Assertion-based Repair of Complex Data Structures

Bassem Elkarablieh Ivan Garcia Yuk Lai Suen Sarfraz Khurshid
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712
{elkarabl,garcia,suen,khurshid}@ece.utexas.edu

ABSTRACT

Programmers have long used assertions to characterize properties of code. An assertion violation signals a corruption in the program state. At such a state, it is standard to terminate the program, debug it if possible, and re-execute it. We propose a new view: instead of terminating the program, use the violated assertion as a basis of repairing the state of the program and let it continue.

We present a novel algorithm to repair complex data structures. Given a structure that violates an assertion that represents its integrity constraints, our algorithm performs a systematic search based on symbolic execution to repair the structure, i.e., mutate it such that the resulting structure satisfies the given constraints. Heuristics to prune search and minimize mutations enable efficient and effective repair.

Experiments using libraries and applications, such as a naming architecture and a database engine, show that our prototype efficiently repairs complex structures while enabling systems to recover from potentially crippling errors.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic execution*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Error handling and recovery*

General Terms

Reliability, Verification

Keywords

Data structure repair, Assertion-driven development, Symbolic execution, Systematic search, Error recovery

1. INTRODUCTION

Software systems are steadily growing in complexity and size. At the same time, reliability is becoming a more and more vital concern. A commonly used methodology to develop reliable software is software verification, which aims at detecting bugs and in some cases showing their absence.

ASE'07, November 5–9, 2007, Atlanta, Georgia, USA.

Several verification techniques are based on the use of specifications. A form of specification that programmers use routinely is *assertions*—statements that evaluate boolean expressions that represent desired properties. If an assertion evaluates to false at runtime, the program is deemed to have reached an erroneous state.

Errors—however seemingly innocuous—in a program state, if left untreated, can have serious consequences. The standard approach when an error is detected at runtime, is to terminate the program, debug it if possible, and re-execute it. An alternative to program termination is *repair* [10, 17, 28]: instead of terminating the program, repair its state and let it continue. The aim of repair is to generate a state that allows the program to recover on-the-fly from an erroneous state and resume its operation.

1.1 Assertion-based repair

We present a new repair technique: *assertion-based repair*, which uses a violated assertion as the basis of performing repair. We present an algorithm that takes as inputs (1) an assertion that describes *what* properties the program state must satisfy and (2) a state that violates them, and generates a new state that satisfies the desired properties. Our repair algorithm performs a systematic state-space exploration of a neighborhood of the given state and uses *symbolic execution* [25] as well as heuristics to perform efficient and effective repair.

We focus on repairing structurally complex data, which pervade object-oriented languages and are characterized by *class invariants* that represent structural integrity constraints. Good programming practice advocates the use of class invariants in assertions, by writing the invariants as predicates, often named `repOk`, which return `true` if and only if their input satisfies its constraints [26].

Given a `repOk` predicate and a structure that violates the constraints, our algorithm performs *repair actions* that mutate the given structure so that it satisfies the constraints. For efficient and effective repair, we employ (1) pruning techniques based on our previous work on the Korat framework for assertion-based testing [3]; (2) decision procedures for primitive data [24]; and (3) heuristics, such as limiting the number of repairs when exploring field assignments. To solve the path conditions that arise during symbolic execution, we implement a solver for difference constraints [13], and use the automatic theorem prover CVC Lite [2] for more complex ones. To enable effective repair, we employ the following heuristic: preserve reachability from the structure's root, as much as possible.

We have implemented our repair algorithm and evaluated its efficiency on repairing a variety of structurally complex subjects, including library classes from the Java Collection Framework [31] as well as stand-alone applications, such as an intentional naming architecture for dynamic networks [1] and a database engine [32].

Experimental results show that our prototype efficiently and effectively repairs corrupt structures with a small number of errors

and up to a few thousand objects. In addition to repairing a corrupt structure, our framework provides feedback to the user to point out the fields that were repaired and an abstraction of the state of the structures before and after repair.

Assertion-based repair enables a unified framework for software verification and resilient computing—two methodologies that traditionally have employed very different algorithms for software reliability. For example, using Korat in conjunction with our repair framework, a program annotated with assertions is (1) systematically tested before deployment and (2) trusted to execute without corruption once deployed—using the very same assertions. The unification has the potential to make a profound impact on improving the quality of software by providing software verification and resilient computing together at the cost of writing assertions—which programmers are already familiar with.

We first introduced the idea of assertion-based repair in a workshop paper [22]. This paper builds on our previous work (Section 2) and makes the following contributions:

- **Algorithm** Our algorithm repairs complex structures using assertions that state *what* properties a valid structure should have and are written in the programming language itself, and handles all of the Java data-types. Our use of symbolic execution is non-conventional.
- **Heuristics** Our algorithm deploys various heuristics that enable efficient and effective repair.
- **Abstractions** (1) We abstract the repair to report a log, which summarizes the repair actions and can assist with debugging and (2) we allow abstraction functions to compare states before and after repair.
- **Implementation** We have implemented a repair framework for Java programs. Our framework repairs Java data structures and uses the CVC Lite theorem prover.
- **Evaluation** Experiments using a variety of subjects, including two stand-alone applications, show that it is feasible to efficiently repair complex structures.

2. RELATED WORK

Repair is not a new idea in the context of fault-tolerance and error recovery; developers have been incorporating *dedicated* repair routines in software systems for a couple of decades. File system utilities, such as `fsck` and `chkdsk`, routinely check and correct the underlying file structure. Some commercially developed systems, such as the IBM MVS operating system [28] and the Lucent 5ESS telephone switch [17], have provided routines for monitoring and maintaining properties of their data structures. These routines, however, do not perform repair using a description of the data structure constraints, but implement special algorithms that can only repair specific data structures.

The use of structural integrity constraints to perform repair is relatively new. Demsky and Rinard [10, 12] present a framework that repairs based on constraints written in a new declarative language that is similar to the first-order relational language Alloy [21]. Repair is performed by translating the constraints to disjunctive normal form and solving them using an ad hoc search. To help the user formulate constraints, they have explored [11] integrating repair with Daikon's dynamic invariant generation [14].

