Automatic Data Structure Repair using Separation Logic

Guolong Zheng* Quang Loc Le† ThanhVu Nguyen* Quoc-Sang Phan‡

* University of Nebraska-Lincoln
{gzheng, tnguyen}@cse.unl.edu

† Teesside University
q.le@tees.ac.uk

‡ Fujitsu Labs. of America
sphan@us.fujitsu.com

ABSTRACT

Software systems are often shipped and deployed with both known and unknown bugs. On-the-fly program repairs, which handle runtime errors and allow programs to continue successfully, can help software reliability, e.g., by dealing with inconsistent or corrupted data without interrupting the running program.

We report on our work-in-progress that repairs data structure using separation logic. Our technique, inspired by existing works on specification-based repair, takes as input a specification written in a separation logic formula and a concrete data structure that fails that specification, and performs on-the-fly repair to make the data conforms with the specification. The use of separation logic allows us to compactly and precisely represent desired properties of data structures and use existing analyses in separation logic to detect and repair bugs in complex data structures.

We have developed a prototype, called STARFIX, to repair invalid Java data structures violating given specifications in separation logic. Preliminary results show that tool can efficiently detect and repair inconsistent data structures including lists and trees.

1. INTRODUCTION

Software systems are often shipped and deployed with (both known and unknown) bugs [9]. Many automatic program repair approaches [16, 21–23] focus on *offline* repairs, e.g., they analyze the program code, generate fixes, patch the program, and then recompile and run the patched program. In contrast, *on-the-fly* repairs [7–9, 24–26] repair a *running* program, allowing it to recover from errors and continue to run. Although offline repair, e.g., halt a failed program to repair, is ideal in many situations, the ability to handle and recover from runtime errors on-the-fly can improve software reliability considerably, e.g., in situations involving critical systems that cannot be interrupted or dealing with inconsistent/-corrupted files [8].

Specification-based repair is a popular on-the-fly repair approach focusing on data structures, such as lists or trees [8, 9, 24–26]. This approach takes as inputs a specification (e.g., a `repOK` predicate [9] or a first order logic formula) that encodes the required properties of data structures and a program state that fails that specification, and then creates a new program state to satisfy the specification. For example, given a predicate that checks for valid linked lists, this approach can automatically fix bad program states due to corrupted input lists (e.g., by changing some `next` field, which originally points to an incorrect node, to point to a correct node), thus allowing the program to continue its run correctly.

Our work is inspired by specification-based repair and also focuses on dynamically-allocated data structures (e.g., those created via the `new` keyword in Java or C and stored in the memory heap). However, instead of using predicates or formulas in first-order logic, we represent desired properties of data structures using heap predicates in *separation logic* (SL) [11, 20]. SL, which extends classical logic, allows for compact and precise representations of heap-based program semantics and reasoning to be localized to small portions of memory. There are several benefits of using SL specifications: (i) SL formulae are specifically designed to describe memory shape properties, which can be difficult to express using predicate or first-order logic formula, (ii) SL heap predicates naturally and succinctly encode the recursive structures of commonly-used and standard data structures (such as lists and trees), and (iii) we can use existing SL analyses such as model checking [4] and predicate unrolling [3, 17, 19] to check for bugs and perform on-the-fly repairs.

We present a new technique to repair data structures violating SL specifications. Given an inductive SL predicate and a concrete data structure input, we first check that the data conforms to the given predicate by iteratively unfolding and matching the predicate with the concrete data structures. Checking allows us to both detect bugs (indicated by unmatched results) and localizing faults (identifying unmatched parts of the data). Next, we analyze unmatched results to generate a candidate fix, e.g., computing a new value for an unmatched field, and recheck if the modified data satisfies the SL predicate. If it does not, we backtrack and compute a new fix. The algorithm can generate multiple valid fixes with respect to the given SL specification.

We have developed a repair prototype[1], called STARFIX, to repair invalid inputs violating given SL predicates. STARFIX is implemented in Java and is designed to check for inconsistent Java data structures. More specifically, STARFIX uses Java StarFinder [2], an extension of the Java PathFinder platform [1] that supports SL assertions. Currently, STARFIX can detect and repair corrupted data structures including lists and trees.

