}, y \notin \mathfrak{f}$; thus, we learn the conjuncts (say, #1 to #6) from the body of (22). Note that a and b must be distinct (due to #1) and cannot *both* be in R (due to #4 and #6; otherwise, there would be alternative paths, violating the ZOPG invariant of the callee's footprint in the post-state). We draw the contradiction by instantiating the last conjunct, (Pre), of the precondition of `testZopgObligations`: $\forall x \in \mathfrak{g}, y.\ \mathsf{P}(\mathfrak{g}, x, y) \wedge \mathsf{P}(\mathfrak{g}, x, u) \implies \neg \mathsf{P}(\mathfrak{g}, r, y)$. With our path partitioning formulas (9), #2 and #3 imply $\mathsf{P}_0(\mathfrak{g}, x, a)$ and $\mathsf{P}_0(\mathfrak{g}, x, b)$, resp. Together, #4 and #5 express that $a \ldots y$ is a *newly created path*; hence either $a = u$ or $y = u$ (see Fig. 12). If $a = u$, $y \neq u$, then $b \in R$; we draw the contradiction by instantiating (Pre) with x, b for x, y. Otherwise, $a \neq u$, $y = u = b$, then $a \in R$; we draw the contradiction by instantiating (Pre) with x, a for x, y. □

Similarly, we can describe the bad configuration in Fig. 13 (β) using the following formula:

$$\begin{aligned}\exists x \in \mathfrak{h}, a, b \in \mathfrak{f}, y \notin \mathfrak{h} \bullet\ & a \neq b \wedge \mathsf{P}_0(\mathfrak{f}, a, y) \wedge \mathsf{P}_0(\mathfrak{f}, a, y) \\ & \wedge \mathsf{P}(\mathfrak{h}, x, a) \wedge \neg \mathsf{P}_0(\mathfrak{h}, x, a) \wedge \mathsf{P}(\mathfrak{h}, x, b)\end{aligned} \tag{23}$$

[3]It is also possible to get the information about the old reachability relation from a modified version of (22) where $\neg \mathsf{P}_0(\mathfrak{h}, a, y)$ is dropped and all other path predicates are evaluated in the pre-state.

In this configuration, the source of the alternative paths falls into the callee footprint, and their end into the frame; this results in alternative paths $x \ldots a \ldots y$ and $x \ldots b \ldots y$. In our example of Fig. 11, the new outgoing paths that `ringInsert` creates originate in u; all other outgoing paths also existed *before* the call (due to the last postcondition). In order to avoid the bad configuration (β), `testZopgObligations` requires that *no paths* may originate in the attached node u. Thus, any *new outgoing path* must pass through R before it reaches the callee's footprint. Since `ringInsert` preserves the paths that start in R and end in the frame (due to its last postcondition), and we assumed that the callee's footprint is a ZOPG before and after the call, the last three conjuncts in (23) cannot be satisfied. Hence, our specification is strong enough to prevent (β). □

The scenario depicted in Fig. 13 (γ) illustrates that *any new path* $a \ldots b$ created by the method call, combined with suitable frame paths, may violate the ZOPG invariant:

$$\exists x, b \in \mathfrak{f}, a \in \mathfrak{h}, y \notin \mathfrak{h} \bullet \mathsf{P}_0(\mathfrak{f}, x, y) \wedge \mathsf{P}_0(\mathfrak{f}, x, a) \wedge \mathsf{P}_0(\mathfrak{f}, b, y) \wedge \mathsf{P}(\mathfrak{h}, a, b) \wedge \neg \mathsf{P}_0(\mathfrak{h}, a, b). \tag{24}$$

In order to avoid the bad configuration (γ), we must ensure that an arbitrary frame node x that reaches the footprint node a does not reach any of the frame nodes (e.g., y) that will be reachable from a after the call. In our example of Fig. 11, the precondition of `testZopgObligations` is strong enough to prevent (γ) after the call to `ringInsert`. The nodes x and y in the last conjunct of this precondition can be thought of as those in (24) and Fig. 13 (γ); the condition rules out the possibility that the effect of the call will connect up such alternative path. □