Checkpointing and roll-back are standard database mechanisms to recover data to the last known good state. DIRA [30] adapts these mechanisms to detect buffer overflow attacks and repair the structures damaged by the attack. While such mechanisms provide protection against attacks and erroneous interactions with the environment, they cannot correct the behavior of a buggy program.

Runtime verification (RV) [18] allows synthesizing monitors from specifications for debugging as well as checking safety properties at run-time. Monitoring-oriented programming (MOP) [5] generalizes RV by supporting logic plug-ins that allow users to specify both monitor deployment and error recovery. Our repair mechanism enables MOP to provide automatic default error recovery.

A heuristic we use during repair is to bound the number of repair actions to be performed and iteratively increase the bound (Section 4.2). Such bounded searches are prevalent in AI. They are also gaining popularity in the software verification community. For example, *beam search* has shown to work efficiently in finding deadlocks using the Java Pathfinder model checker [16]. More recently, bounded context switching has been shown to be effective for systematic testing of multi-threaded programs [29].

We introduced a basic algorithm for assertion-based repair in a workshop paper at SPIN 2005 [23]. Building on that work, this paper makes several new contributions: all of the Java constructs including arrays are handled; heuristics, abstractions and repair logs are introduced; and repair algorithm is evaluated using subjects that include stand-alone applications. Our framework can now handle structures that are more than ten times larger than those that were possible with our previous work.

Assertion-based repair is closely related to assertion-based test input generation since both perform constraint solving. This paper shows how to repair using state-space pruning that we developed for test generation in the Korat framework [3]. Our recent work on test generation shows how repair enables efficient generation of large inputs [13]. Our key insight is that while the problem of generating an input that satisfies all the given constraints is hard, generating a structure at random, which may not satisfy the constraints but contains a desired number of objects is straightforward. Indeed, a random structure is unlikely to satisfy any desired constraint. However, we can repair it to satisfy all desired constraints.

We are exploring a static analysis to enable more efficient repair. The initial results suggest the practicality of using assertion-based repair even when structures contain a hundred thousand nodes.

3. EXAMPLE

We present an example of repairing a circular doubly-linked list to illustrate the repair algorithm. We illustrate how the repair algorithm can on-the-fly repair faults in the structure as well as the primitive fields of the list.

```
class DoublyLinkedList {
  Node header;
  int size;

  static class Node {
    int element;
    Node next;
    Node prev;
  }
}
```

The above code snippet shows a declaration of a doubly linked list. The `DoublyLinkedList` class declares an internal `Node` class that models the nodes of the list. Each list has a `header` field, and stores the number of nodes reachable from `header` in the `size` field. Each `Node` instance holds two pointers, `next` and `prev`, and an integer field, `element`.

The structural integrity constraints of `DoublyLinkedList` are: (1) circular structure along `next`; (2) transpose relation between

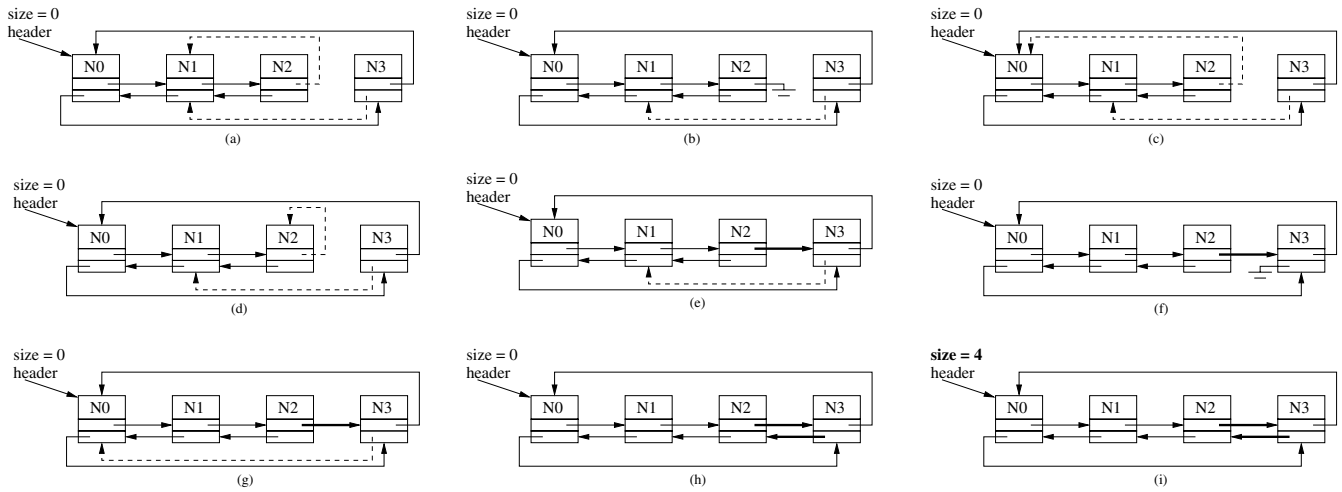

Figure 1: Repairing a circular doubly-linked list. The dashed arrows represent violations of the structural constraints. The bold arrows represent repaired fields. (a) A corrupt structure with three corruptions: `next` field of node `N2`; `prev` field of node `N3`; and `size` field of the `DoublyLinkedList` object. (b–i) Mutations that our repair algorithm performs to repair the corrupt structure.

`next` and `prev` fields; and (3) number of nodes reachable from the `header` following `next` cached in `size`. An empty list has a `null` `header` and its `size` is 0. The following `repOk` predicate represents the class invariant:

```
boolean repOk() {
  if (header == null)
    return (size == 0); // empty
  Set visited = new HashSet();
  visited.add(header);
  Node current = header;
  while (true) {
    Node n = current.next;
    if (n == null) return false; // circularity
    if (!visited.add(n)) {
      if (visited.size() != size) return false;
        else break;
    }
    // prev is transpose of next
    if (n.prev != current) return false;
      current = n;
  }
  return true;
}
```

An assertion can invoke `repOk` to check the structural constraints. For example, the following Java `assert` statement checks them at the beginning of the `add` method in `DoublyLinkedList`:

```
Object add(int element) {
  assert repOk();
  ...
}
```

To illustrate repair, consider the structure shown in Figure 1 (a). This list has three corruptions: (1) the `next` of `N2` is `N1` but the `prev` of `N1` is not `N2`; (2) the `prev` of `N3` is `N1` but the `next` of `N1` is not `N3`; and (3) the value of `size` is not equal to the number of nodes in the list.

