Dynamic Symbolic Data Structure Repair

Ishtiaque Hussain, Christoph Csallner
Computer Science and Engineering Department
University of Texas at Arlington
Arlington, TX 76019, USA
ishtiaque.hussain@mavs.uta.edu,csallner@uta.edu

ABSTRACT

Generic repair of complex data structures is a new and exciting area of research. Existing approaches can integrate with good software engineering practices such as program assertions. But in practice there is a wide variety of assertions and not all of them satisfy the style rules imposed by existing repair techniques. I.e., a "badly" written assertion may render generic repair inefficient or ineffective. In this paper we build on the state of the art in generic repair and discuss how generic repair can work effectively with a wider range of correctness conditions.

We motivate how dynamic symbolic techniques enable generic repair to support a wider range of correctness conditions and present DSDSR, a novel repair algorithm based on dynamic symbolic execution. We implement the algorithm for Java and report initial empirical results to demonstrate the promise of our approach for generic repair.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Error handling and recovery, symbolic execution*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Assertion checkers, class invariants, reliability*

General Terms

Algorithms, Reliability, Verification

Keywords

Data structure repair, data structure invariants, dynamic symbolic execution

1. INTRODUCTION

Generic repair of complex data structures is a new and exciting approach to software robustness [5, 4, 6, 7, 10]. It promises to mutate the state of a running program in such a way that the resulting state satisfies a given assertion or correctness condition. It is generic in the sense that a single repair algorithm can repair many kinds of data structures. It thereby differs from traditional repair, as in traditional repair for each kind of data structure we need a separate repair algorithm. This makes generic repair exciting and potentially very powerful. Indeed, initial generic repair techniques [5, 6] and implementations such as Juzi [7] are very promising.

Besides runtime data structures and their respective correctness conditions, a generic repair algorithm does not need any additional inputs. This makes the correctness conditions the focal point of generic repair. A key challenge is that a badly written correctness condition may render generic repair inefficient or ineffective. In this paper we build on the state of the art in generic repair and discuss how generic repair can work effectively with a wider range of correctness conditions. Specifically, this paper makes the following contributions.

- We motivate how dynamic symbolic techniques enable generic repair to support a wider range of correctness conditions.
- We present DSDSR (Dynamic Symbolic Data Structure Repair), the first dynamic symbolic algorithm for generic repair of complex data structures.
- We describe an implementation of our algorithm within our new dynamic symbolic execution framework for Java.
- We present initial empirical results to demonstrate the promise of our approach for generic repair.

We describe our implementation in terms of object-oriented software and especially Java programs, but the algorithm equally applies to related languages (C++, C#, etc.) and related programming paradigms (i.e., functional and procedural languages).

2. BACKGROUND AND MOTIVATING EXAMPLE

We start with the necessary background on program assertions and generic repair, which we will need for a motivating example.

2.1 Assertions and Correctness

Software customers and developers likely have an informal notion of the conditions under which the state and behavior of their programs are correct. Such informal notions are the

ICSE '10, May 2-8 2010, Cape Town, South Africa

notions of program correctness that typically matter most in real-world software applications. Large parts of software engineering are therefore concerned with capturing informal correctness notions and transforming them into more formal ones, culminating in the fully formal notion of source code. This influences the terminology we use in this paper. By *correct* we mean correct in the informal sense of the user. A *correctness condition* tries to capture this informal correctness in a more formal notion.

An easy way to write down a correctness condition is to add to the program text a simple if-condition or program assertion. Empirical evidence suggests that programmers write assertions into their code and that code that contains more assertions tends to contain fewer bugs [9].

Correctness conditions may be expressed in different styles and languages, ranging from formal modeling languages to program assertions. For this work, we concentrate on program assertions. Assertions are attractive as programmers do not have to learn a separate language in order to write down correctness conditions.

The main assumption we use in this paper is that correctness conditions have been engineered to be correct. This assumption is also used in all previous work we are aware of. However, we do not require the related assumption that is often made in this area, namely that correctness conditions also satisfy style rules that are specific to a certain repair technique. These rules may conflict with other concerns. Our goal is therefore to provide an approach that does not depend on such repair-specific style rules.

2.2 Juzi and Generic Repair

Assertions are evaluated periodically as part of the program execution, to determine if the program is still in a correct state. If the program state is found to be incorrect, maybe due to a bug on a rarely exercised execution path, the program is typically aborted, analyzed, fixed, redeployed, and finally restarted. But there may be situations in which this is impossible. For example, even aborting may not be a viable option if a program is required to keep running to support critical services. Such scenarios are typically addressed with a combination of techniques, including redundancy and traditional, specialized repair routines. An emerging additional technique that may be included in such a mix is generic repair.

