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.

Categories and Subject Descriptors
D.2.0 [Software Engineering]: [General]; D.2.1 [Software Engineering]: [Requirements/Specifications]; D.2.4 [Software Engineering]: [Software/Program Verification]; D.2.5 [Software Engineering]: [Testing and Debugging]; F.3.1 [Logics and Meanings of Programs]: [Specifying and Verifying and Reasoning about Programs]

Keywords
automatic repair; error handling and recovery; data structure; separation logic

1. INTRODUCTION
Software systems are often shipped and deployed with both known and unknown bugs [9]. Many automatic program repair approaches [16, 17, 22, 24] 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, 25, 27] 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 [5, 9, 25, 27]. This approach takes as inputs a specification (e.g., a rep0rk 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 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, 21]. 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, 18, 20] 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, 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 specification. If it is not the case, STARFix modifies the data to satisfy the specification. More specifically, STARFix uses an iterative algorithm that unfolds and matches heap predicates of the specification 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.

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 1a is a correct repair. Figures 1e and 1f are two other valid repairs generated by STARFIX.

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

\[
\begin{align*}
\text{pred } dll & (head) \equiv (\text{emp} \land \text{head} = \text{null}) \\
& \lor (\exists p, n. \text{head} = \text{Node}(p, n) \land \text{lst}(\text{head}, p, n)) \\
\text{pred } lst & (h, \text{prev}, \text{cur}, \text{next}) \equiv (\text{emp} \lor \text{prev} = \text{cur} = \text{next} = h) \\
& \lor (\exists n. \text{next} = \text{Node}(\text{cur}, n) \land \text{lst}(h, \text{prev}, \text{next}, n))
\end{align*}
\]

Here, `emp` indicates that the heap is empty (e.g., the list is null) and `head = Node(p, n)` indicates that `head` points to an allocated `Node(p)` object. The existentially quantified variables `p` and `n` represent the previous 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 pointing to a sublist. In `lst`, `h` is the head of the list, and `prev` is the previous field of the head, pointing to the final node. Both `h` and `prev` 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 adapted 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 `H1` but the previous field of `H1` is not `N2` and (2) similarly, the previous field of `N3` is `H1` but the next field of `N1` is `N2`. Figure 1b shows that to repair the data structure given in Figure 1a, STARFIX generates several repairs attempts shown in Figure 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 1b for inconsistency with the given symbolic heap represented `dll`. In the first iteration, STARFIX unfolds `dll(x)` to obtain the symbolic heaps:

\[
\begin{align*}
\Delta_1 & \equiv \text{emp} \land x = \text{null} \\
\Delta_2 & \equiv \exists p_1, n_1. x \rightarrow \text{Node}(p_1, n_1) \land \text{lst}(x, p_1, x, n_1)
\end{align*}
\]

STARFIX then matches these symbolic models with the concrete model `M0` (for illustration purpose, we assume that `x` points to `N0` in Figure 1a). For `\Delta_1`, the matching fails because `N0` is not null, and we do not continue with this model. For `\Delta_2`, the match succeeds because we can find values from `M0` to concretize `\Delta_2`, i.e., we can generate a new concrete model `M1`.

In the second iteration, we unfold `\Delta_2` and obtain the symbolic heaps:

\[
\begin{align*}
\Delta_2 & \equiv \exists p_1, n_1. x \rightarrow \text{Node}(p_1, n_1) \land x = n_1 \\
\Delta_4 & \equiv \exists p_1, n_1, n_2. x \rightarrow \text{Node}(p_1, n_1) \land n_1 \rightarrow \text{Node}(x, n_2) \land \text{lst}(x, p_1, n_1, n_2)
\end{align*}
\]

STARFIX prunes `\Delta_3` due to failed match and matches `\Delta_4` to obtain the new concrete model `M2`:

\[
M_2 \equiv \{x \rightarrow N0; p_1 \rightarrow N3; n_1 \rightarrow N1; n_2 \rightarrow N2\}
\]

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

\[
\begin{align*}
\Delta_5 & \equiv \exists p_1, n_1, n_2, n_3. x \rightarrow \text{Node}(p_1, n_1) \land n_1 \rightarrow \text{Node}(x, n_2) \land n_2 \rightarrow \text{Node}(n_1, n_3) \land \text{lst}(x, p_1, n_2, n_3) \\
\Delta_3 & \equiv \exists p_1, n_1, n_2, n_3. x \rightarrow \text{Node}(p_1, n_1) \land n_1 \rightarrow \text{Node}(x, n_2) \land n_2 \rightarrow \text{Node}(n_1, n_3) \land \text{lst}(x, p_1, n_2, n_3)
\end{align*}
\]

The model `\Delta_4` does not match as `n_1 = n_2 \land n_3 = x` is unsatisfied under `M3`. The model `\Delta_5` also does not match because `M3` requires `n_1 = n_2` (underlined and in blue color pairs in `M3`) and `\Delta_5` requires `n_1` is not equal to `n_3` through the separating conjunction `\land`. 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 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 `n_1` or `n_3`. Each modification leads to a distinct fix.