Given the corrupt structure in Figure 1 (a), and the `repOk` predicate, the repair algorithm first invokes `repOk` on the structure, and then monitors the field accesses during the execution of `repOk`. When `repOk` returns `false` due to a constraint violation, the repair algorithm systematically mutates the last field accessed by `repOk` (Section 4).

To illustrate, Figures 1 (b–i) show the sequence of mutations that our repair algorithm performs on the corrupt structure. During the first invocation of `repOk`, the last field accessed is `next` of `N2`. Thus, the algorithm systematically mutates this field to: (1) `null`; (2) a list node already encountered during `repOk`'s invocation (`N0` and `N2`); and (3) a list node not yet encountered during `repOk`'s invocation (`N3`). (Note that the algorithm does not use node `N1`, since it is the original value of the `next` field, and it has already been checked by the first execution of `repOk`.) After each mutation the algorithm invokes the `repOk` predicate again to check for constraint satisfaction. Setting `N2.next` to `N3` allows execution of `repOk` to proceed further. The algorithm then detects the corruption in the `prev` field of node `N3`, and repairs it similarly (Figures 1 (e-h)). Finally the algorithm detects and repairs the `size` field of the `DoublyLinkedList` object.

To repair faults in the primitive fields of the structure, the repair algorithm uses on-demand symbolic execution (Section 7.2) where a corrupt field is treated symbolically, and a path condition is computed for that field during the execution of `repOk`. Once the path condition is computed, the algorithm then uses an automated theorem prover, CVC-lite [2], to solve the path condition, and determine the correct value to repair the field. In this example, the repair algorithm sets the value of the `size` field to the number of nodes visited during the execution of `repOk` and in turn generates the repaired structure in Figure 1 (i).

For this example, the repair algorithm performed a total of seven mutations to repair the violations in the structure of initial corrupt list in addition to one mutation to repair the violation in the `size` field. We term these mutations *repair actions* in the rest of the paper.

4. REPAIR ALGORITHM

The problem of assertion-based repair is: given a structure `s` and a predicate method `repOk` that represents desired structural integrity constraints such that `!s.repOk()`, perform *repair actions*, on `s` to transform it into a structure `s'` such that `s'.repOk()`.

Figure 2 gives a high-level overview of the repair algorithm, which performs a systematic search and uses symbolic execution [25]. Two key classes enable repair: `Explorer` and `PathCondition`.

```
boolean repair(Method repOk, Object s)
    throws Exception {

  Explorer.initialize(s);
  boolean done = false;
  do {
    PathCondition.initialize();
    if (repOk.invoke(s)) {
      if (!PathCondition.isFeasible()) continue;
      done = true;
      break;
    }
  } while (Explorer.nextState());
  return done;
}
```

Figure 2: Repair algorithm.

The `Explorer` class provides a framework for state space exploration. It implements an `initialize` method which takes an input structure, and initializes the search space. To initialize the search space, the `initialize` method traverses the structure, and for each type T, primitive or reference, it records $domain(T)$ which holds all the values of type T that are stored in the structure. The recorded domains represent the candidates for repairing the corrupt fields in the structure. The `Explorer` class also keeps track of the current state of the search and provides the `nextState` method which puts the search into the next possible state. The `nextState` method returns `false` if the entire state space is explored. The order through which the search candidates are enumerated is described in Section 4.1.

The `PathCondition` class enables tracking the path conditions that arise in symbolic execution. The path condition is a set of constraints on program variables that, when satisfied, enable the program execution to take a specific program path. The `initialize` method clears the path condition; `isFeasible` checks whether the current path condition is satisfiable; and `update` adds a constraint to the path condition.

The algorithm starts by initializing its search environment by calling `Explorer.initialize` on the structure. This includes traversing the structure, recording the datatypes, building domains with objects (values) encountered for each type, and initializing internal structures to keep track of the search progress. All the encountered objects and values are initially marked as *non-visited*.

The algorithm proceeds by repeatedly invoking `repOk` on the given structure. During each invocation of `repOk` the repair algorithm: (1) monitors the order of field accesses and (2) if `repOk` returns `false`, non-deterministically updates the value of the last field accessed—if all values have been checked, systematically backtracks to update the value of the second last field accessed and so forth (Section 4.1).

The algorithm terminates when `s` is repaired, i.e., `s.repOk()` returns `true` and the corresponding path condition is satisfiable, or when the search is exhausted, i.e., all the field mutations fail to repair the structure.

4.1 Non-deterministic field assignments

When `repOk` reads an object field or an array element for the first time, our algorithm non-deterministically assigns it a value from the domains computed during initialization, and marks the value as *visited*. The case for array element is identical to that of an object field, because each array index can be viewed as a unique field of the array object [3].

Let `repOk` read field f of object o such that $o.f == v$ for some value v (at the time of read). There are two cases to consider: reference field access and primitive field access.

References. Let f be a reference field of type T. The algorithm non-deterministically assigns $o.f$:

- v, i.e., its current value;
- `null`, if $v \neq$ `null`;
- a visited value w of a type T, such that $v \neq w$ and w has already been encountered during `repOk`'s invocation;
- if T is a basic type, a new (non-visited) value of type T, if v is not different from all values of type T already encountered during `repOk`'s invocation;
- if T is an array type with element type S, a new array of type $S[]$ of non-deterministic length $l \geq 0$.

Primitives. Let f be a primitive field of type T. The algorithm non-deterministically assigns $o.f$:

- v, i.e., its current value;
- a new symbolic value V, and adds the constraint $V \neq v$ to the current path condition.