2. APPROACH AT A GLANCE

Given a specification in SL and a concrete data structure, STARFIX checks whether the data satisfies the predicate. If it is not the case, STARFIX modifies the data to satisfy the predicate. More specifically, STARFIX uses an iterative algorithm that unfolds and matches the heap predicate to the concrete data to find inconsistencies and repairs. STARFIX stops when it reaches a search limit (e.g., defined by the user) or has explored all possible matching attempts.

Heap Predicates. We use SL heap predicates to represent data structures. For example, the following predicates `dll` and `list` define circular doubly-linked lists:

$$\begin{aligned}&\texttt{pred dll}(head) \equiv (\texttt{emp} \wedge head{=}\texttt{null})\\ &\quad\vee\ (\exists p, n.head{\mapsto}\texttt{Node}(p,n) {*} \texttt{lst}(head, p, head, n))\\ &\texttt{pred lst}(h, prev_h, cur, next) \equiv (\texttt{emp} \wedge prev_h{=}cur{=}next{=}h)\\ &\quad\vee\ (\exists n, next{\mapsto}\texttt{Node}(cur, n) {*} \texttt{lst}(h, prev_h, next, n))\end{aligned}$$

Here, `emp` indicates that the heap is empty (e.g., the list is `null`) and $head{\mapsto}\texttt{Node}(p,n)$ indicates that $head$ points to an allocated `Node(p)` object. The existentially quantified variables `p` and `n` represent the previous

[1] `https://github.com/guolong-zheng/starfix`

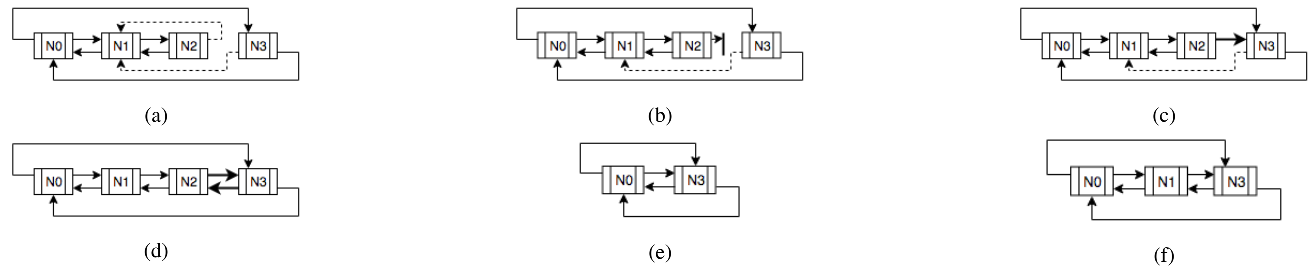

Figure 1: StarFix repairs the invalid circular doubly-linked list shown in Figure 1a by iteratively generating the repair candidates shown in Figures 1b, 1c, and 1d. Figure 1d is a correct repair. Figures 1e and 1f are two other valid repairs generated by StarFix.

and next fields of a `node` object, respectively. The operator $*$ is the separating conjunction in SL; intuitively `A` $*$ `B` holds when both `A` and `B` hold in disjoint heap regions.

The given predicate `lst` represents a circular doubly-linked list, recursively defined as a current node `cur` with its next field `next` pointing to a sublist. In `lst`, h is the head of the list, and $\texttt{prev}_\texttt{h}$ is the previous field of the head, pointing to the final node. Both `h` and $\texttt{prev}_\texttt{h}$ are free, i.e. unbounded, variables. In the base case, `next` is `null` and the list has only one node. In the recursive case, `next` points to a node, whose previous field points to the current node `cur`, and the next field `n` points to a sublist, recursively defined in the same way. The predicate `dll` is built on top `lst` to include the case when the list is empty.

Example. We illustrate StarFix using the example given in Figure 1, adopted from [9]. Figure 1a shows a data structure that *does not* satisfy `dll`, as illustrated by the *dashed* links: (i) the next field of `N2` is `N1` but the previous field of `N1` is not `N2` and (2) similarly, the previous field of `N3` is `N1` but the next field of `N1` is not `N3`. Figure 1 shows that to repair the data structure given in Figure 1a, StarFix generates several repairs attempts shown in Figures 1b and 1c and finally obtains the correct one in Figure 1d. In the following, we use the conventional terminologies *symbolic heap models* (or just *symbolic heaps*) for SL predicates and *concrete models* for concrete data structure instances.

