A Case for Automated Debugging Using Data Structure Repair

Muhammad Zubair Malik Khalid Ghori Bassem Elkarablieh Sarfraz Khurshid
University of Texas, Austin TX 78712

***Abstract*—Automated debugging is becoming increasingly important as the size and complexity of software increases. This paper makes a case for using constraint-based data structure repair, a recently developed technique for fault recovery, as a basis for automated debugging. Data structure repair uses given structural integrity constraints for key data structures to monitor their correctness during the execution of a program. If a constraint violation is detected, repair performs mutations on the data structures, i.e., corrupt program state, and transforms it into another state, which satisfies the desired constraints.**

The primary goal of data structure repair is to transform an erroneous state into an acceptable state. Therefore, the mutations performed by repair actions provide a basis of debugging faults in code (assuming the errors are due to bugs). A key challenge to embodying this insight into a mechanical technique arises due to the difference in the concrete level of the program states and the abstract level of the program code: repair actions apply to concrete data structures that exist at runtime, whereas debugging applies to code. We observe that static structures (program variables) hold handles to dynamic structures (heap-allocated data), which allows bridging the gap between the abstract and concrete levels.

We envision a tool-chain where a data structure repair tool generates repair logs that are used by a fault localization tool and a repair abstraction tool that apply in synergy to not only identify the location of fault(s) in code but also to synthesize debugging suggestions. An embodiment of our vision can significantly reduce the cost of developing reliable software.

I. Introduction

Debugging faults in code is tedious and can itself be error-prone. Using a traditional debugging environment, a programmer has to manually trace the execution of the program. On finding a corrupted program state the programmer has to make assumptions about fault location(s) and create possible fix(es). This can be quite time-intensive; moreover the fix may introduce some new bugs. Sometimes it is hard to trace the root fault as the fault may seem to propagate from one place to another. A variety of tools and techniques have been developed that help localizing bugs in programs [1]–[3]. Sometimes localizing bugs and fixing code manually can create more bugs or different types of bugs that might not have been present earlier.

We envision a constraint-based technique for automated repair of buggy programs that eases the burden of debugging by suggesting likely fixes to faulty code. Our technique first uses the buggy program to generate corrupt program states, next it repairs these states by invoking an off-the-shelf data structure repair tool, and then it abstracts the repair actions and synthesizes code that represents a likely fix. To gain confidence in the correctness of the suggested program repair, our technique supports scope-bounded testing, for example, using the Korat [4] test generation framework.

A variety of tools [5], [6] have recently been developed to repair structurally complex data that do not satisfy the desired structural integrity constraints at run-time. Conventional use of these tools has been to enable continued execution of programs in case of otherwise fatal data structure corruption. One such tool is Juzi [7]–[10]. To fix corrupt data structures, Juzi generates and applies *repair actions* that represent mutations to the structures so that the transformed structures satisfy the desired constraints. We envision a non-conventional use of data structure repair in this paper, namely for automated debugging through a transformation of concrete repair actions into source code which represents a likely bug fix.

The primary contribution of this paper is three-fold [11]: (1) to introduce the idea of using data structure repair as a basis of automated debugging, (2) to make a case using an illustrative example for the feasibility of developing a mechanical technique that embodies our idea; and (3) to illustrate fault scenarios that an effective automated debugging technique must handle.

While there is a large body of research on automated debugging and fault localization, e.g., using delta debugging [3] and statistical debugging [12], to the best of our knowledge, our work [11] is the first to propose data structure repair for repairing faulty code and generating debugging suggestions for complex data structures.

In the following section we provide background on Juzi framework and its search; following that we demonstrate the likely usefulness and potential usage of our technique on an example.

II. Background: Juzi

Juzi is an automated framework for on-the-fly repair of data structures [7]–[10]. Given a corrupt data structure, as well as a `repOk` method that describes the structural integrity constraints, Juzi systematically mutates the fields of the corrupt data structure so that it satisfies the given constraints. In addition to repairing the given structure, Juzi reports the *repair actions* it performed on the corrupt structure in a log-file that holds a sequence of tuples $\langle o,\ f,\ o' \rangle$, i.e., an assignment to field f of object o the value o'—each tuple represents a repair action.