We illustrate the repair by modifying the value of `n_3`. First, STARFIX backtracks the unrolling process to the third iteration above where we obtain `\Delta_3` that creates `n_3`. Next, STARFIX explores possible values for `n_3`, such as `[null, N3, N0, N2]`, respectively. STARFIX first modifies the value of `n_3` from `N1` to `null` as shown in Figure 1b `\Delta_7` and `\Delta_8` are both unsatisfied under this new model, so STARFIX explores another possibility by modifying the value of `n_3` from `null` to `N3` as shown in 1c where we obtain a new model:

\[
M_3' \equiv \{x \rightarrow N0; p_1 \rightarrow N3; n_1 \rightarrow N1; n_2 \rightarrow N2; n_3 \rightarrow N3\}
\]

With the new concrete model, STARFIX unfolds `\Delta_0` and obtains `\Delta_7` and `\Delta_8` as shown above. Although `\Delta_7` is pruned as before, matching `n_3 \rightarrow \text{Node}(n_2, n_1)` in `\Delta_8` returns a contradiction: `M_4'` requires `n_2 \rightarrow N2`, but the concrete model has `n_2 \rightarrow N1`. Thus, STARFIX generates a fix by changing the value of the previous field in `N3` in the concrete heap to `null` as shown in Figure 1b.
With this fix, we can continue the unfolding process on $\Delta_3$ to obtain:

$$\Delta_3 \equiv \exists p_1, n_2, n_3, n_4. \ x \to \text{Node}(p_1, n_1) \land \text{Node}(x, n_2) \land \text{Node}(n_1, n_3) \land \text{Node}(n_2, n_4) \land p_1 = n_3 \land x = n_4$$

$\Delta_{10} \equiv \ldots$

The symbolic model $\Delta_3$ matches with the generated concrete model shown in Figure[13]. Moreover, we can no longer unfold $\Delta_3$ 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[11], STARFix also generates fixes as shown in Figures[14] and [15]. All four cases are valid circular doubly linked lists. The fix in Figure[14] modifies the next field of N0 to N3 and prev field of N3 to N0. The fix in Figure[15] modifies the 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][21] 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 $\text{Node}$ (using keyword data), a fixed finite collection $\text{Fields}$, a disjoint set Loc of locations (heap addresses), a set of non-address values Val, such that null $\in$ Val and Val $\cap$ Loc $=$ $\emptyset$ (i.e., no pointer arithmetic). We assume a set of integers $\mathbb{Z}$ and $\mathbb{Z} \subseteq \text{Val}$. 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 predicate $\exists v \in \text{Node}$ (where $\in \text{Node}$ and $v$ are variables corresponding to fields of $\text{Node}$) occurrences of inductive predicates $P(\bar{v})$.

STARFix, a predicate, like [15] (and in contrast to Facebook’s Infer[3]), is defined by the user using symbolic heaps with keyword $\text{pred}$. A pure formula is an arithmetical constraint.

3.2 StarLib

In [18][19], 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 [12][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][19][20], 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 $\Delta$ and a concrete data structure $T$, STARFix checks if $T$ conforms with $\Delta$. If not, STARFix generates one or multiple modified versions of $T$ that conform with $\Delta$.

4.1 Bug detection

In essence, the specification $\Delta$ describes the set $\Sigma$ 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[11] describes how STARFix performs a depth-first search for such a $\sigma$. Similar to SAT solver, it iteratively builds a map $M$, of which each entry is a pair $(o_\text{h}, o_\text{n})$ of a symbolic heap object $o_\text{h}$ and concrete heap object $o_\text{n}$.

In the beginning, $\Delta$ is pushed on the stack, and $M$ maps the root states of the symbolic and concrete heaps, for instance, $M$ stores (head, $\emptyset$) in our illustrative example. STARFix then iteratively does the following procedure. First, it takes out the current symbolic heap configuration $\Delta_t$ from the top of the stack. If $\Delta_t$ can be concretized to $T$, the $T$ conforms to $\Delta$, i.e., it contains no bug, and the search ends.