Juzi is the state of the art approach to generic repair using assertions [6, 7]. It builds on the Korat framework [1] and monitors the execution of the assertion to determine the order in which the assertion accesses data structure fields. When the assertion returns false, Juzi mutates the value of the field that was accessed just before the assertion failed. If this does not result in a satisfactory state, Juzi backtracks in the list of field accesses and continues with the field that the assertion has accessed earlier. Each repair attempt mutates the original state in one or more fields.

Depending on the field type, Juzi uses different techniques to determine candidate values. For integer fields, it employs a constraint solver, for reference fields Juzi uses the Korat technique of skipping isomorphic structures in an otherwise exhaustive trial and error approach.

2.3 Motivating Example

To illustrate how our algorithm works and how it improves upon the state of the art, we now consider the singly linked list data structure given in Figure 1. The list has a correctness condition, namely that the first node has a value that is equal to the number of nodes in the list. This condition is implemented by the repOk method, which first stores the value field of the first node in a temporary variable named length. Then the method iterates over the list nodes to count them. This loop terminates prematurely once the node count exceeds the value of the length variable. This also prevents lists that are circular from forcing repOk into an infinite loop. Finally, the length variable is compared with the node count, to produce the desired answer.

```
public class Node {
  int value;
  Node next;
  // ..
}

public class LinkedList {
  Node header;
  // ..
  public boolean repOk() {
    Node n = header;
    if (n==null)
      return true;
    int length = n.value;
    int count = 1;
    while(n.next != null) {
      count += 1;
      n = n.next;
      if (count > length)
        return false;
    }
    if(count != length)
      return false;

    return true;
  }
}
```

Figure 1: Example singly linked list data structure, abbreviated, consisting of a Node class and a LinkedList class. Method repOk is a contrived correctness condition for the linked list, which may be invoked by assertions throughout the program.

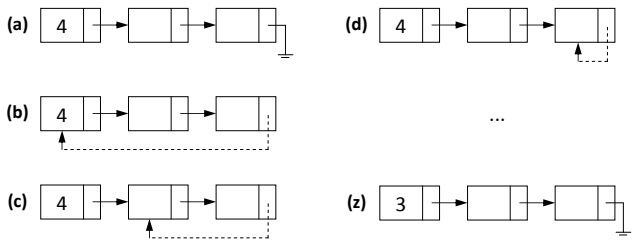

Figure 2: Exhaustive approach in Juzi. Initially (a), the LinkedList is corrupt, as the first node contains value 4, which is incorrect according to the Figure 1 repOk correctness condition. Dotted lines show the first three repair attempts (b, c, d). Omitted are several subsequent repair attempts. Ultimately repair culminates in the correct list (z).

Figure 2 (a) shows an example linked list that consists of

three nodes. The first node has a corrupt value. According to the correcntess condition encoded in the Figure 1 repOk method, the value should be equal to the number of nodes in the list, 3, but it is off by one, 4. To repair the corruption, Juzi first (b, c, d) tries all possible mutations of the field that the repOk method accessed last—the next pointer of the last node. Subsequently (omitted from the figure), Juzi backtracks in the list of fields accessed by the repOk method and continues repair actions in an exhaustive fashion. Finally, Juzi reaches a field that the repOk method had accessed very early, the corrupt value field of the first node, and now Juzi repairs the list successfully (z). For each repair attempt Juzi executes the repOk method, to check if the resulting list satisfies the repOk correctness condition.

In situations like this, an exhaustive approach works well for repairing small data structure instances, containing few nodes. But when repairing larger structures, at some point exhaustive search becomes inefficient. The number of possible mutations grows exponentially and most mutations do not result in a correct state.

Our key insight is that we can guide data structure repair by mutating the data structure in such a way that the repaired data structure takes a predetermined execution path. In our example, we want to invert the outcome of the last if-condition such that, instead of returning false, repOk returns true. Indeed, if we take the path condition of the original path, which returned false, invert the last conjunct, and solve the resulting path condition, we can obtain the correct repair action directly.