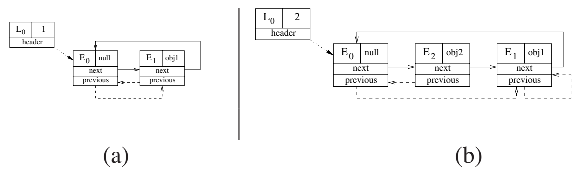

Figure 1. Doubly-linked circular list with sentinel `header`. (a) An empty list (size 0). (b) An erroneous list of size 1 containing element 0. A small box-pair represents a list object and is labeled with the object's identity and the value of its `size`. Large box-pairs represent entry objects and are labeled with object identity and value of `element`.

To illustrate Juzi and its repair mechanism, consider the example of repairing corrupt doubly linked lists (Section III). Consider the list in Figure 1 (b). The list has one corruption in the `previous` field on node E_0. Given the corrupt structure and the `repOk` method (Section III), Juzi first invokes `repOk` on the corrupt structure and monitors the fields accessed by `repOk` during its execution. When `repOk` returns `false` due to a constraint violation, Juzi systematically mutates the *last* field accessed by `repOk` by non-deterministically setting it to : (1) null, (2) nodes that have already been visited during `repOk`'s execution, and (3) one node that has not yet been visited.

To illustrate, by monitoring the execution of `repOk`, Juzi detects the fault in the `previous` field of node E_0, and mutates its value first to `null`, which does not repair the fault, and then to node(s) that have been previously encountered during the execution of `repOk`. Since E_0 is the original value of the field, Juzi does not need to try it again (unless some other fields are modified first). Therefore, Juzi tries node E_1 next, and this repair action fixes the fault in the structure.

In addition to repairing the corrupt structure, Juzi reports the tuple $\langle E_0$, `previous`, $E_1\rangle$ to indicate the repair action that fixed the corruption. Note that although Juzi tries several mutations on corrupt fields, only the repair actions that result in repairing the fields are reported.

To provide more effective repair, Juzi tries to preserve the reachability of the data in the original structure. In particular, if a sequence of repair actions generates a structure that satisfies the constraints but has fewer data than the original structure, Juzi performs further repair actions to preserve the reachability, if possible.

The next section describes how to translate these repair actions into code statements that can be used as effective suggestions for debugging faulty code.

III. Example

Consider the following declaration of a class that implements doubly-linked circular lists based on the class `java.util.LinkedList` from Java libraries:

```
public class LinkedList {
    private Entry header = new Entry(null, null, null);
    private int size = 0;

    public LinkedList() {
        header.next = header.previous = header;
    }

    private static class Entry {
        Object element;
        Entry next;
        Entry previous;

        Entry(Object element, Entry next,
              Entry previous) {
            this.element = element;
            this.next = next;
            this.previous = previous;
        }
    }
```

Each list object has a sentinel `header` node and caches the number of nodes in the field `size`. The class `Entry` implements the list nodes. Each node has an `element`, and `next` and `previous` pointers to other nodes. Figure 1 (a) illustrates an empty list.

The structural invariants (called *class invariants* in object-oriented programs) of doubly-linked lists are sentinel `header` nodes, circularity along `next` fields, transpose relation between `next` and `previous` fields, and correct value for `size`. Any valid list must satisfy these invariants (in all publicly visible states). These invariants can be represented using a `repOk` [13] method that traverses its input structure and returns true if and only if the input satisfies all the invariants:

```
public boolean repOk() {
    if (header == null) return false;
    if (header.element != null) return false;
    Set visited = new HashSet();
    visited.add(header);
    Entry current = header;
    while (true) {
        Entry next = current.next;
        if (next == null) return false;
        if (next.previous != current) return false;
        current = next;
        if (!visited.add(next)) break;
    }
    if (current != header) return false;
    if (visited.size() - 1 != size) return false;
    return true;
}
```

Class invariants implicitly form a part of the preconditions and postconditions of public methods. Thus, all executions of a public method are expected to terminate in a state where the class invariants hold.

Consider the following implementation of the method `addFirst`, which uses the (erroneous) helper method `addBefore`:

```
public void addFirst(Object e) {
    addBefore(e, header.next);
}

private Entry addBefore(Object e, Entry entry) {
    Entry newEntry =
            new Entry(e, entry, entry.previous);
    newEntry.previous.next = newEntry;
    newEntry.next.previous = entry; // fault
    size++;
    return newEntry;
}
```

The method `addBefore` has a fault in its third assignment statement (Line 46), which erroneously sets a `previous` field to `entry` instead of `newEntry` (as correctly implemented by `java.util.LinkedList.addBefore`). To illustrate the effect of this fault, consider the following code snippet:

```
50.         LinkedList l = new LinkedList();
51.         assert l.repOk(); // pass
52.         l.addFirst(0);
53.         assert l.repOk(); // fail
```

The second assertion (Line 53) fails. Figure 1 (b) illustrates the erroneous list in the poststate of `addFirst`. The `previous` field of the `header` node (E_0) is erroneously set to the node itself (instead of E_1).