On the other hand, if it determines that $\Delta_t$ 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 $\Delta_t$ 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 $\Delta_t$ will result in a set $S$, and each element of $S$ is pushed on the stack for further searching. Details about the subroutines are explained in the following.

4.1.1 match($\Delta_t$, $T$)

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

- true: $\Delta_t$ can be concretized to $T$ using $M$. This means $\Delta_t$ does not contain inductive predicates, and all heap variables are con-
tained in \( M \). For example, \( \Delta_0 \) with assignment \( M_3 \) in our illustrative example.

- **false**: \( \Delta_t \) cannot be concretized to \( T \). This is caused by a conflict in \( M \). For example, the conflict of \( n_3 \) and \( n_1 \) in assignment \( M_3 \) in the illustrative example.

- **unknown**: neither true nor false. There is no conflict in the partial map \( M \), but there are variables of \( \Delta_t \) which are not in \( M \) as they are defined by inductive predicates. These variables also get concretized, as variable \( p_1 \) and \( n_1 \) in \( \Delta_3 \) in the illustrative example.

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

### 4.1.2 \( \text{unfold}(\Delta_t) \)

This function is part of the STARLIB library \[19\]. Its input is a formula \( \Delta_t \), 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 \( \gamma \); (ii) rename all existentially quantified variables to avoid clashing. The result of this procedure is a new formula with variables 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 \( S \) of new formulas.

### 4.2 Automatic repair

When STARFIX determines that \( T \) does not conform with \( \Delta_t \), it will generate one or multiple modified versions of \( T \) that conforms with \( \Delta_t \). This procedure is defined in Algorithm \[13\]. This algorithm shares many common subroutines with Algorithm \[1\] the main difference is that \( \text{match} \) will never return true, and when \( \text{match} \) returns false, STARFIX will attempt to generate a fix (or fixes), and then continues its search.

#### Algorithm 2: Repair(\( \Delta_t, T \))

```plaintext
stack ← ∅
\( \Gamma \) ← ∅
```

```
stack.push(\( \Delta_t \))
while stack.empty() = false do
    \( \Delta_t \leftarrow \text{stack.pop()} \)
    if match(\( \Delta_t, T \)) = false then
        \( \gamma \leftarrow \text{generateFix(\( \Delta_t, T \))} \)
        \( \Gamma \leftarrow \Gamma \cup \{ \gamma \} \)
        \( S \leftarrow \text{unfold}(\Delta_t) \)
    for \( \Delta_t \in S \) do
        stack.push(\( \Delta_t \))
return \( \Gamma \)
```

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

- \( \exists o_1^1, o_2^1, o_3^1.(o_1^1, o_2^1, o_3^1) \in M \land o_1^1 \neq o_2^1 \)
- \( \exists o_1^2, o_2^2,(o_1^2, o_2^2) \in M \land o_1^2 \) should map to \( o_2^1 \) in the concrete heap. For example, the reason explained to make the fix in \[1d\].

The first type of conflict only occurs when \( o_1^1 \) and \( o_1^2 \) are existentially quantified variables. For example, the conflict of \( n_3 \) and \( n_1 \) in assignment \( M_3 \) in the illustrative example. STARFIX can generate two possible fixes by modifying either \( o_1^1 \) or \( o_2^1 \). In order to do so, it needs to backtrack to the state where \( o_1^1 \) (or \( o_2^1 \)) is instantiated and generates a new corresponding concrete heap object \( o_1^1 \) for \( o_1^1 \). To generate \( o_1^2 \), STARFIX considers the following options: (i) \text{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 \( o_1 \) in the concrete heap in a way that it can be concretized form \( o_1 \). We perform the similar search on possible values for \( o_1 \) in a similar way.

After fixing, there may be the cases that all symbolic objects are in the map \( M \), while some concrete objects are not in \( M \). 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:

\[
\text{pred tree}(\text{root}) \equiv (\text{emp} \land \text{root} = \text{null}) \lor (\exists l, r. \text{root} \rightarrow \text{Node}(l, r) \rightarrow \text{tree}(l) \rightarrow \text{tree}(r))
\]

We can then use the predicate \( \text{tree} \) to repairs tree data structures. For example, Figure \[5a\] 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 \( \text{tree} \). 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 \( \text{null} \) \[15\] and changing the left field of \( e \) to \( \text{null} \) \[15\]. Both fixes are valid with respect to the \( \text{tree} \) specification.

### 6. RELATED WORK

STARFIX is inspired by the line of research on specification-based repair for data structures. In this line of work, Elkababil 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. \[25–27\] 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 \[25\] and abstracting and memorizing repairs to apply to similar repairs \[27\]. 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 [3] 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 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 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 [25][22], 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