Contract-Driven Data Structure Repair: A Novel Approach for Error Recovery

Razieh Nokhbeh Zaeem, Ph.D.

The University of Texas at Austin

January 4, 2016
Khajeh Nasir Toosi University of Technology (KNTU)

Joint work with Prof. Sarfraz Khurshid
## Outline

### 1 Introduction
- Motivation
- Problem Definition and Approach
- Related Work

### 2 Contract-Driven Data Structure Repair
- Example
- Background
- History-Aware Data Structure Repair
- Repair Abstractions

### 3 Structure Generation with Dynamic Programming
- Problem Definition and Approach
- Example
- Test Input Generation Using Dynamic Programming

### 4 Conclusion and Future Work
- Ideas on Using Dynamic Programming for Repair
- Conclusion
- Contributions
Motivation

- Software reliability is important
- A lot of research is dedicated to before-deployment activities
- Software failure is still frequent and costly
- Examples of software failures
  - Ariane-5 rocket crashed: overflow exception (left)
  - Mars polar lander crashed: premature shut down (center)
  - USS Yorktown was dead in the water: division by zero (right)

Photos: ESA/CNES  Photos: JPL/NASA  Photo: navsource.org
Repair Versus Halt-on-Error

- Many systems get deployed with unknown or unfixed bugs
- When bugs cause failures, the traditional halt-on-error approach
  - May fix the bug permanently
  - But is not always desirable or feasible

![Diagram of traditional and repair approaches to errors]

**Figure:** Traditional approach to errors (left) versus repair approach (right).
Our Repair Approach

- Data structure repair is continuing program operation
  - By fixing the effect of bugs
  - In the program state (i.e., data structures)
  - On-the-fly
- Our approach: contract-driven data structure repair
- Contracts are formal interface specifications for software components including
  - Abstract data types and invariants
  - Pre-conditions and post-conditions
- The idea is to transmute contracts into efficient implementations
Related Work: Constraint-Based Repair

- Constraint-based repair uses data structure invariants (aka repOK)
  - Written using first-order logic [DemskyRinardOOPSLA03] or
  - Java assertions [Khurshid+SPIN05, Elkarablieh+ASE07]
- Does not necessitate writing dedicated repair routines as preceding repair frameworks did
- But data structure constraints are too weak for error recovery

**Figure:** Constraint-based repair is limited to data structure invariants.
Outline

1. Introduction
   - Motivation
   - Problem Definition and Approach
   - Related Work

2. Contract-Driven Data Structure Repair
   - Example
   - Background
   - History-Aware Data Structure Repair
   - Repair Abstractions

3. Structure Generation with Dynamic Programming
   - Problem Definition and Approach
   - Example
   - Test Input Generation Using Dynamic Programming

4. Conclusion and Future Work
   - Ideas on Using Dynamic Programming for Repair
   - Conclusion
   - Contributions
Example

- Binary search tree

```java
1 class BinarySearchTree {
2     Node root;
3     int size;
4
5     boolean remove(int x) {...}
6
7     class Node {
8         Node left, right;
9         int element;
10     }
11 }
```

Listing 1: A binary search tree deceleration in Java.

- Invariants include acyclicity, search property, correct size, etc.
- remove method post-conditions include correct remove, correct remove result
Example Bug

- `left.right = parent (bug)` instead of `parent.right = left`
- Invariants and/or post-conditions might be violated

**Figure**: Example bug manifested as a faulty output.
Our Previous Work: Contract-Based Data Structure Repair Using Alloy

- **Contract-Based Data Structure Repair Using Alloy**
- A Master’s thesis [ZaeemKhurshidABZ10, ZaeemKhurshidECOOP10, ZaeemMS10]
  - Introduced the basic idea of contract-based data structure repair
  - Used Alloy tool-set
    - Example Alloy constraint
      - `all n: t.root.*(left+right) | n !in n.^(left+right) //directed acyclicity`
  - Tarmeem
- Necessary background
Formal Definition of Repair

**Definition**

**Definition:** Let $\phi$ be a method post-condition that relates pre- and post-states such that $\phi(r, t)$ if and only if pre-state $r$ and post-state $t$ satisfy the post-condition. Given a valid pre-state $u$, and an invalid post-state $s$ (i.e., $\neg \phi(u, s)$), mutate $s$ into state $s'$ such that $\phi(u, s')$.