Notice that a primitive field access may introduce symbolic (integer) values. For these values, invocations of `repOk` follow forward symbolic execution [24,25], and we check satisfiability of path conditions using CVC Lite [2]. Since all fields initially have concrete values, the first execution of `repOk` follows standard Java semantics for these values. Thus, if the structure is initially not corrupt, `repOk` simply returns `true` indicating that the structure is valid.

Our repair algorithm builds on our previous work on the Korat test input generator [3] and symbolic execution [23, 24], and explores only non-isomorphic structures [3].

4.2 Heuristics

In this section, we describe some heuristics that the repair algorithm incorporates to perform data structure repair.

Bounded number of repairs. The repair algorithm bounds the number of repairs it performs on any invocation of `repOk` and iteratively relaxes the bound. Doing so provides two benefits: (1) the perturbations to the structure (in terms of number of mutations) are minimized; and (2) repair is more efficient. Moreover, as we discuss in Section 7.1, we expect the given structure to contain a small number of errors. Therefore, we do not expect the algorithm to go through a large number of iterations.

Domains for primitive values. Due to the enormous number of values that primitive variables can generally take and the arbitrary nature of constraints that programs can impose on their values, our algorithm bounds the possible values for a variable by defining *domains*. For a primitive type T, the algorithm computes $domain(T)$ by: (1) traversing the given (corrupt) structure and collecting all values of type T encountered during the traversal; and (2) generating at random a fixed number of new values of type T. Doing so allows the algorithm to re-use old values as much as possible and introduce new ones only when necessary. We also support user-defined domains.

On-the-fly path condition simplifications. CVC Lite provides a C++ API for checking validity of formulas over several interpreted theories including linear arithmetic on integers and reals, arrays and uninterpreted functions. Since CVC Lite is implemented in C++, it can be expensive to make calls to it from a Java program.

Our algorithm implements on-the-fly simplifications of path conditions. The simplifications not only allow it to generate smaller path conditions but also, in some cases, let it decide satisfiability without having to call the CVC Lite routines. The simplifications include transforming constraints in a path condition to a canonical form, performing subsumption checking for simple cases, and propagating constants.

4.3 Abstraction

To enable the user to understand the mutations performed during repair, we provide them an abstraction of the repair performed, if they so desire. The abstraction specifies the set of fields that are mutated. Such information can help the user debug their program (if the corrupt structure was a result of a bug in the program). Moreover, we also provide the user a pair of abstract values that represent the structure before and after repair. The users can choose to provide their own abstraction functions if they like. As a default, we provide a function, α, that counts the number of values of each type reachable from root o: $\alpha(o) = \{\langle n, T\rangle | n$ is number of values of type $T\}$. We also report the new object count for each type.

To see the utility of this feedback, consider structure s that is repaired to s'. Let T be a type, n be the number corresponding to T in $\alpha(s)$, and m be the number corresponding to T in $\alpha(s')$. If $m == n$ and the algorithm reports that no new objects of type T were allocated, the user knows that the repair only re-structured original values of T, while re-using all of them without introducing any new ones.

5. IMPLEMENTATION

We have implemented the repair algorithm for Java programs. To enable non-deterministic field assignments and symbolic execution, (1) we provide library classes that implement the data structures for backtracking, path conditions and algebraic expressions over integers, and (2) we instrument the given `repOk` predicate into a functionally equivalent predicate that uses our libraries to enables repair. We use the Bytecode Engineering Library (BCEL) [8] and the Java Programming Assistant (Javassist) [6] for performing the instrumentation at the bytecode level. We used a similar approach in previous work with source-code instrumentation to perform test generation [3, 24]. We next illustrate the code instrumentation and implementation in detail.

5.1 State Space Exploration

We have briefly described the `Explorer` class that enables state space exploration in Section 4. This section provides more details.

To support non-deterministic choices, the `Explorer` class provides a `choose` method that takes an integer which represents the number of non-deterministic choices and returns an integer which represents one of these choices. For example, the assignment

```
x = Explorer.choose(3);
```

non-deterministically assigns the values $0, 1, 2, 3$ to `x`. Such non-deterministic choice operators are an essential feature of software model checkers [19, 33].

To keep track of the current choice, `Explorer` holds a counter for each call site for the method `choose` in the program. At each call site, the first call to `choose` adds a counter in the `Explorer` class and initializes its value to 0. Further calls at a call site return the value of the counter. `Explorer` also provides a `nextState` method that increments the value of the last added counter. Once all the possible choices for a counter are explored, the corresponding counter is deleted. The `nextState` returns `true` if a counter is incremented, and `false` when all the counters are deleted (this indicates that the state space is explored). To illustrate, consider the following example:

```
class DoublyLinkedList {
  static Set visitedLists, nonVisitedLists;
  static Set visitedNodes, nonVisitedNodes;

  Node header; boolean header_is_initialized;
  SymbolicInt size; boolean size_is_initialized;

  void header(Node n) {...} // added set method
  Node header() {...} // added get method

  void size(SymbolicInt i) {...} // added set method
  SymbolicInt size() {...}  // added get method

  static class Node {
    SymbolicInt element; boolean element_is_initialized;
    Node next; boolean next_is_initialized;
    Node prev; boolean prev_is_initialized;

    void next(Node n) {...} // added set method
    Node next() {...} // added get method

    void prev(Node n) {...}  // added set method
    Node prev() {...} // added get method
  }
}
```

Figure 3: The instrumented `DoublyLinkedList` class. A boolean field and two accessor methods are added for reference fields (to support non-deterministic field assignment) and for primitive fields (to support symbolic execution). Two sets are added for each reference type to keep track of the visited and non-visited objects of that type.

```
void spaceExploration() {
L1.  Explorer.initialize();
L2.  do {
L3.    int i = Explorer.choose(1);
L4.    int j = Explorer.choose(2);
L5.    System.out.println(i + "  " + j);
L6.  } while(Explorer.nextState());
   }
```

The calls to the `choose` method at lines `L3` and `L4` set the search space by creating two counters that count from 0 to 1 and 0 to 2 respectively. The `nextState` increments the value of the counters. The `do..while` loop executes until both counters reach their maximum value and the `nextState` returns `false`. The output of executing the method `spaceExploration` is:

```
0 0
0 1
0 2
1 0
1 1
1 2
```

By associating field domain values with integer indices the `choose` method enables non-deterministic field assignments.