StarFix uses an iterative, unrolling and matching algorithm to check the concrete model given in Figure 1a for inconsistency with the given symbolic heap represented `dll`. In the first iteration, StarFix unfolds $\texttt{dll}(x)$ to obtain the symbolic heaps:

$$\Delta_1 \equiv \texttt{emp} \wedge x{=}\texttt{null}$$
$$\Delta_2 \equiv \exists p_1, n_1.\ x{\mapsto}Node(p_1, n_1){*}\texttt{lst}(x, p_1, x, n_1)$$

StarFix then matches these symbolic models with the concrete model $\mathcal{M}_0 \equiv \{x{\mapsto}N0\}$ (for illustration purpose, we assume that x points to `N0` in Figure 1a). For Δ_1, the matching fails because `N0` is not null, and we do not continue with this model. For Δ_2, the match succeeds because we can find values from $\mathcal{M}_0$ to concreterize Δ_2, i.e., we can generate a new concrete model $\mathcal{M}_1 \equiv \{x{\mapsto}N0; p_1{\mapsto}N3; n_1{\mapsto}N1\}$. We continue the unfolding and matching process using Δ_2 and $\mathcal{M}_1$.

In the second iteration, we unfold Δ_2 and obtain the symbolic heaps:

$$\Delta_3 \equiv \exists p_1, n_1.\ x{\mapsto}Node(p_1, n_1){*}p_1{=}x \wedge x{=}n_1$$
$$\Delta_4 \equiv \exists p_1, n_1, n_2.\ x{\mapsto}Node(p_1, n_1){*}n_1{\mapsto}Node(x, n_2){*} \\ \texttt{lst}(x, p_1, n_1, n_2)$$

StarFix prunes Δ_3 due to failed match and matches Δ_4 to obtain the new concrete model $\mathcal{M}_2 \equiv \{x{\mapsto}N0; p_1{\mapsto}N3; n_1{\mapsto}N1; n_2{\mapsto}N2\}$.

Similarly, in the third iteration, we obtain a matched symbolic heap and create a concrete model (we do not show the unmatched heap):

$$\Delta_6 \equiv \exists p_1, n_1, n_2, n_3.\ x{\mapsto}Node(p_1, n_1){*}n_1{\mapsto}Node(x, n_2){*} \\ n_2{\mapsto}Node(n_1, n_3){*}\texttt{lst}(x, p_1, n_2, n_3)$$

$$\mathcal{M}_3 \equiv \{x{\mapsto}N0; p_1{\mapsto}N3; \underline{n_1{\mapsto}N1}; n_2{\mapsto}N2; \underline{n_3{\mapsto}N1}\}.$$

Next, in the fourth iteration, we unfold Δ_6 and obtain:

$$\Delta_7 \equiv \exists p_1, n_1, n_2, n_3.\ x{\mapsto}Node(p_1, n_1){*}n_1{\mapsto}Node(x, n_2){*} \\ n_2{\mapsto}Node(n_1, n_3){\wedge}p_1{=}n_2{\wedge}n_3{=}x$$
$$\Delta_8 \equiv \exists p_1, n_1, n_2, n_3, n_4.\ x{\mapsto}Node(p_1, n_1){*}n_1{\mapsto}Node(x, n_2){*} \\ n_2{\mapsto}Node(n_1, n_3){*}n_3{\mapsto}Node(n_2, n_4){*}\texttt{lst}(x, p_1, n_3, n_4)$$

The model Δ_7 does not match as $p_1{=}n_2{\wedge}n_3{=}x$ is unsatisfied under $\mathcal{M}_3$. The model Δ_8 also does not match because $\mathcal{M}_3$ requires $n_1 = n_3$ (underlined and in blue color pairs in $\mathcal{M}_3$) and Δ_8 requires n_1 is not equal to n_3 through the separating conjunction $*$. Because the concrete model does not match with any symbolic heaps, StarFix stops the unrolling process and concludes that the data structure given in Figure 1a does not satisfy `dll`.

StarFix now attempts to repair the data structures the concrete model given in Figure 1a. We leverage information computed during the unrolling process to reason that we can likely make the repair by modifying either $\texttt{n}_1$ or $\texttt{n}_3$. Each modification leads to a distinct fix.