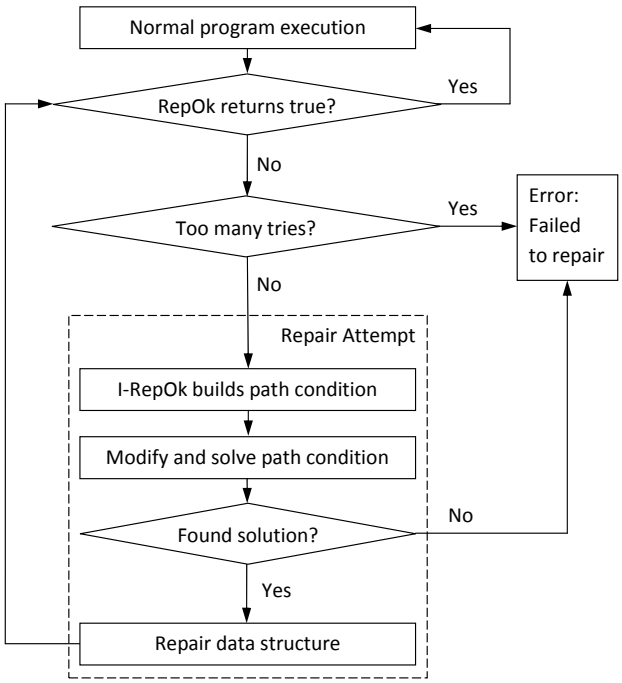

Figure 3: Overview of our dynamic symbolic data structure repair algorithm (DSDSR). RepOk is a method that implements a given correctness condition. I-RepOk is the instrumented version of repOk.

3. PROPOSED APPROACH

Dynamic symbolic data structure repair (DSDSR) consists of two parts. At the lower level is a dynamic symbolic execution engine that has a broad interface to allow modification of path conditions, etc. Using the dynamic symbolic engine, at the top layer sits our generic repair algorithm. In this section we briefly describe both components.

3.1 Dynamic Symbolic Execution Engine

Our dynamic symbolic execution engine automatically inserts instrumentation code into a given repOk method, which yields an instrumented version of repOk. The execution of the instrumented version behaves just like the original, except that it also creates a symbolic representation of the program execution state. In that our engine is similar to previous ones such as Dart, jCute, and Pex [8, 12, 13]. When we apply dynamic symbolic execution to repOk, we obtain a complete symbolic representation of the path taken by the repOk correctness condition. For example, we now have a symbolic representation of the last if-condition, whose concrete value triggered repOk to return false.

3.2 Algorithm for Data Structure Repair

Figure 3 gives a high-level overview of our algorithm. As part of its normal execution, a program invokes assertions or other methods that implement a correctness condition. In our description we follow previous work and name such a method repOk [6, 7]. When the correctness condition fails, i.e., repOk returns false, execution is handed over to our extended dynamic symbolic engine, which in turn invokes the instrumented version of repOk. Executing the instrumented repOk builds the path condition of the execution path that leads to the point at which repOk failed.

With the full symbolic path condition in hand, we can now modify the path condition to obtain a different path. I.e., if we invert the last if-condition we obtain a path that does not return false at the point at which the original execution failed. At the same time, solving such a new path condition can give us an input state that will trigger the new path. If the new state satisfies the repOk correctness condition, we can mutate the existing state to resemble the new one, which completes the repair.

The algorithm relies on a faithful encoding of the path condition and other program constraints in a format suitable for automated reasoning. It further relies on a powerful automated constraint solver that can simplify such constraints and, if a solution exists, can produce a concrete solution. Finally, the solution of the constraint solver needs to be mapped back into the program state, to repair the existing data structures.

We repair the data structure according to the solution of the constraint solver and invoke repOk to check if the resulting structure satisfies the repOk correctness condition. If repOk again returns false, we may make another iteration and attempt another repair. To prevent an infinite loop of repair attempts, the algorithm terminates after reaching a user-defined number of futile attempts. If the repOk method returns true, we consider the repair attempt to be successful and resume normal program execution.

The main advantage of our approach is that, unlike Juzi, in the search for a data structure that satisfies a repOk correctness condition, we do not need to exhaustively generate all possible candidate data structures. Instead, DSDSR derives conditions directly from the repOk implementation to

generate a single data structure that satisfies the correctness condition.

3.3 Implementation

We implement our generic repair algorithm on top of our new dynamic symbolic execution engine for Java, called Dsc. Dsc works on top of any standard Java virtual machine. It does not require modifications of the virtual machine or the user code. This means we can repair existing Java code when it is executed on a standard JVM.

Dsc analyzes user code at the bytecode level. It uses the instrumentation facilities provided by Java 5 to instrument user code at load-time, using the ASM bytecode instrumentation framework [3, 2]. By instrumenting code at the bytecode level, we can repair third-party libraries that are only available in bytecode form.