5.2 Structure Mutation

Our repair algorithm mutates the structure based on `repOk`'s executions. To enable non-deterministic field assignments we instrument the Java bytecode of both the structure classes and the predicate method.

Class instrumentation: For each field in the structure, we add a boolean variable `field_is_initialized` that indicates whether

```
boolean repOk() {
  if (header() == null)
    return (size().ifEQ(new IntConstant(0));
  Set visited = new HashSet();
  visited.add(header());
  Node current = header();
  while (true) {
    Node n = current.next();
    if (n == null) return false;
    if (!visited.add(n)) {
      // symbolic check; updates the pathcondition
      if (size().ifNEQ(visited.size())) return false;
        else break;
    }
    if (n.prev() != current) return false;
    current = n;
  }
  return true;
}
```

Figure 4: The instrumented `repOk` method. All the field accesses are transformed into method invocations. Operations on primitive values are changed into operation on symbolic primitives. Conditional statements are changed into method invocations that update the path condition.

a field is being accessed for the first time. To monitor field accesses and allow non-deterministic assignment, we add *get* and *set* methods for each field. These methods provide the functionality for accessing and setting field values, and enable an observer to note the order of accesses. Additionally, for each class type, we use two new `java.util.Set` fields that represent the sets of visited and non-visited objects so far.

To illustrate, the instrumented `DoublyLinkedList` from Section 3 is displayed in Figure 3.

`repOk` instrumentation: To monitor the order of field accesses in `repOk`, we instrument the method's bytecode by changing all the field accesses to method invocations of the added accessor methods. To illustrate, the following bytecode corresponds to the statement "`Node current = header;`" from the `DoublyLinkedList`:

```
26:  aload_0
27:  getfield #24; //Field header:
LDoublyLinkedList\$Node;
30:  astore_2
```

The instrumented bytecode is:

```
32: aload_0
33: invokevirtual #165; //Method header:()
LDoublyLinkedList\$Node;
36: astore_2
```

For ease of understanding, Figure 4 illustrates the instrumented code at the source-code level.

The non-deterministic assignment is performed by the methods added during class instrumentation. Figure 5 shows an example of the added methods for the `next` field of the `DoublyLinkedList` class. The first method (the set method) simply sets the value of `next` and marks it as accessed (initialized) by assigning the `next_is_initialized` variable to `true`. The second method (the get method) returns the current value of `next` if it is previously accessed (initialized). If it is not, the get method non-deterministically chooses a value for `next`. The first choice is the original (possibly corrupt) value. This choice reflects the normal behavior of a field access and is made to maintain the normal execution of the structure in case there is no error. The other choices are `null`, a visited

```
// set method for next
void next(Node n) {
  next = n;
  next_is_initialized = true;
}

// get method for next
Node next() {
  if (!next_is_initialized) {
    next_is_initialized = true;

    // non-deterministic choice based
    // on the number of visited nodes
    int i = Explorer.choose(visitedNodes.size() + 2);

    // return the original value and
    // add the node object to the visited node set
    if (i == 0) {
      if (next != null)
        if (visitedNodes.add(next))
          nonVisitedNodes.remove(next);
    }
    // assign null to next
    else if (i == 1) {
      if (next == null)
        Explorer.backtrack();
      next = null;
    }
    // assign an already visited node to next
    else if (i > 1 && i < visitedNodes.size() + 2) {
      Node temp = getVisitedNodeAt(i - 2);
      if (next == temp)
        Explorer.backtrack();
      next = temp;
    }
    // assign a new non-visited node to next
    else if (i == visitedNodes.size()+2) {
      Node temp = getANonVisitedNode();
      if (temp != null) {
        next = temp;
        visitedNodes.add(next);
        nonVisitedNodes.remove(next);
      }
    }
    else
      Explorer.backtrack();
  }
  return next;
}
```

Figure 5: The added accessor methods for `next`. The set method sets the value of `next` and updates the status of the next field by setting the `next_is_initialized` variable to `true`. The get method returns the value of `next` if the field is initialized, and it performs non-deterministic choice and changes the value of `next` according to the selected choice otherwise.

`Node`, and a new non-visited `Node` (as described in Section 4). Note that the algorithm keeps track of the visited and non-visited nodes for each field so that the correct choices are made according to the order in which the fields are accessed in `repOk`.

5.3 Symbolic Execution

To enable symbolic execution, instrumentation replaces (1) type declarations of primitive integer fields and variables with library class `SymbolicInt` and (2) expressions over primitive integer values with invocations of library methods. A conservative reachability analysis allows us to determine which expressions to instrument.

Data structure	repOk (ncnb LOC)	source (ncnb LOC)
Singly-linked acyclic list	10	58
Doubly-linked circular list	21	317
Binary tree	26	128
Red-black tree	129	851
Disjoint set	51	138
Intentional name	21	36,919
Relational database	87	72,140

Table 1: Subjects. Structures, non-comment non-blank (ncnb) lines of code (LOC) in the `repOk`, and ncnb LOC in the subject's implementation (excluding `repOk`).

To allow symbolic execution to explore different program paths, we use a non-deterministic boolean choice whenever there's a branch in bytecode that cannot be deterministically resolved on-the-fly.

5.4 Limitations

Since `repOk` is an arbitrary Java method, finding an input for which the method returns true is undecidable. In fact, non-linear constraints over integers are undecidable. We have not found this to be a problem in practice. A reason for that is that `repOk` predicates are special methods that focus on structural integrity and the constraints of commonly used data structures seldom involve complex arithmetic. Even when a `repOk` uses complex arithmetic, bounded enumeration enables exhaustive exploration of a bounded input space.

Repairing data values in a structure requires care. For example, in repairing `DoublyLinkedList`, while we expect repair to re-establish structural constraints of a doubly linked list, we do not expect repair to modify any particular `element`. If repair introduces spurious elements or arbitrarily re-assigns values to elements, it is not likely to be useful. We mitigate this by allowing users to specify fields that should not be mutated by repair. By declaring `element` as unmodifiable, the user is assured that repair will only re-structure the existing tree entries to satisfy the invariants.