We illustrate the repair by modifying the value of $\texttt{n}_3$. First, StarFix backtracks the unfolding process to the third iteration above where we obtain Δ_6 that creates $\texttt{n}_3$. Next, StarFix explores possible values for $\texttt{n}_3$, such as {`null`, N3, N0, N2}, respectively. StarFix first modifies the value of $\texttt{n}_3$ from N1 to `null` as shown in Figure 1b. Δ_7 and Δ_8 are both unsatisfied under this new model, so StarFix explore another possibility by modifying the value of $\texttt{n}_3$ from `null` to N3 as shown in 1c, where we obtain a new model:

$$\mathcal{M}'_3 \equiv \{x{\mapsto}N0; p_1{\mapsto}N3; n_1{\mapsto}N1; n_2{\mapsto}N2; n_3{\mapsto}N3\}.$$

With the new concrete model, StarFix unfolds Δ_6 and obtains Δ_7 and Δ_8 as shown above. Although Δ_7 is pruned as before, matching $n_3{\mapsto}Node(n_2, n_4)$ in Δ_8 returns a contradiction: $\mathcal{M}'_3$ requires $n_2{\mapsto}N2$, but the concrete model has $n_2{\mapsto}N1$. Thus, StarFix generates a fix by changing the value of the previous field in $N3$ in the concrete heap to `N2` as shown in Figure 1d.

With this fix, we can continue the unfolding process on Δ_8 to obtain:

$$\Delta_9 \equiv \exists p_1, n_1, n_2, n_3, n_4.\ x{\mapsto}Node(p_1, n_1){*}n_1{\mapsto}Node(x, n_2){*} \\ n_2{\mapsto}Node(n_1, n_3){*}n_3{\mapsto}Node(n_2, n_4){\wedge}p_1{=}n_3{\wedge}x{=}n_4$$
$$\Delta_{10} \equiv \ldots.$$

The symbolic model Δ_9 matches with the generated conrete model shown in Figure 1d. Moreover, we can no longer unfold Δ_9 because it contains no inductive predicate. This indicates that the repaired data structure satisfies the specification, i.e., a valid circular doubly-linked list.

Note that StarFix could generate multiple fixes. Other than the fix shown in 1d, StarFix also generates fixes as shown in Figures 1e and 1f, both are valid circular doubly linked list. The fix in Figure 1e modifies `next` field of $N0$ to $N3$ and `prev` field of $N3$ to $N0$. The fix in Figure 1f

Predicate defn	*Pred* $::=$ $\mathtt{pred}\ \mathtt{P_i}(\bar{v}_i)\equiv\Phi_i;\quad \tau ::= \mathtt{Int} \mid c$
Data structure	*Node* $::=$ $\mathtt{data}\ c_i\{\tau_1\ f_{i_1};..;\tau_j\ f_{i_j}\}$
Symbolic heap	$\Phi ::= \Delta \mid \Phi_1 \vee \Phi_2 \qquad \Delta ::= \exists\bar{v}.\ (\kappa\wedge\pi)$
Spatial formula	$\kappa ::= \mathtt{emp} \mid x{\mapsto}c(v_i) \mid \mathtt{P}(\bar{v}) \mid \kappa_1{*}\kappa_2$
Pure formula	$\pi ::= \mathtt{true} \mid \alpha \mid \neg\pi_1 \mid \exists v.\ \pi \mid \pi_1{\wedge}\pi_2 \mid \pi_1{\vee}\pi_2$
Arithmetic	$\alpha ::= a_1{=}a_2 \mid a_1{\leq}a_2$
	$a ::= k \mid v \mid k{\times}a \mid a_1{+}a_2 \mid -a$

Figure 2: Grammar of separation logic formulas

modifies `next` field of $N1$ to $N3$ and `prev` field of $N3$ to $N1$.

3. BACKGROUND

We briefly describe SL formulae and the `Java StarFinder` tool, in which our STARFIX is built upon.

3.1 Separation logic

In the last two decades, separation logic formalism [11, 20] has been successfully applied to analyzing and verifying heap-manipulating programs. This formalism can support the concise and precise abstraction of shapely data structures via symbolic heaps. The syntax of symbolic heaps in this work is presented in Figure 2. To support type-based heap semantics (like Java), we define concrete heap models via a fixed finite collection *Node* (using keyword `data`), a fixed finite collection *Fields*, a disjoint set *Loc* of locations (heap addresses), a set of non-address values *Val*, such that $\mathtt{null}{\in}Val$ and $Val{\cap}Loc{=}\emptyset$ (i.e., no pointer arithmetic). We assume a set of integers $\mathbb{Z}$ and $\mathbb{Z}{\subseteq}Val$. We use k to denote an integer constant.