4. PRELIMINARY RESULTS

We conducted a small experiment using the linked list subject of Figure 1. The experiment applies the motivating example of Figure 2 to lists of different lengths. Specifically, each run constructs a correct singly linked list of a given length, corrupts the value of the first node by increasing it by one, invokes one of the repair tools, and measures the time the tool takes to suggest the correct repair.[1] In the experiment both tools succeed in that they terminate with producing the correct repair action in all cases.

We conducted the experiment with the latest version of Juzi (0.0.0.1) which we obtained from the Juzi website[2] and took all measurements on a Sun HotSpot JVM 1.6.0_17 running on Windows on an intel laptop 2.26GHz Core2 Duo processor.

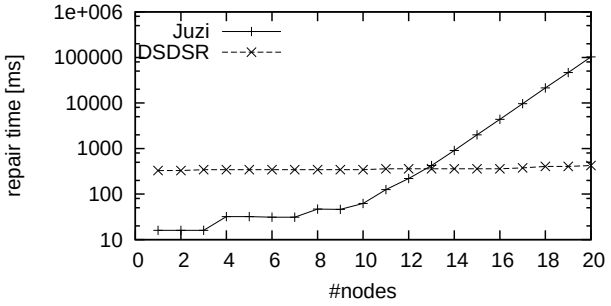

Figure 4: Result of applying the linked list example of Figure 2 to lists of different lengths. #nodes is the number of nodes in the list. Repair time is the time a tool took to produce the correct repair action. Smaller repair times are better.

Figure 4 shows the result of our experiment. Juzi repairs small lists more efficiently than our prototype implementation. But starting with 13 nodes, our approach is more efficient. This makes sense intuitively, as an exhaustive approach such as Juzi is bound to be inefficient for larger data structures, motivating more directed approaches such as ours.

[1]Our prototype suggests but not yet performs a repair action. The implementation of performing a suggested repair is straightforward and will only add a negligible overhead.
[2]http://users.ece.utexas.edu/~elkarabl/Juzi/index.html

5. RELATED WORK

Generic data structure repair, pioneered by Demsky and Rinard [5], is a relatively new area of research. Non-generic data structure repair is not new, classic examples include the IBM MVS/XA operating system [11].

6. CONCLUSIONS

Our prototype DSDSR implementation is available at `http://cseweb.uta.edu/~ishtiaqu/repair/`

Acknowledgments

We thank Bassem Elkarablieh and Sarfraz Khurshid for helping us with Juzi.

7. REFERENCES

[1] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 123–133. ACM, July 2002.

[2] É. Bruneton, R. Lenglet, and T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. In *Proc. ACM SIGOPS France Journées Composants 2002: Systèmes à composants adaptables et extensibles (Adaptable and extensible component systems)*, Nov. 2002.

[3] G. A. Cohen, J. S. Chase, and D. L. Kaminsky. Automatic program transformation with Joie. In *Proc. USENIX Annual Technical Symposium*, pages 167–178. USENIX, June 1998.

[4] B. Demsky. *Data Structure Repair Using Goal-Directed Reasoning*. PhD thesis, Massachusetts Institute of Technology, 2006.

[5] B. Demsky and M. C. Rinard. Automatic detection and repair of errors in data structures. In *Proc. 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 78–95. ACM, Oct. 2003.

[6] B. Elkarablieh, I. Garcia, Y. L. Suen, and S. Khurshid. Assertion-based repair of complex data structures. In *Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 64–73. ACM, Nov. 2007.

[7] B. Elkarablieh and S. Khurshid. Juzi: a tool for repairing complex data structures. In *Proc. 30th ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 855–858. ACM, May 2008.

[8] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223. ACM, June 2005.

[9] G. Kudrjavets, N. Nagappan, and T. Ball. Assessing the relationship between software assertions and faults: An empirical investigation. In *Proc. 17th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 204–212. IEEE, Nov. 2006.

[10] M. Z. Malik, K. Ghori, B. Elkarablieh, and S. Khurshid. A case for automated debugging using data structure repair. In *24th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Nov. 2009.

[11] S. Mourad and D. Andrews. On the reliability of the IBM MVS/XA operating system. *IEEE Transactions on Software Engineering (TSE)*, 13(10):1135–1139, Oct. 1987.

[12] K. Sen and G. Agha. Cute and jCute: Concolic unit testing and explicit path model-checking tools. In *Proc. International Conference on Computer Aided Verification (CAV)*, pages 419–423. Springer, Aug. 2006.

[13] N. Tillmann and J. de Halleux. Pex - white box test generation for .Net. In *Proc. 2nd International Conference on Tests And Proofs (TAP)*, pages 134–153. Springer, Apr. 2008.