Given the erroneous list and the `repOk` method, Juzi repairs the list by performing the following repair action: $\langle E_0$, `previous`, $E_1\rangle$, i.e., by setting the `previous` field of E_0 to E_1, thereby generating a valid list of size 1 containing the element 0—the list a correct implementation of `addBefore` would generate.

To transform Juzi's repair action into a debugging suggestion in the form of a Java statement, we envision maintaining a map, `Map<Variable, Object>`, from statically declared variables that are visible at the control point (including the input parameters, such as `this`) to their values at that control point for the current execution. To illustrate, consider executing `addFirst` on the empty input list shown in Figure 1 (a). The map at Line 46 for this execution is:

variable	value
`newEntry`	E_1
`this`	L_0
`e`	0
`entry`	E_0

Note the map is with respect to the variables that are visible in the context of the method that contains the control point. Thus, if a method invokes a helper method, the map is updated to reflect the invocation.

Given the map, we can generate handles as *path expressions* to the objects that appear in a repair action. To illustrate, consider the repair action $\langle E_0$, `previous`, $E_1\rangle$. Applying a path-expression-based *abstraction* to this action's source object E_0 using the variable-value map at the method exit point generates two loop-free expressions "`newEntry.next`" and "`entry`"—both expressions evaluate to E_0. Since methods that manipulate input object graphs often use local variables as pointers into the input graphs for traversing them and accessing their desired components, we can give priority to local variables in breaking a tie. Thus the object E_0 is abstracted by the expression "`newEntry.next`". Moreover, the object E_1 is abstracted by the expression "`newEntry`". Therefore, the repair action $\langle E_0$, `previous`, $E_1\rangle$ abstracts to the following Java statement:

```
        newEntry.next.previous = newEntry;
```

This statement serves as *abstract repair code* and provides a debugging suggestion: append the sequence at the tail of the execution path (just before the `return` statement). While this suggestion is likely to fix the specific erroneous execution, it does so by undoing any erroneous field mutations of the execution—technically, this may qualify as a bug fix, however, the user may have to go through a tedious process of determining what fault each assignment statement is fixing. Ideally, we would like to mechanically determine where the erroneous mutations are located in the faulty code and to replace them with repaired mutations.

We envision an algorithm that takes as inputs the faulty method, the input that exhibits an erroneous output, the sequence of assignment statements that represent repair actions, termed *repair statements*, and the corresponding sequence of concrete repair actions. Intuitively, the algorithm determines for each repair statement where to place it in the faulty method. There are two placement possibilities: (1) replace an existing statement with it, or (2) insert it in the execution path as a new statement. The algorithm first tries to find an existing statement for replacement, but if it fails to find such a statement, it inserts it as a new statement.

To illustrate, in the faulty method `addBefore`, for the repair statement "`newEntry.next.previous = newEntry;`" and the corresponding repair action $\langle E_0$, `previous`, $E_1\rangle$, the statement on Line 46 is the one to replace, since it assigns to the same object field as the repair action.

IV. Constraint-based automated debugging — initial thoughts and fault scenarios

Given a Java method that takes as input structurally complex data, the structural invariants that the method must preserve, and an input that leads to an invariant violation by the method, we envision a program repair technique that performs three core steps:

- It uses off-the-shelf data structure repair (Juzi [10]) to transform the erroneous output into a program state that satisfies the structural invariants;
- It abstracts the set of concrete repair actions into a sequence of Java statements using variables visible within the scope of the method; and
- It determines (heuristically) where in the method the generated statements fit.

There are various fault scenarios that an effective automated debugging technique must handle. We next illustrate some such scenarios.

A. Straight-line code; missing assignment

Consider the `addLast` method, which invokes the erroneous `addBefore`:

```
 1.    public void addLast(Object e) {
 2.        addBefore(e, header);
 3.    }
 4.
 5.    private Entry addBefore(Object e, Entry entry) {
 6.        Entry newEntry =
 7.               new Entry(e, entry, entry.previous);
 8.        newEntry.previous.next = newEntry;
 9.        // newEntry.next.previous = newEntry;
10.        size++;
11.        return newEntry;
12.    }
```

Assume `addBefore` erroneously omits the assignment on line 9. Given the input list in Figure 1 (a), `addLast` generates the erroneous output shown in Figure 1 (b).