**Definition**

**Definition:** Graph edit distance, defined as the minimum number of edge/node additions/deletions to change a graph to another, is used to measure the similarity between two states.
Background on Alloy and Kodkod

- Alloy is a relational first order logic language
  - Models everything as a relation
  - \( LB(R) \subset instance(R) \subset UB(R) \)
- Alloy Analyzer/Kodkod check Alloy specifications
  - Translate the model to a satisfiability problem
  - Use scope bounded SAT solving
- Relaxation: let the SAT solver decide about the values
- E.g., relax the dotted edge
  - \( LB(right) = \{(N0, N1)\} \)
  - \( UB(right) = \{(N0, N1), (N2, N0), (N2, N1), (N2, N2)\} \)

Faulty output of remove(5)

\[
\begin{align*}
T0.root &\rightarrow N0 : 3 \\
 &\rightarrow N1 : 5 \\
 &\rightarrow N2 : 4 \\
null &\rightarrow l \\
null &\rightarrow l \\
null &\rightarrow l \\
\end{align*}
\]

- \( \text{root} = \{(T0, N0)\} \)
- \( \text{size} = \{(T0, 2)\} \)
- \( \text{right} = \{(N0, N1), (N2, N0)\} \)
- \( \text{left} = \{(N1, N2)\} \)
- \( \text{element} = \{(N0, 3), (N1, 5), (N2, 4)\} \)

Figure: Alloy models.
Basic Contract-Based Repair (Implemented in Tarmeem)

- Basic repair method is oblivious to the current faulty post-state
  - Holds the pre-state constant, relaxes everything in the post-state
    
    root = {(T0, ?)}
    size = {(T0, ?)}
    right = {(N0, ?), (N1, ?), (N2, ?)}
    left = {(N0, ?), (N1, ?), (N2, ?)}
    element = {(N0, ?), (N1, ?), (N2, ?)}
  - Displayed for visualization: ? can be anything

- Tarmeem adds heuristics

faulty output of remove(5)

expected output

basic method repair result

Figure: Basic contract-based repair result.
Challenges for Repair

Repair should be

- **Efficient**
  - When repairing an error
  - When nothing goes wrong

- **Scalable**
  - In enforcing contracts, locating and fixing the error

- **Effective**
  - I.e., introduce little perturbation

- **Usable**
  - I.e, provide logs and feedback

- **Tarmeem was not efficient nor scalable**
  - It was independent of the program dynamic execution
  - Each call to SAT acted independently
History-Aware Contract-Based Repair

- The idea is to use [Zaeem^TACAS12]
  1. Dynamic program history of field writes and reads (through barriers)
     - A barrier is a code sequence to do additional work while writing to or reading from the heap
     - It is widely supported in languages with automatic memory management, like Java
  2. Repair history (through unsatisfiable cores SAT solvers return)
     - Unsat core includes contradicting parts of the contract and state
     - It is sound and irreducible

1 if (!assertContracts()){
2     relaxSAT(writeBarrierLog);
3     if (!assertContracts()){
4         relaxSAT(writeBarrierLog, readBarrierLog);
5         if (!assertContracts()){
6             relaxSAT(unsatCoreFields);
7             if (!assertContracts()){
8                 reportModelInconsistency();
9             }}
10          reportModelInconsistency();
11     }}

History-Aware Repair Example

(a) input

constraint-based repair

(b) expected output of remove(5)

history-aware contract-based repair

(c) faulty output of remove(5)

(d) contract-based repair without history

Figure: History-aware repair result. Dotted and dashed lines indicate written and read fields respectively.
Cobbler implements history-aware repair for Java programs

Subject programs for experiments:
- Singly-linked list
- ANTLR, from DaCapo [Blackburn+06] (BaseTree data structure)
  - A tool to build recognizers, interpreters, compilers, and translators from grammars
- Kodkod (Red-black tree data structure)
  - A SAT-based constraint solver
  - The back-end of Alloy Analyzer
- Cobbler found and repaired a real bug in ANTLR
Experimental Results for Cobbler I

**Singly-Linked List Repair Time (s) - Logarithmic Scale**

- **Logging**
- **Check**
- **Repair**

**Figure**: Performance: repairing singly-linked lists with Cobbler (C).

- **Subject tools**
  1. **Tarmeem (T)** [ZaeemKhurshidABZ10, ZaeemKhurshidECOOP10]
  2. **Enhanced Juzi (J)** [ElkarabliehKhurshidICSE08, Elkarablieh+ASE07]
  3. **PBnJ (P)** [Samimi+ECOOP10]

- **Time measurements**
  1. **Logging time**: the overhead due to logging read and write actions
  2. **Check time**: the time to detect a contract violation
  3. **Repair time**: the time to search and find a repaired data structure
Experimental Results for Cobbler II

Singly-Linked List Repair Time (s) - Logarithmic Scale

Logging  Check  Repair

Figure: Performance: repairing singly-linked lists with Cobbler (C).