In Figure 2, we use $\bar{x}$ to denote a sequence of variables. A formula is a disjunction of symbolic heaps. A symbolic heap is a universally quantified conjunction of a spatial formula and a pure (non-heap) formula. A spatial formula is a *separating* conjunction of empty predicates `emp`, points-to predicates $x{\mapsto}c(\bar{v})$ (where $\mathtt{c} \in$ *Node*and $\bar{v}$ are variables corresponding to fields of `c`), occurrences of inductive predicates $\mathtt{P}(\bar{v})$. In STARFIX, a predicate, like [15] (and in contrast to Facebook's Infer [5]), is defined by the user using symbolic heaps with keyword `pred`. A pure formula is an arithmetical constraint.

3.2 StarLib

In [17, 18], Pham *et al.* introduced STARLIB, the main component for SL functions in JAVA STARFINDER. This library provides JAVA API's for parsing SL formula, unfolding inductive predicate, dispatching satisfiability, etc. We note that the satisfiability API in STARLIB actually invokes the state-of-the-art SL solver presented [13, 15] to discharge the satisfiability problems. In this work, we extend STARLIB with matching function to instantiate existentially quantified variables.

STARFIX uses the popular `Unfold−and−Match` technique to deal with inductive predicates (which represent unbounded data structures) in SL. This technique has been used for entailment [6], symbolic execution [3, 17, 19], satisfiability [15], and frame inference [14]. Intuitively, given a formula with inductive predicates, this technique helps to expose all possible heap structures through unfolding and instantiate the quantified heap-based variables through matching. For model checking problem, this technique first helps instantiate heap structure of a formula and then match this structure against a given concrete heap structure. To our best knowledge, we are the first to apply `Unfold−and−Match` to the model checking and repair problems.

4. AUTOMATIC DATA STRUCTURE REPAIR

Given a specification Δ and a concrete data structure T, STARFIX checks if T conforms with Δ. If not, STARFIX generates one or multiple modified versions of T that conform with Δ.

4.1 Bug detection

In essence, the specification Δ describes the set Σ of valid *symbolic* heap configurations. On the other hand, the data structure T is a *concrete* heap configuration. T contains no bug if there exists $\sigma \in \Sigma$ that can be concretized to T. This is a model checking problem. Algorithm 1 describes

Algorithm 1: ModelChecking(Δ, T)

```
stack ← ∅
stack.push(Δ)
while stack.empty() = false do
    Δt ← stack.pop()
    if match(Δt, T) = true then
        return true
    if match(Δt, T) = false then
        continue
    S ← unfold(Δt)
    for Δi ∈ S do
        stack.push(Δi)
return false
```

how STARFIX performs a depth-first search for such a σ. Similar to SAT solver, it iteratively builds a map $\mathcal{M}$, of which each entry is a pair $\langle o_s,\ o_c\rangle$ of a symbolic heap object o_s and concrete heap object o_c. The concretization is the process of assigning the concrete objects, i.e. values, in T to the symbolic objects in (a derivation of) Δ.

In the beginning, Δ is pushed on the stack, and $\mathcal{M}$ maps the root states of the symbolic and concrete heaps, for instance, $\mathcal{M}$ stores $\langle \mathtt{head},\ \mathtt{N_0}\rangle$ in our illustrative example. STARFIX then iteratively does the following procedure. First, it takes out the current symbolic heap configuration Δ_t from the top of the stack. If Δ_t can be concretized to T, the T conforms to Δ, i.e. it contains no bug, and the search ends.

On the other hand, if it determines that Δ_t cannot be concretized to T, it continues the search for different paths (we use the `continue` instruction as in Java or C/C++). If matching is not decidable at the current state as Δ_t contains inductive predicates, these predicates need to be unfolded.

Since an inductive predicate is often a disjunction of several symbolic heaps (e.g., one base case and several recursive cases), unfolding Δ_t will result in a set $\mathcal{S}$, and each element of $\mathcal{S}$ is pushed on the stack for further searching. Details about the subroutines are explained in the following.