The most subtle bad configuration is Fig. 13 (δ), where both alternative paths $x \ldots y$ go via the footprint of the method call. This heap configuration can be expressed via the following formula:

$$\begin{aligned}\exists x, c, d \in \mathfrak{f}, a, b \in \mathfrak{h}, y \notin \mathfrak{h} \bullet\ & a \neq b \wedge c \neq d \wedge \mathsf{P}_0(\mathfrak{f}, x, a) \wedge \mathsf{P}_0(\mathfrak{f}, x, b) \wedge \mathsf{P}_0(\mathfrak{f}, c, y) \wedge \mathsf{P}_0(\mathfrak{f}, d, y) \\ & \wedge \mathsf{P}(\mathfrak{h}, a, c) \wedge \neg \mathsf{P}_0(\mathfrak{h}, a, c) \wedge \mathsf{P}(\mathfrak{h}, b, d).\end{aligned} \tag{25}$$

This configuration can be realized when a and b are *mutually unreachable in both states* (otherwise, the configuration is covered by (α) and (β)). This configuration cannot occur in the post-state of our example because after the method call u is attached to the ring. □

Completeness proof sketch. To derive the four cases in Fig. 13, consider a ZOPG subheap $\mathfrak{g}$ comprised of the ZOPG frame $\mathfrak{f}$ and the (relatively convex) ZOPG footprint $\mathfrak{h}$ of a method call. Assume that the method call creates at least one new path s.t. the ZOPG invariant of its footprint is maintained while the ZOPG invariant of the (larger) client's footprint is violated. Consider as well two nodes x and y that are connected (in the state after the call) by *multiple* (at least two) $\mathfrak{g}$-local paths $x \ldots y$. We assume that x and y are both in $\mathfrak{g}$; if y is outside $\mathfrak{g}$, we first apply (10), providing some node $u \in \mathfrak{g}$ s.t. $\mathsf{P}(\mathfrak{g}, x, u) \wedge \mathsf{E}(\mathfrak{g}, u, y)$; we then continue the argument for $x \ldots u$ instead of $x \ldots y$.

Multiple paths $x \ldots y$ may not be entirely inside just one of the two subheaps $\mathfrak{f}$ or $\mathfrak{h}$ because that would violate our assumption that these are ZOPG subheaps. Therefore, *at least one of these paths must cross the border between* $\mathfrak{f}$ *and* $\mathfrak{h}$. We proceed with a case analysis based on the distribution of the nodes x and y between (disjoint) subheaps $\mathfrak{h}$ and $\mathfrak{f}$:

- The case $x, y \in \mathfrak{h}$ cannot be realized because a path starting in x may leave $\mathfrak{h}$ just once and *may never come back* to reach y (otherwise our convexity assumption would be violated).
- If $x \in \mathfrak{h}, y \in \mathfrak{f}$, then again, the paths starting in x may leave $\mathfrak{h}$ just once and may never come back due to $\mathfrak{h} \prec \mathfrak{g}$. Since two different paths starting in x may not merge in $\mathfrak{h}$ (otherwise, alternative paths will exist within $\mathfrak{h}$, contradicting our assumption that it is a ZOPG), such paths must reach *two different frame nodes* $a, b \in \mathfrak{f}$, creating alternative paths of the form $x \ldots a \ldots y$ and $x \ldots b \ldots y$, as covered by case (β) of Fig. 13.
- If $x \in \mathfrak{f}, y \in \mathfrak{h}$, then no path that starts in the frame node x can enter the footprint more than once due to $\mathfrak{h} \prec \mathfrak{g}$. Next, since in this case these paths must end in y, they cannot leave the

(relatively convex) footprint $\mathfrak{h}$ at all. Finally, these paths *may not merge until at least one of them enters* $\mathfrak{h}$ (otherwise, alternative paths will exist within $\mathfrak{f}$, contradicting our assumption that it is a ZOPG). This gives us *two different footprint nodes* $a, b \in \mathfrak{h}$, creating alternative paths of the form $x \ldots a \ldots y$ and $x \ldots b \ldots y$, as covered by case (α) of Fig. 13.

- In the most subtle case of $x, y \in \mathfrak{f}$, each pair of alternative paths of the form $x \ldots y$ is s.t. either *just one* or *each of the two* alternative paths *enters and exits* the footprint exactly once, as covered by cases (γ) and (δ) of Fig. 13, resp. □

The simplicity of our formulas (22), (23), (24), (25) is due to the fact that, in our technique, footprints of method calls must be relatively convex, limiting the number of bad configurations to just four. The bad configurations that we have identified are helpful for deriving weakest preconditions for method calls that operate over ZOPGs, like in our `testZopgObligations` example. In combination with local heap updates (for which (21) is the efficient ZOPG preservation criterion), we have explained how our technique is generalized for modular reasoning about ZOPGs.