6. EXPERIMENTS

We evaluate our algorithm by applying it to seven subjects, including two from standard Java libraries, as well as two stand-alone applications. For each subject structure, we evaluate the performance of our repair routine by *injecting errors*, i.e., corrupting object fields and repairing them as follows. Given s, the desired size of a structure, and e the desired number of fields to corrupt:

1. Generate a structure of size s;
2. Corrupt e fields at random in the structure; a corruption is a triple $\langle o, f, v\rangle$, where object o's reference field f is assigned value v, which is either `null` or a reference to an object of a compatible type. Primitive fields are similarly corrupted.
3. Repair the corrupt structure

For each subject, we repeat these steps for 50 different randomization seeds and report the average of repair times.

We next describe the subjects, and the repair results. All experiments used a 2.0GHz Pentium 4 with 512MB of RAM.

6.1 Subjects

Table 1 lists the subjects, and the lines of code in `repOk` and subject's implementation. Doubly-linked circular lists and red-black trees [7], which implement balanced binary search trees using node colorings, respectively follow the declarations of `java.util` classes `LinkedList` and `TreeMap`. Red-black trees, with the most complex constraints, have the longest `repOk`. For applications, such as intentional naming (INS [1]), a naming architecture for dynamic network, and relational database (HSQLDB [32]), a popular open-source database, `repOk` focuses on the most complex data structures and its length is a small fraction of the length of the implementation. In particular, the `repOk` for INS specifies the *name specifiers*, which are rooted trees that represent service descriptions in dynamic networks, and the `repOk` for HSQLDB specifies AVL trees [7] that store records using balanced binary search trees.

For all subjects, except the database, we re-used `repOk` predicates that were developed previously [3, 9]. Re-use of specifications is a key strength of assertion-based repair. Indeed, if a `repOk` is already available for a subject, say because it was tested with Korat [3] or automatically generated (Section 7.5), it is used in repair for free.

6.2 Results

We evaluate how the repair time varies with the structure size and the number of corrupt fields.

Table 2 tabulates the results for all seven subjects. For each subject, we tabulate different structure sizes and the time to repair when there are 1, 5, 10, 15 and 20 errors injected. We chose these numbers of errors because in a real situation, we expect a small number of corruptions (Section 7.1).

We label the last five columns $[\#errors \leq n]$ since it is possible (though unlikely) for a randomly generated error to set the value of a field to its original value. The repair times for sizes less than 50 are negligible and not shown here.

For singly-linked acyclic list, Table 2 shows times for $[\#errors \leq 1]$ only as a singly-linked list either has zero or exactly one cycle, irrespective of how many `next` fields are mutated. Since at most one fault can occur is the structure of a singly linked list, our repair algorithm can handle lists with 100,000 objects within 2 seconds.

Among the other subjects, the binary tree and the intentional naming structures are the easiest to repair as the structural constraints are acyclicity, single parent, and reachability from the root. For these structures our algorithm can handle structures with up to 5,000 nodes within ten seconds.

Red-black and AVL trees have more complex constraints than the binary tree and international names and almost always take more time to repair. Despite their complex constraints, our algorithm repairs efficiently. This is in part due to the effectiveness of our heuristics.

Doubly-linked circular lists and disjoint set are the most difficult among the above subjects for our program to repair. The complexity of these structures is in preserving reachability since each node must be reachable from any node in the structure. The first valid structure might not have the same number of nodes as the corrupt structure, and thus, the repair algorithm keeps searching for a valid structure with the original size, if possible. For these structures, the algorithm can repair structures with 400 nodes and $[\#errors \leq 20]$ in around nineteen seconds.

Notice, in some cases, the essentially linear growth in repair time as size is increased while number of errors is kept constant. Notice also the essentially linear growth in repair time as the number of errors is increased while the size of a structure is kept constant. Figure 6 plots the repair times for red-black trees.

We point out that the injected errors cripple the subject implementations, causing failures ranging from unhandled exceptions, such as `ArrayIndexOutOfBoundsException`, to functionally in-

Subject	Size	Repair (ms) [#errors≤1]	Repair (ms) [#errors≤5]	Repair (ms) [#errors≤10]	Repair (ms) [#errors≤15]	Repair (ms) [#errors≤20]
Singly-linked acyclic list	100	≤ 2	not applicable	not applicable	not applicable	not applicable
	1,000	12				
	10,000	138				
	100,000	1,466				
Doubly-linked circular list	50	11	42	83	147	293
	100	37	129	239	451	751
	200	84	619	1,241	2,417	3,922
	400	175	2,502	3,321	6,032	9,906
Binary tree	100	≤ 2	12	27	38	55
	500	14	63	144	210	292
	1000	33	127	341	484	643
	5000	163	844	2,012	3,199	4,560
Red-black tree	250	58	238	581	931	1,190
	500	126	627	1,522	2,441	3,291
	1,000	312	1,430	3,316	4,736	6,323
	2,000	782	3,120	7,218	10,224	14,298
Disjoint set	50	32	167	337	516	793
	100	102	488	1,017	1,682	2,014
	200	467	1491	2,979	5,241	7,377
	400	1,233	5,066	10,735	14,726	19,438
Intentional name	100	6	39	88	131	171
	500	33	192	327	624	808
	1,000	71	422	774	1,212	1,709
	5,000	402	2,088	4,630	6,849	8,709
Relational database	250	74	381	787	1,026	1,499
	500	165	739	1,692	2,215	3,155
	1,000	377	1,837	3,309	4,629	6,800
	2,000	832	3,620	6,844	9,318	13,641

Table 2: Results for applying the repair algorithm on seven subjects including five library structures and two stand-alone applications. Times tabulated are in milliseconds. For some structures, the repair time grows essentially linearly as the size increases for a fixed number of errors.