4.1.1 $\mathtt{match}(\Delta_t, \mathrm{T})$

This function takes two inputs: the symbolic heap Δ_t and the concrete heap T, and it has three possible outputs:

- `true`: Δ_t can be concretized to T using $\mathcal{M}$. This means Δ_t does not contain inductive predicates, and all heap variables are contained in $\mathcal{M}$. For example, Δ_9 with assignment $\mathcal{M}'_3$ in our illustrative example.
- `false`: Δ_t cannot be concretized to T. This is caused by a conflict in $\mathcal{M}$. For example, the conflict of n_3 and n_1 in assignment $\mathcal{M}_3$ in the illustrative example.
- `unknown`: neither `true` nor `false`. There is no conflict in the partial map $\mathcal{M}$, but there are variables of Δ_t which are not in $\mathcal{M}$ as they are defined by inductive predicates. These variables also get concretized, as variable p_1 and n_1 in Δ_2 in the illustrative example.

In the beginning, $\mathcal{M}$ contains only the root states of the symbolic and concrete heaps. `match` then updates $\mathcal{M}$ by comparing the shapes of the symbolic and concrete heaps.

4.1.2 `unfold(Δt)`

This function is part of the StarLib library [18]. Its input is a formula Δ_t, which contains a variable `x` (or a set of variables) being defined by an inductive predicate. Therefore, it is necessary to unfold the predicate to capture the resources accessed by `x`, a.k.a. the *footprints* of `x`. The unfolding procedure includes the following two sub-procedures: (i) replace one occurrence of inductive predicates by its definition ; (ii) rename all existentially quantified variables to avoid clashing. The result of this procedure is a new formula with `x` typically being defined by a point-to predicate, while the newly renamed variables may still be defined by inductive predicates. As the predicate is often defined by a disjunction of several symbolic heaps, the output of the function is a set $\mathcal{S}$ of new formulas.

4.2 Automatic repair

When StarFix determines that T does not conform with Δ, it will generate one or multiple modified versions of T that conforms with Δ. This procedure is defined in Algorithm 2. This algorithm shares many common subroutines with Algorithm 1, the main difference is that `match` will never return `true`, and when `match` returns `false`, StarFix will attempt to generate a fix (or fixes), and then continues its search.

Algorithm 2: Repair(Δ, T)

```
stack ← ∅
Γ ← ∅
stack.push(Δ)
while stack.empty() = false do
    Δt ← stack.pop()
    if match(Δt, T) = false then
        γ ← generateFix(Δt, T)
        Γ ← Γ ∪ {γ}
    S ← unfold(Δt)
    for Δi ∈ S do
        stack.push(Δi)
return Γ
```

All functions in Algo. 2 have already been explained in the previous section. We now explain how `generateFix` mutates T to generate fixes that conform with Δ. When $\mathtt{match}(\Delta_t, \mathrm{T})$ returns `false`, there are typically two types of conflict in $\mathcal{M}$.

- $\exists\, o_s^1, o_s^2, o_c.\langle o_s^1, o_c\rangle, \langle o_s^2, o_c\rangle \in \mathcal{M} \wedge o_s^1 \neq o_s^2$
- $\exists\, o_s, o_c^1, o_c^2.\langle o_s, o_c^1\rangle \in \mathcal{M}$ but o_s should map to o_c^2 in the concrete heap. For example, the reason explained to make the fix in 1d.

The first type of conflict only occurs when o_s^1 and o_s^2 are existentially quantified variables. For example, the conflict of n_3 and n_1 in assignment $\mathcal{M}_3$ in the illustrative example. StarFix can generate two possible fixes by modifying either o_s^1 or o_s^2. In order to do so, it needs to backtrack to the state where o_s^1 (or o_s^2) is instantiated and generates a new corresponding concrete heap object o_c^1 for o_s^1. To generate o_c^1, StarFix considers the following options: (i) `null`, (ii) initialized objects in the concrete heap, and (iii) a new node.

For the second type of conflict, StarFix generates only one possible fix by modifying o_c in the concrete heap in a way that it can be concretized form o_s. We perform the similar search on possible values for o_c in the similar way.

After fixing, there may be the cases that all symbolic objects are in the map $\mathcal{M}$, while some concrete objects are not in $\mathcal{M}$. In these cases, those concrete objects will be deleted, as they are not bounded by the specification. The fixes are still valid.

5. PRELIMINARY RESULTS

StarFix is implemented in Java and works with Java bytecode programs. We use Java reflection to collect concrete heaps from running program and StarLib to parse and unfold SL formulae. Currently, we implement a simple model checker to check the concrete heap with respect to the specification as described in Algorithm 1. In future work, we would extend the solver, e.g. [15], to obtain a more powerful model checker to support more expressive properties involving general inductive definitions, pointer-based (dis)equalities and arithmetic.