5 EVALUATION

We have evaluated our technique on a variety of challenging example programs taken from the literature, illustrating our technique for different classes of graphs and data structures (including the running examples of closely-related work).

5.1 Experimental Setup

We encoded each example by-hand into the Viper verification language [Müller et al. 2016b]: an intermediate verification language designed for expressing heap-based verification problems, and with native support for separation logic reasoning. Although manual, our encoding of each example was performed methodically, simulating the translation that a front-end verification tool could perform. Each example consists of a common set of background definitions and axioms, along with a translation of the code of the example, statement by statement, according to the technique presented

Table 1. Experimental results. We indicate example features via ✓ where and denote examples with *greater-than-one outdegree* and *with sharing*, resp.; $\prec$ means *convex framing*.

Example	Variant	Class			$\prec$	Time	Notes
Merge (Fig. 1)	Tree	DAG	✓		✓	16.1	Path-partitioning,
	DAG	DAG	✓	✓	✓	14.5	Unbounded cut-points
	Fail 1	DAG	✓	✓	✓	13.2	Bug in code
	Fail 2	DAG	✓	✓	✓	33.9	Bug in spec.
Left-Child-Right-Sibling	Tree, add sibl.	DAG	✓		✓	10.5	Encodes n-ary tree as binary
	Tree, add child	DAG	✓		✓	15.0	— ” —
	DAG, add sibl.	DAG	✓	✓	✓	10.1	Unbounded cut-points
	DAG, add child	DAG	✓	✓	✓	17.1	— ” —
Harris List	Original	DAG		✓		14.5	From [Krishna et al. 2018]
Acyclic List	Reverse	DAG				7.9	From [Lev-Ami et al. 2009]
	Append	DAG				6.9	— ” —
Ring-Insert: Impl.	Sorted	ZOPG	✓	✓		87.2	Functional spec.
	Anywhere	ZOPG	✓	✓		10.1	— ” —
Ring-Insert: Client (Fig. 11)	Closed $\{u\}$	ZOPG	✓	✓	✓	11.5	Non-convex frame,
	Open $\{u\}$	ZOPG	✓	✓	✓	10.8	ZOPG obligations
	Fail 1	ZOPG	✓	✓	✓	12.4	Failure due to (β)
	Fail 2	ZOPG	✓	✓	✓	10.7	Failure due to (α), (γ)

in Sec. 3 and Sec. 4. For instance, a source-level method call is encoded with additional **`assume`** and **`assert`** statements before and after the call which enable reachability framing on relatively convex method footprints, as defined in Sec. 3.4.

The background definitions common to our examples are organized in separately-included library files, and we make heavy use of Viper's macros to improve the readability of our encoded examples. Our examples are verified with Viper's standard Boogie-based [Leino 2008] verifier, which uses the Z3 SMT solver [de Moura and Bjørner 2008] for checking verification conditions. We indicate Viper's run time for each example in Tab. 1. The experiments were performed on a laptop running macOS, with a 2.8 GHz Intel Core i7 CPU, with Z3 version 4.8.5 - 64 bit. The Viper files used in our experiments are available as artifact of this paper [Ter-Gabrielyan et al. 2019b].

An important practical issue arising in the successful use of SMT-based verification tools is controlling the instantiation of quantifiers; our technique employs a large number of quantified formulas, and we have carefully selected appropriate *triggers* [Barrett et al. 2017; de Moura and Bjørner 2008; Detlefs et al. 2005; Moskal 2009] for these, guided by the intended situations in which these formulas are relevant; for the rich reachability properties expressed by our technique, such triggers are essential for performance. Since our source-level specifications can also contain quantified formulas, we require these to be annotated with appropriate triggers (for simple cases, Viper can also infer appropriate choices if omitted).

5.2 Experiments

Tab. 1 gives an overview of our experiments. The "Merge" example is our first running example of Fig. 1, in variants with both tree and DAG structures for the underlying graphs (obtaining the DAG variant simply requires dropping the tree requirements throughout; no other changes are necessary). "Left-Child-Right-Sibling" is a technique for encoding trees with arbitrary multiplicities using only two fields (representing a list of children at each node), as employed in binomial heaps [Cormen et al. 2009], and recently proposed as a verification challenge [Müller 2018]. We again show a DAG variant (directly obtained by removing tree requirements), and verify adding sibling and child structures. As with the running example, these are non-functional graphs with (in the DAG case) sharing and requiring our convex framing to frame reachability across sub-calls; to our knowledge, they are beyond reach for all existing automated graph-verification techniques.