B. Code with loop; fault outside of loop body

We next illustrate faulty code that has a loop but does not perform destructive updates to object fields within the loop body. Consider the following method `add` that takes as input an `index` and an `element` to insert in `this` list at the given `index`:

```
 1.    public void add(int index, Object element) {
 2.        addBefore(element,
 3.               (index==size ? header : entry(index)));
 4.    }
 5.
 6.    private Entry entry(int index) {
 7.        if (index < 0 || index >= size)
 8.            throw new IndexOutOfBoundsException(
 9.                          "Index: "+index+
10.                          ", Size: "+size);
11.        Entry e = header;
12.        if (index < (size >> 1)) {
13.            for (int i = 0; i <= index; i++)
14.                e = e.next;
15.        } else {
16.            for (int i = size; i > index; i--)
17.                e = e.previous;
18.        }
19.        return e;
20.    }
```

Invocation of `add` first invokes the method `entry`, which uses `for`-loops to find the appropriate entry, and then invokes the (erroneous) method `addBefore`, which performs destructive updates.

Assume the following erroneous implementation of `addBefore`, which has a different fault from the ones considered before (Sections III and IV-A):

```
21.    private Entry addBefore(Object e, Entry entry) {
22.        Entry newEntry =
23.               new Entry(e, entry, entry.previous);
24.        newEntry.previous.next = null; // newEntry;
25.        newEntry.next.previous = newEntry;
26.        size++;
27.        return newEntry;
28.    }
```

The assignment to `next` is erroneous (Line 24). Figure 2 illustrates an input list `l` for the invocation `l.add(0, 10)` and the corresponding erroneous output list.

Consider repairing the erroneous output with Juzi. It detects the corruption in the `next` field of E_0. Setting it to E_0 allows the list to be repaired by a second repair action that sets the `previous` of E_0 to E_0. However, Juzi rules out this

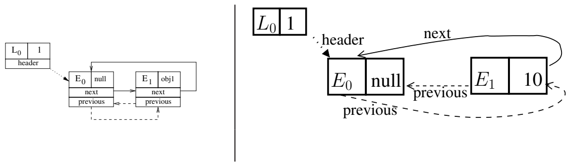

Figure 2. (a) Input list `l` for method invocation `l.add(0, 10);`. (b) Erroneous output list.

sequence of repair actions since it results in the loss of a data element (10) which was reachable in the corrupt structure (Section II). Juzi continues to backtrack on E_0`.next`, setting it to E_1, which generates a repaired structure that preserves reachability. Thus, Juzi's repair action is $\langle E_0$, `next`, $E_1\rangle$.

C. Code with loop; fault inside of loop

We next illustrate a case where destructive updates are performed within the body of a loop. Consider the following method `remove`, which removes the first occurrence of the given object from the list:

```
 1.    public boolean remove(Object o) {
 2.        if (o==null) {
 3.            for (Entry e = header.next; e != header;
 4.                 e = e.next) {
 5.                if (e.element==null) {
 6.                    remove(e);
 7.                    return true;
 8.                }
 9.            }
10.        } else {
11.            for (Entry e = header.next; e != header;
12.                 e = e.next) {
13.                if (o.equals(e.element)) {
14.                    remove(e);
15.                    return true;
16.                }
17.            }
18.        }
19.        return false;
20.    }
21.
22.    private Object remove(Entry e) {
23.        if (e == header)
24.            throw new NoSuchElementException();
25.
26.        Object result = e.element;
27.        e.previous.next = e.next;
28.        e.next.previous = e.next; // e.previous;
29.        e.next = e.previous = null;
30.        e.element = null;
31.        size--;
32.        return result;
33.    }
```

Invocation of the public method `remove` uses a for-loop to identify the relevant node and a helper method to remove the corresponding entry from the list.

The assignment on Line 28 is erroneous. Figure 3 illustrates an input list `l` for the invocation `l.remove(10)` and the corresponding erroneous output list.