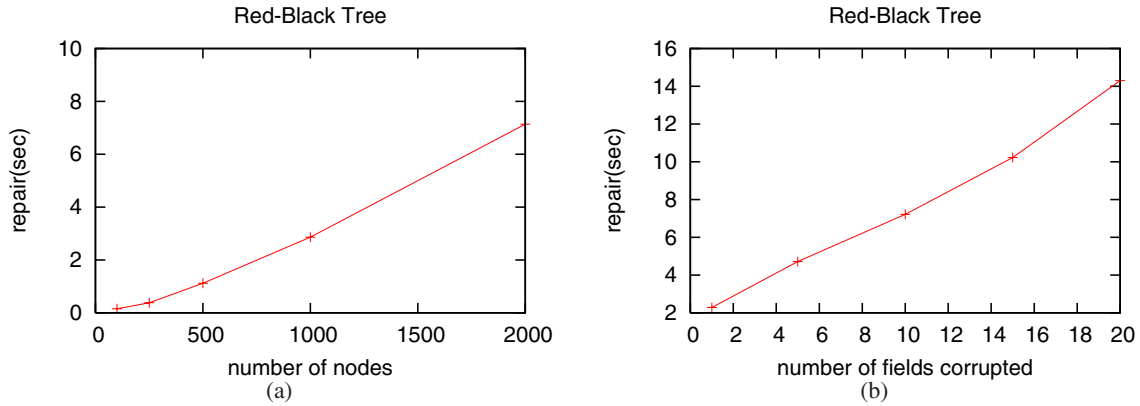

Figure 6: Red-black tree results. (a) Time to repair $\leq$ 10 corrupt fields as tree size increases. (b) Time to repair trees with 2000 nodes as number of corrupt fields increases.

correct outputs, to infinite loops. To illustrate, a cycle in a name specifier structure causes an infinite loop in INS's method to add new advertisements to a name-tree. HSQLDB's execution also goes in an infinite loop or raises a `NullPointerException` on structures that are not acyclic or have an incorrect value for `parent`.

We note that for all these seven subjects, our repair algorithm successfully repaired the corrupt structures and enabled the respective applications to continue to execute. The repaired structures satisfy their integrity constraints, contain no spurious values, and preserve the reachability of nodes.

7. DISCUSSION

We next discuss some characteristics of our approach and present some promising future directions.

7.1 Why assertion-based repair works

There are two key dimensions across which we can compare repair methodologies: ease of application and scalability. Assertion-based repair offers benefits across both.

Ease of application Assertion-based repair requires the user to provide assertions. Indeed, any repair technique has an associated

cost, e.g., the cost of writing a repair routine. We argue that writing assertions poses minimal burden.

An assertion describes *what* a desired state should be, i.e., its properties. A repair routine describes *how* to generate a desired state. Writing the repair routine already requires knowledge of desired properties. Moreover, it requires translating them into a procedure that correctly establishes them. Assertion-based repair, in contrast, requires writing the properties using the programming language. To illustrate, consider red-black trees. Writing a repair routine involves implementing complex re-balancing operations to satisfy the constraints on height, color, etc. In contrast, writing an assertion requires writing conceptually simple tree traversals that *check* the constraints.

In some cases, code already contains assertions, as is advocated by defensive programming [26]. For example, the SGLIB C library [34] comes with annotated class invariants.

Scalability Repair routines need to be efficient. Manually written routines can be highly optimized. We draw an analogy with propositional satisfiability (SAT) to argue why our approach is also likely to be efficient in practice.

Assertion-based repair considers the problem of generating one possibly large structure that satisfies the assertion. A related problem, which arises in test generation, is enumeration of a large number of small structures that satisfy an assertion. Our previous work on Korat [3] presents a feasible solution to the enumeration problem. Results from SAT indicate that repair should be easier than enumeration. The analog for enumeration in SAT is model counting, and the analog for repair is finding one solution, which is believed to be easier than model counting [35].

Furthermore, when repairing a deployed system, we expect that with high probability the repaired structure lies in a small neighborhood of the corrupt structure. This is because if the corruption was due to a major implementation flaw, it would likely have been uncovered during systematic testing before deployment, and if the corruption was due to an external event, such as cosmic radiation, only a few field values are likely to have been corrupted. Repair, therefore, is analogous to the problem of finding one solution for a SAT formula that is *satisfiable with high probability*. For this SAT problem, local search is expected to work well [20].

7.2 On-demand symbolic execution

Our use of symbolic execution is non-conventional not only in our application to data structure repair, but also in how we perform it. Symbolic execution is usually performed either by treating all program inputs as symbolic [25] or by a priori determining which inputs to treat as symbolic and which to treat as concrete (e.g., symbolic primitives and concrete references [24]).

We take a different approach. Our algorithm starts by invoking `repOk` on a structure, all of whose fields have concrete values. During subsequent invocations of `repOk`, the algorithm makes values of certain fields symbolic. However, these values do not have to stay symbolic during all subsequent invocations. A field regains a concrete value once a concrete value that satisfies the data constraints is computed. The hybrid approach enables exploration of a neighborhood of a given structure and efficient generation of a new structure that is heuristically similar to the one given.

While our hybrid approach aims at repair, a combination of concrete and symbolic executions has recently been explored in software testing with significant success [4, 15].

7.3 Sensitivity of repair to `repOk`

Repair actions performed by our algorithm depend on how `repOk` is formulated. Recall that the algorithm backtracks on the *last* field accessed by `repOk` and modifies that field. This means that for the same corrupted structure, two different `repOk` implementations that access fields in different orders may cause our algorithm to produce different structures. Even though this sensitivity to the way constraints are written may be considered a limitation, in fact, it allows the user to control how the structure may be repaired. By ordering constraints appropriately the user can ensure that the algorithm will not perturb the values of certain fields (that the user deems unlikely to get corrupted) unless absolutely necessary.

7.4 Incremental repair

Even though the experiments show the feasibility of assertion-based repair for structures that have a few thousands of objects and a small number of errors, repairing larger structures, say those with tens of thousands of nodes, is still time consuming. We believe an incremental approach to performing repair holds much promise. One example of incremental repair is to implement a stateful search, which allows real backtracking similar to that in the Java PathFinder model checker [33] and obviates the need of repeated invocations of `repOk` from the beginning. Another example is to build a summary of field writes as a program executes so that when there's an assertion violation, repair can focus on repairing first the fields that were modified last.