The "Fail" variants of Merge are buggy, with the bug being (1) negation of the branch condition in the body of `merge` and (2) missing `merge`'s last precondition. We have observed that the failure time does not diverge from the time of a successful verification attempt. This is important in practice if a program's implementation and specification are developed iteratively, with multiple invocations of the verifier.

Lev-Ami et al. verify reachability for linked-list reverse and append methods [Lev-Ami et al. 2009]; the recent Flows framework [Krishna et al. 2018] uses the Harris List as running example. In both cases, we prove the same invariants and reachability specifications, simply encoded in our language. In the latter case, we use two reachability relations based on different edges.

"Ring-Insert" is a series of six 0–1-path graph examples. We wrote two variants of the Ring-Insert method. "Sorted" is an implementation that traverses a sorted ring and inserts a newly allocated node into the right place. We can prove both reachability (the ring remains a ring) and sortedness; our connection to separation-logic reasoning makes layering additional functional specifications of this kind straightforward. "Anywhere" is the version discussed in Sec. 4, where the insertion happens at an arbitrary point in the ring. We also verified two types of clients of Ring-Insert. "Closed $\{u\}$" is the example of Fig. 11, where the attached node does not have outgoing paths, whereas "Open $\{u\}$" permits the attached node to be both reachable from the frame and have outgoing paths. The latter requires a more subtle precondition to satisfy the 0–1-path preservation

criteria. In the final two cases, we show that our technique allows us to automatically identify the type of bad configurations that may violate the 0–1-path invariant in cases where the heap is under-constraint before a method call Sec. 4.2.

5.3 Results

Our experiments show that reachability properties are amenable to SMT-based verification for a broad class of heap-manipulating programs. In particular, we have observed that our technique is well-suited for this task despite heavy usage of quantified formulas. While developing the specifications, we have experienced that our technique helps the programmer to better understand the subtle effects of heap operations on data structure invariants. Even with good tool support, writing consistent preconditions and postconditions requires particular craftsmanship, especially for recursive methods, like `merge`. Additionally, SMT-based verification with quantifiers requires the programmer to annotate the specifications with triggers.

6 RELATED WORK

Most work on separation logic focuses on data structures with limited sharing, with some notable exceptions. Iterated separating conjunction has been used to verify the Schorr-Waite graph marking algorithm [Yang 2001a], but without any tool support or automation. Recent work on Flows [Krishna et al. 2018] allows one to prove the *preservation* of a rich variety of graph invariants including reachability properties, but requires fixpoint computations that are hard to automate. Methods can operate on a subgraph; under the condition that *interfaces* [Krishna et al. 2018] of these subgraphs are preserved, a view on the client's graph can be reconstructed. They make no convexity restriction, but the interface preservation conditions rule out the possibility of method calls adding or removing paths between nodes in subgraph boundaries. By contrast, our reachability framing technique explicitly enables such side-effectful methods, and the reconstruction of appropriate changes in the client's footprint. For instance, in our running example of Fig. 2, new reachability relations are first *established* (by creating an edge from `link` to the root of `rdag`) and then *propagated* (by the enclosing method calls) to the larger context (the entirety of the client's footprint).

We adapted the precise transitive-closure update formulas from Dong and Su [1995] to program heaps and separation logic, rather than mathematical graphs. Their work also inspired our `DEP` relation; however, our version of the `DEP` relation is compatible with the *reflexive* reachability relation and is used only in the internal encoding, whereas theirs is exposed to programmers.

Table 2. Supported data structure categories. **"Itz."** denotes [Itzhaky et al. 2014]; **"Gr."** denotes GRASShopper [Piskac et al. 2014]; **"Nv."** indicates whether, to our knowledge, our technique enables automated verification of modularly specified reachability properties for the first time.