Consider repairing the erroneous output with Juzi. It detects the corruption in the `previous` field of E_0. Setting it to E_1 generates a repaired list. Juzi accepts this list as a valid repaired structure since it has the same reachability of elements as the corrupt structure—the entry E_2 in the

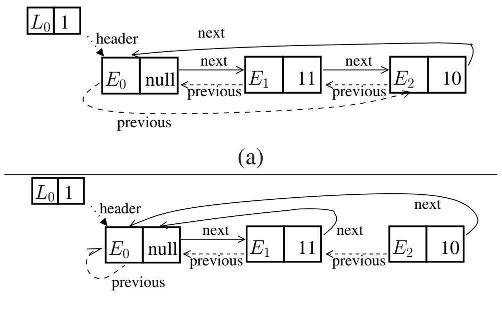

Figure 3. (a) Input list `l` for method invocation `l.remove(10);`. (b) Erroneous output list.

output list is not reachable from the `header`. Thus, Juzi's repair action is $\langle E_0$, `previous`, $E_1\rangle$.

V. Discussion and Conclusion

This paper presented the idea of using constraint-based data structure repair as a basis of automated debugging. An illustrative example demonstrated the feasibility of developing a technique that embodies the idea. Fault scenarios illustrated some key types of faults that an effective automated debugging technique must support.

To increase confidence in the correctness of the repaired method, constraint-based program repair allows a direct application of the Korat framework for systematic testing [4], [14] to automatically generate valid inputs and check outputs using `repOk`. Moreover, any bugs discovered by Korat can feedback into the repair technique to use it to iteratively debug a faulty program that has multiple faults along different control-flow paths.

While this paper proposes a technique for automated debugging, algorithms that embody such a technique enable other novel applications, e.g., for highly optimized data structure repair. Abstract repair code could be injected into the faulty method to allow it to repair its own output on-the-fly without having to repeatedly run Juzi to repair the output. This approach has the potential of providing a substantial speed-up since Juzi performs a systematic search and requires repeated executions of `repOk` on each candidate repair action. Injecting abstract repair code would replace the search and perform repair in a negligible amount of time.

Another application is for programming by sketching [15]. The user could annotate the right-hand-side of a field assignment statement as unspecified, which can be treated initially as `null` and then repaired using our technique.

We plan to develop our core technique for program repair to handle a large class of faults and to explore various novel applications in future work.

Acknowledgment

This material is based upon work partially supported by the NSF under Grant Nos. IIS-0438967, CCF-0702680, and CCF-0845628, and AFOSR grant FA9550-09-1-0351.

References

[1] N. Gupta, H. He, X. Zhang, and R. Gupta, "Locating faulty code using failure-inducing chops," in *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, 2005.

[2] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *Proc. 24th International Conference on Software Engineering (ICSE)*, 2002.

[3] A. Zeller, "Isolating cause-effect chains from computer programs," in *Proc. 10th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2002.

[4] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated testing based on Java predicates," in *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, Jul. 2002.

[5] B. Demsky and M. Rinard, "Automatic detection and repair of errors in data structures," in *Proc. Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2003.

[6] B. Demsky, "Data structure repair using goal-directed reasoning," Ph.D. dissertation, Massachusetts Institute of Technology, Jan. 2006.

[7] S. Khurshid, I. García, and Y. L. Suen, "Repairing structurally complex data," in *12th SPIN Workshop on Model Checking of Software*, San Francisco, CA, Aug. 2005.

[8] B. Elkarablieh, I. Garcia, Y. L. Suen, and S. Khurshid, "Assertion-based repair of complex data structures," in *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Atlanta, GA, 2007.

[9] B. Elkarablieh and S. Khurshid, "Juzi: A tool for repairing complex data structures," in *Proc. 30th International Conference on Software Engineering (ICSE)*, Leipzig , Germany, May 2008, research Demo Paper.

[10] B. Elkarablieh, "Assertion-based repair of complex data structures," Ph.D. dissertation, University of Texas at Austin, 2009.

[11] K. Ghori, "Constraint-based program repair," Master's thesis, Department of Electrical and Computer Engineering, University of Texas at Austin, Aug. 2006.

[12] T. Chilimbi, B. Liblit, K. Mehra, A. Nori, and K. Vaswani, "Holmes: Effective statistical debugging via efficient path profiling," in *31st International Conference on Software Engineering (ICSE 2009)*, Vancouver, Canada, May 2009.

[13] B. Liskov and J. Guttag, *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.

[14] D. Marinov, "Automatic testing of software with structurally complex inputs," Ph.D. dissertation, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 2004.

[15] A. Solar-Lezama, R. Rabbah, R. Bodik, and K. Ebcioglu, "Programming by sketching for bit-streaming programs," in *Proc. ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, 2005.