7.5 Constraint generation

Our repair algorithm expects the user to provide the integrity constraints by writing the `repOk` predicate. For complex constraints, writing a precise predicate is error-prone. Existing constraint generation tools can be used to help users formulate the predicates correctly. We have recently developed Deryaft [27], a tool that specializes in generating constraints of complex data structures. Deryaft takes as input a handful of concrete data structures of small sizes and generates a `repOk` predicate that represents their structural integrity constraints. The constraints generated by Deryaft can directly be used for repair using our framework. For example, for all our benchmarks except red-black trees and disjoint set, Deryaft can generate the precise `repOk`'s using five sample structures for each subject. Even in cases when Deryaft is unable to output a complete `repOk` predicate, Deryaft's output helps the users correctly formulate the predicate, say by using the output as a skeletal implementation.

8. CONCLUSION

We have presented a novel assertion-based repair algorithm that uses systematic search and symbolic execution to repair corrupt program states. Experiments with our prototype on a variety of subjects including library classes and stand-alone applications show that it can feasibly repair complex structures, even those with a few thousand nodes.

Assertion-based methodologies can have a significant impact on improving software quality. Programmers are already comfortable with writing assertions. Providing new analyses for them can make them even more attractive.

The use of assertions in hardware is already immensely popular. Assertion-based verification—a design verification methodology—is an integral part of modern-day chip design. We believe the time has also come to realize the benefits that assertions have long offered in software.

Acknowledgments

We thank the anonymous referees and Darko Marinov for helpful comments. This work was supported in part by NSF ITR-SoD award #0438967 and the EDGE scholar program.

9. REFERENCES

[1] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *Proc. 17th ACM Symposium on Operating Systems Principles (SOSP)*, December 1999.

[2] Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proceedings of the 16th International Conference On Computer Aided Verification*, Boston, MA, July 2004.

[3] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, July 2002.

[4] Cristian Cadar and Dawson Engler. Execution generated test cases: How to make systems code crash itself. In *Proc. 12th SPIN Workshop on Software Model Checking*, 2005.

[5] Feng Chen and Grigore Roşu. Java-MOP: A monitoring oriented programming environment for java. In *Proc. 11th Conference on Tools and Algorithms for the construction and analysis of systems (TACAS)*, 2005.

[6] Shigeru Chiba. Javassist—a reflection-based programming wizard for Java. In *Proceedings of the ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, October 1998.

[7] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.

[8] Markus Dahm. Byte code engineering library. `http://bcel.sourceforge.net/`.

[9] Paul T. Darga and Chandrasekhar Boyapati. Efficient software model checking of data structure properties. In *Proc. Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Portland, OR, 2006.

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

[11] Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCamant, Jeff H. Perkins, and Martin Rinard. Inference and enforcement of data structure consistency specifications. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, July 2006.

[12] Brian Demsky and Martin Rinard. Automatic detection and repair of errors in data structures. In *Proc. Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2003.

[13] Bassem Elkarablieh, Yahya Zayour, and Sarfraz Khurshid. Efficiently generating structurally complex inputs with thousands of objects. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, July 2007.

[14] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington Department of Computer Science and Engineering, Seattle, Washington, August 2000.

[15] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005.

[16] Alex Groce and Willem Visser. Heuristics for model checking java programs. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(4), 2004.

[17] G. Haugk, F. Lax, R. Royer, and J. Williams. The 5ESS(TM) switching system: Maintenance capabilities. *AT&T Technical Journal*, 64(6 part 2), 1985.

[18] Klaus Havelund and Grigore Rosu. *Workshops on Runtime Verification (RV'01, RV'02, RV'04*. 2001, 2002, 2004.

[19] Gerald Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[20] Holger H. Hoos. SAT-encodings, search space structure, and local search performance. In *Proc. Sixteenth International Conference on Artificial Intelligence (IJCAI)*, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[21] Daniel Jackson. *Software Abstractions: Logic, Language and Analysis*. The MIT Press, Cambridge, MA, 2006.

[22] Sarfraz Khurshid, Iván García, and Yuk Lai Suen. Repairing structurally complex data. In *Proc. 12th SPIN Workshop on Software Model Checking*, 2005.

[23] Sarfraz Khurshid, Iván García, and Yuk Lai Suen. Repairing structurally complex data. In *12th SPIN Workshop on Model Checking of Software*, San Francisco, CA, August 2005.

[24] Sarfraz Khurshid, Corina Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Proc. 9th Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Warsaw, Poland, April 2003.

[25] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7), 1976.

[26] Barbara Liskov and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.

[27] Muhammad Zubair Malik, Aman Pervaiz, and Sarfraz Khurshid. Generating representation invariants of structurally complex data. In *Proc. 13th Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2007.

[28] Samiha Mourad and Dorothy Andrews. On the reliability of the IBM MVS/XA operating system. *IEEE Transactions on Software Engineering*, 13(10), 1987.

[29] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proc. ACM SIGPLAN'07 Conference on Programming Language Design and Implementation (PLDI)*, 2007. (To appear).

[30] Alexey Smirnov and Tzi-cker Chiueh. DIRA: Automatic detection, identification, and repair of control-hijacking attacks. In *The 12th Annual Network and Distributed System Security Symposium*, San Diego, CA, February 2005.

[31] Sun Microsystems. *Java 2 Platform, Standard Edition, v1.3.1 API Specification*. `http://java.sun.com/j2se/1.3/docs/api/`.

[32] The hsqldb Development Group. HSQL database engine. `http://www.hsqldb.org/`.

[33] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In *Proc. 15th Conference on Automated Software Engineering (ASE)*, Grenoble, France, 2000.

[34] Marian Vittek. SGLIB—A Simple Generic Library for C. `http://sglib.sourceforge.net/`.

[35] Wei Wei and Bart Selman. A new approach to model counting. In *Proc. 8th Conference on Theory and Applications of Satisfiability Testing (SAT)*, June 2005.