Data structure	Class	Itz.	Gr.	Nv.	Author
Arbitrary linked-list structures	ZOPG	✓			–
2-opt, 3-opt, etc.	ZOPG			✓	Croes [1958]; Lin [1965]
Hierarchical rings	ZOPG			✓	Fredman et al. [1986]
The priority inheritance protocol	ZOPG			✓	Sha et al. [1990]
Trees encoded via Java's LinkedList	ZOPG			✓	Sun, Oracle
Union-Find	ZOPG/DAG	✓			Tarjan [1975]
Trees	ZOPG/DAG		✓		–
Binary Decision Diagrams	DAG			✓	Akers Jr. [1978]; Lee [1959]
General DAGs (DFG, VCS, etc.)	DAG			✓	–

Reachability has been integrated into separation logic before (e.g., in GRASShopper [Piskac et al. 2014]), but only in a limited way that supports lists and trees but not heap structures with sharing.

Our work was inspired by Itzhaky et al. [2014, 2013]. Their verification technique allows one to prove reachability properties in various forms of list data structures. A focus of their work is to obtain decidable proof obligations. We sacrificed decidability in favor of supporting arbitrary acyclic graphs (with bounded out-degree) as well as 0–1-path graphs; our evaluation shows that we nevertheless achieve good automation. In contrast to Itzhaky et al., we integrated our work into separation logic, which allows us to verify concurrent programs and to reason about reachability and other properties in a uniform way. Moreover, we do not restrict method footprints in the number of entry and exit points or the number of SCCs in them. Tab. 2 summarizes the expressiveness of our technique and compares it with closely-related work.

7 CONCLUSIONS

We presented a specification and verification technique that allows one to reason about heap reachability properties modularly. The technique is integrated into separation logic and, thus, benefits immediately from the plurality of techniques and tools in this area. The key challenge of this integration is to specify reachability locally, within the footprint of a method. We solved this challenge by specifying reachability relatively to a given heap fragment and introducing a novel form of reachability framing to extend reachability properties in the footprint of a callee method to the larger footprint of the client. Even though reasoning about general reachability properties is difficult to automate, the proof obligations required by our technique are amenable to SMT solvers, which we demonstrate in our experiments.

As future work, we plan to extend our technique to graphs with unbounded outdegree. This can be done by using a generalized version of iterated separating conjunction [Müller et al. 2016a] that can specify permissions to sets of resources. Another direction of future work is to adapt our technique to separation logic with fractional permissions [Boyland 2003] to distinguish read and write access, especially in concurrent settings. We also plan to investigate the extent to which our approach to cyclic graphs is always precise. Finally, we are planning to implement a front-end verification tool that will simplify the process of writing modular reachability specifications.

ACKNOWLEDGMENTS

We are grateful to the anonymous referees for their thoughtful comments. We also thank Uri Juhasz, Marco Eilers, Gishor Sivanrupan, Jérôme Dohrau, Sviatlana-Maryia Zdobnikava, Alexandra Bugariu, Siddharth Krishna, Felix Wolf, Vytautas Astrauskas, Federico Poli, Martin Clochard, Shachar Itzhaky, and K. Rustan M. Leino for their help. This work was funded in part by the Swiss National Science Foundation under project 200021-156980.

REFERENCES

Sheldon B. Akers Jr. 1978. Binary Decision Diagrams. *IEEE Trans. Comput.* 27, 6 (1978), 509–516.

Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2017. *The SMT-LIB Standard: Version 2.6.* Technical Report. Department of Computer Science, The University of Iowa. Available at `www.SMT-LIB.org`.

John Tang Boyland. 2003. Checking Interference with Fractional Permissions. In *SAS (Lecture Notes in Computer Science)*, Vol. 2694. Springer, 55–72.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, 3rd Edition.* MIT Press.

G. A. Croes. 1958. A Method for Solving Traveling-Salesman Problems. *Operations Research* 6, 6 (1958), 791–812.

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (Lecture Notes in Computer Science)*, C. R. Ramakrishnan and Jakob Rehof (Eds.), Vol. 4963. Springer, 337–340.

David Detlefs, Greg Nelson, and James B. Saxe. 2005. Simplify: A Theorem Prover for Program Checking. *Journal of the ACM* 52, 3 (2005), 365–473.

Guozhu Dong and Jianwen Su. 1995. Incremental and Decremental Evaluation of Transitive Closure by First-Order Queries. *Inf. Comput.* 120 (1995), 101–106.

Michael L. Fredman, Robert Sedgewick, Daniel Dominic Sleator, and Robert E. Tarjan. 1986. The pairing heap: A new form of self-adjusting heap. *Algorithmica* 1 (1986), 111–129.