To evaluate StarFix, we manually create corrupted data structures by injecting errors into a correct data structure. For example, we randomly pick nodes in a list and change their `next` or `prev` fields to point to `null` or some random nodes in the list.

In addition to lists (e.g., the doubly-linked list shown in Section 2), StarFix can repair tree data structures, e.g., a binary tree defined by the predicate:

$$\begin{aligned}&\mathtt{pred}\ \mathtt{tree}(root) \equiv (\mathtt{emp} \wedge root{=}\mathtt{null})\\ &\quad\vee\ (\exists l, r.root{\mapsto}\mathtt{Node2}(l, r){*}tree(l){*}tree(r))\end{aligned}$$

We can then use the predicate `tree` to repairs tree data structures. For example, Figure 3a shows a corrupted tree that has both the `right` field of `b` and `left` field of `e` pointing to `g`, violating the separating conjunction in `tree`. When given this tree, StarFix detects the inconsistency (at iteration 20) and generates two fixes (at iteration 89): changing the `right` field of `b` to `null`) (3b) and changing the `left` field of `e` to `null` 3c. Both fixes are valid with respect to the `tree` specification.

6. RELATED WORK

StarFix is inspired by the line of research on specification-based repair for data structures. In this line of work, Elkarablieh et al. [9] use `repOK` predicate functions to check for data structure integrity and mutates invalid data structures to pass these predicates. In addition to `repOK`, Zaeem et al. [24–26] support pre/post-conditions using relational first-order logic formulae in the Alloy's language [12] and use SAT solving to generate repairs. Demsky and Rinard [8] also use specifications written in Alloy's language and perform repairs by translating constraints into disjunctive normal form and solving using an ad hoc search.

Several works explore strategies to improve efficiency and effectiveness of data structure repairs using `repOK` and Alloy specifications, including using execution trace history and unsat core [24] and abstracting and memorizing repairs to apply to similar repairs [26]. Demsky et. al [7] integrates the Daikon dynamic invariant generation [10] to infer desired specifications for data structures (instead of requiring users to manually provide such specifications). We are exploring these ideas to adapt and extend them to StarFix.

The work in [4] presents a model checking technique for general SL inductive predicates to support runtime verification. Given a concrete model and a specification, the checker relies on a fixed point calculator to compute for the specification a finite set of base pairs each of which is a sub-model of the concrete model. In contrast to [4], StarFix is based on `Unfold−and−Match` and focuses on repair, instead of just checking, programs. When the model checker returns `false`, StarFix generates fixes to avoid interrupting the running programs.

7. CONCLUSIONS AND FUTURE WORK

We present our work-in-progress on data structure repair using separation logic. By using separation logic, we can compactly and precisely capture desired properties of data structures and use existing techniques in separation logic to detect and repair complex data structures.

Currently, our prototype StarFix can fix inductive Java data structures

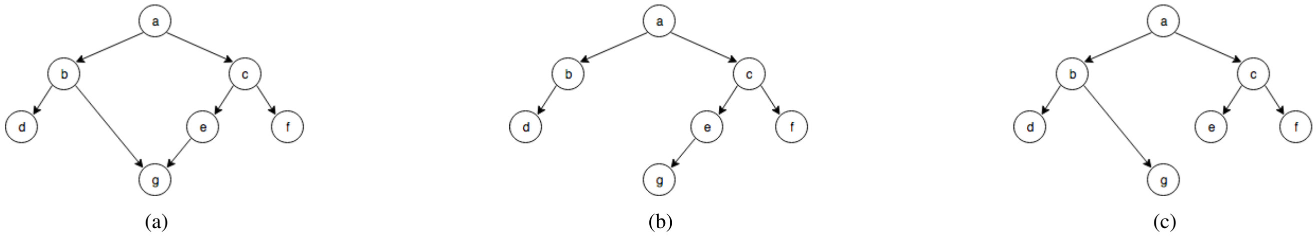

Figure 3: STARFIX fixes the invalid tree shown in Figure 3a by generating two repairs shown in Figures 3b and 3c.

such as trees and lists. We are pursuing four areas to improve the work. First, we would extend an SL satisfiability solver, like [15], to support for a more expressive fragment with general inductive definitions and arithmetic. Secondly, we are evaluating ranking techniques to prioritize fixes generated by STARFIX, e.g., preferring repairs that preserve the original data as much as possible. Thirdly, we are exploring optimizations, e.g., repair abstraction and history-aware strategy [24, 26], to improve STARFIX's performance. Lastly, we are interested in using dynamic inference to automatically generate required separation logic specifications from good program states, as has been proposed in [7].