Shachar Itzhaky, Anindya Banerjee, Neil Immerman, Ori Lahav, Aleksandar Nanevski, and Mooly Sagiv. 2014. Modular Reasoning About Heap Paths via Effectively Propositional Formulas. In *POPL*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 385–396.

Shachar Itzhaky, Anindya Banerjee, Neil Immerman, Aleksandar Nanevski, and Mooly Sagiv. 2013. Effectively-Propositional Reasoning about Reachability in Linked Data Structures. In *CAV*, Natasha Sharygina and Helmut Veith (Eds.). Springer Berlin Heidelberg, 756–772.

Siddharth Krishna, Dennis E. Shasha, and Thomas Wies. 2018. Go with the flow: compositional abstractions for concurrent data structures. *PACMPL* 2, POPL (2018), 37:1–37:31.

C. Y. Lee. 1959. Representation of switching circuits by binary-decision programs. *The Bell System Technical Journal* 38, 4 (1959), 985–999.

K. Rustan M. Leino. 2008. This is Boogie 2. (2008). https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/

K. Rustan M. Leino and Rosemary Monahan. 2009. Reasoning about comprehensions with first-order SMT solvers. In *SAC*, Sung Y. Shin and Sascha Ossowski (Eds.). ACM, 615–622.

Tal Lev-Ami, Neil Immerman, Thomas W. Reps, Mooly Sagiv, Siddharth Srivastava, and Greta Yorsh. 2009. Simulating reachability using first-order logic with applications to verification of linked data structures. *Logical Methods in Computer Science* 5, 2 (2009).

Shen Lin. 1965. Computer solutions of the traveling salesman problem. *The Bell System Technical Journal* 44, 10 (1965), 2245–2269.

Michal Moskal. 2009. Programming with triggers. *ACM International Conference Proceeding Series* (2009).

Peter Müller. 2018. The Binomial Heap Verification Challenge in Viper. In *Principled Software Development*, Peter Müller and Ina Schaefer (Eds.). Springer-Verlag, 203–219.

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016a. Automatic Verification of Iterated Separating Conjunctions using Symbolic Execution. In *CAV (LNCS)*, Swarat Chaudhuri and Azadeh Farzan (Eds.), Vol. 9779. Springer-Verlag, 405–425.

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016b. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *VMCAI (LNCS)*, Barbara Jobstmann and K. Rustan M. Leino (Eds.), Vol. 9583. Springer-Verlag, 41–62.

Matthew J. Parkinson and Gavin M. Bierman. 2005. Separation logic and abstraction. In *POPL*, Jens Palsberg and Martín Abadi (Eds.). ACM, 247–258.

Matthew J. Parkinson and Alexander J. Summers. 2011. The Relationship between Separation Logic and Implicit Dynamic Frames. In *ESOP*, Gilles Barthe (Ed.). 439–458.

Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2014. GRASShopper - Complete Heap Verification with Mixed Specifications. In *TACAS (Lecture Notes in Computer Science)*, Erika Ábrahám and Klaus Havelund (Eds.), Vol. 8413. Springer, 124–139.

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *IEEE Symposium on Logic in Computer Science (LICS '02)*. IEEE Computer Society, 55–74.

Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. 1990. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Trans. Comput.* 39, 9 (1990), 1175–1185.

Jan Smans, Bart Jacobs, and Frank Piessens. 2012. Implicit Dynamic Frames. *ACM Transactions on Programming Languages and Systems* 34, 1, Article 2 (2012), 58 pages.

Robert Endre Tarjan. 1975. Efficiency of a Good But Not Linear Set Union Algorithm. *Journal of the ACM* 22, 2 (1975), 215–225.

Arshavir Ter-Gabrielyan, Alexander J. Summers, and Peter Müller. 2019a. *Modular Verification of Heap Reachability Properties in Separation Logic.* Technical Report. Department of Computer Science, ETH Zurich, Switzerland. arXiv:1908.05799

Arshavir Ter-Gabrielyan, Alexander J. Summers, and Peter Müller. 2019b. Modular Verification of Heap Reachability Properties in Separation Logic (Artifact). https://doi.org/10.5281/zenodo.3367478

Hongseok Yang. 2001a. An example of local reasoning in BI pointer logic: the Schorr-Waite graph marking algorithm. In *Proceedings of the SPACE Workshop*.

Hongseok Yang. 2001b. *Local Reasoning for Stateful Programs.* Ph.D. Dissertation. Advisor(s) Uday S. Reddy.