Acknowledgements. This work is partially supported by the Google Summer of Code 2018 program.

8. REFERENCES

[1] Java PathFinder. `http://babelfish.arc.nasa.gov/trac/jpf/`.

[2] Java StarFinder. `https://github.com/star-finder/jpf-star`.

[3] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic Execution with Separation Logic. In *APLAS*, volume 3780, pages 52–68, Nov. 2005.

[4] J. Brotherston, N. Gorogiannis, M. Kanovich, and R. Rowe. Model Checking for Symbolic-heap Separation Logic with Inductive Predicates. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 84–96, New York, NY, USA, 2016. ACM.

[5] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. Moving Fast with Software Verification. In *NASA Formal Methods*, pages 3–11, Cham, 2015. Springer International Publishing.

[6] W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Automated Verification of Shape, Size and Bag Properties via User-defined Predicates in Separation Logic. *Sci. Comput. Program.*, 77(9):1006–1036, Aug. 2012.

[7] B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. Rinard. Inference and enforcement of data structure consistency specifications. In *ISSTA*, pages 233–244. ACM, 2006.

[8] B. Demsky and M. Rinard. Automatic detection and repair of errors in data structures. In *OOPSLA*, volume 38, pages 78–95. ACM, 2003.

[9] B. Elkarablieh, I. Garcia, Y. L. Suen, and S. Khurshid. Assertion-based Repair of Complex Data Structures. In *ASE*, pages 64–73, New York, NY, USA, 2007. ACM.

[10] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, pages 35–45, 2007.

[11] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, pages 14–26, New York, NY, USA, 2001. ACM.

[12] D. Jackson. Alloy: a lightweight object modelling notation. *TOSEM*, 11(2):256–290, 2002.

[13] Q. L. Le, J. Sun, and W.-N. Chin. *Satisfiability Modulo Heap-Based Programs*, pages 382–404. Springer International Publishing, Cham, 2016.

[14] Q. L. Le, J. Sun, and S. Qin. Frame Inference for Inductive Entailment Proofs in Separation Logic. In *TACAS*, pages 41–60, Cham, 2018. Springer International Publishing.

[15] Q. L. Le, M. Tatsuta, J. Sun, and W.-N. Chin. A Decidable Fragment in Separation Logic with Inductive Predicates and Arithmetic. In *CAV*, pages 495–517, 2017.

[16] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *ICSE*, pages 691–701. ACM, 2016.

[17] L. H. Pham, Q. L. Le, Q.-S. Phan, J. Sun, and S. Qin. Enhancing Symbolic Execution of Heap-based Programs with Separation Logic for Test Input Generation. *CoRR*, abs/1712.06025, 2017.

[18] L. H. Pham, Q. L. Le, Q.-S. Phan, J. Sun, and S. Qin. Testing Heap-based Programs with Java StarFinder. In *ICSE*, pages 268–269, New York, NY, USA, 2018. ACM.

[19] X. Qiu, P. Garg, A. Ştefănescu, and P. Madhusudan. Natural Proofs for Structure, Data, and Separation. In *PLDI*, pages 231–242, New York, NY, USA, 2013. ACM.

[20] J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*, pages 55–74, 2002.

[21] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad. ELIXIR: Effective Object Oriented Program Repair. In *ASE*, pages 648–659. IEEE, 2017.

[22] R. van Tonder and C. L. Goues. Static Automated Program Repair for Heap Properties. In *ICSE*, pages 151–162. ACM, 2018.

[23] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically Finding Patches Using Genetic Programming. In *ICSE*, pages 364–367. IEEE, 2009.

[24] R. N. Zaeem, D. Gopinath, S. Khurshid, and K. S. McKinley. History-aware data structure repair using SAT. In *TACAS*, pages 2–17. Springer, 2012.

[25] R. N. Zaeem and S. Khurshid. Contract-based data structure repair using Alloy. In *ECOOP*, pages 577–598. Springer, 2010.

[26] R. N. Zaeem, M. Z. Malik, and S. Khurshid. Repair abstractions for more efficient data structure repair. In *Runtime Verification*, pages 235–250. Springer, 2013.