- Cobbler performs best for 5 out of 7 types of errors
  - Error 2 skips an update to the size field (not read or written)
  - Error 4 introduces a cycle; Juzi is tailored for such errors
- Cobbler’s overhead on an error-free execution includes
  - Logging time, which is negligible
  - Check time, can we improve it?
Experimental Results for Cobbler III

Figure: Accuracy: repairing singly-linked lists with Cobbler.

- Cobbler, except for one case, always produces exactly the same output as expected (edit distance = 0, similarity = 100%)
- Edit distance is used solely for evaluation and not for optimization
A Bug in ANTLR v3.2

- Public addChild Method of BaseTree class
- The input to this method is a tree
- A check is missing to avoid adding a tree to itself as a child
- Violates acyclicity and ascending child indexes
- Cobbler finds and repairs the erroneous state
  - Zero edit distance
  - 30 s for a tree of 300 nodes
- Contacted ANTLR team
  - They assume acyclicity in the tree but do not check for that
  - Yet, since addChild is public, they should perform checks
Cobbler and Repair Challenges

- Efficiency
  - It speeds up over Tarmeem by reducing the size of the search space

- Scalability
  - It scales ten times better than Tarmeem

- Effectiveness
  - Repair results are 100% to 90% similar to the correct structure in more than 90% of the cases

- Usability
  - Barrier logs can be used as repair reports

- Cobbler still does not scale to real applications
  - Repair overhead should be decreased
    - Use repair abstractions
  - Check overhead is unacceptable
    - Translate Alloy checks to Java and check through JVM, not SAT
The Insight is that once an error is repaired, it might recur.

The idea [Zaeem+RV13]
- Abstract out repair actions
- Reuse them as possible repair action candidates in future
- Malik’s PhD proposal [MalikPhD13] introduced the basic idea of repair abstraction
- We develop it in the context of repair with a SAT back-end
Repair Abstraction Example 1

faulty output of remove(5)

repair result

concrete repair actions:

abstract to:

First(in post-state).right = First.Neighbor.Neighbor(in post-state) and
Repair Abstraction Example II

- Running the same faulty code on a different tree

Faulty output of remove(7)

Repair result

reuse abstract repair actions:

First(in post-state).right = First.Neighbor.Neighbor(in post-state) and

concretize to:

[2].right = [6]
[6].right = null
DREAM (Data structure Repair using Efficient Abstraction Methods)

- A generic framework that piggybacks on different repair frameworks
  1. DREAM concretizes and applies abstracted repair actions, repaired?
  2. If not repaired, DREAM calls the underlying repair framework
  3. DREAM abstracts out the concrete repair actions the underlying repair framework took for future

**Figure:** Placement of DREAM.
Arreh: a Tool for Translating Alloy Checks to Java

- DREAM improves *repair*
- Frequent *checking* of contracts through SAT is time consuming
- Arreh translates Alloy checks to Java, checks through JVM not SAT
- Based on Minshar [Al NaffouriMS04]
  - Arreh adds support for pre- and post-conditions
- Arreh is an extension to Alloy Analyzer
  - Parses the model to Alloy Abstract Syntax Tree (AST)
  - Recursively replaces each operation of AST with a Java method call
- Performance improvement of Arreh is tow-fold
  - Executing Java checks is faster than SAT
  - Translation to Java is a one time operation for Arreh
Alloy Analyzer/Arreh Snapshot

Figure: A snapshot of Arreh.
Arreh Example

```java
public boolean repOK() {
    for (Node n : method0()) {
        if (!method4(n)) { return false; }
    }
    return true;
}
//method0 implements t.root'.*(left'+right')
public Set<Node> method0() {
    Node x0 = getRoot();
    return getTransitiveClosure(x0); }
//method4 implements n !in n.*(left'+right')
public boolean method4(Node n) {
    if (getClosure(n).contains(n)) return false;
    return true; }
```

Listing 4: Simplified Arreh translation to Java.
Experimental Evaluation of DREAM

- DREAM + Cobbler + Arreh
- Same subject errors as Cobbler
- Repair scenario: re-use repair on a structure with a different size but similar fault as repaired before
  E.g., build repair abstraction with respect to an erroneous list with 10 nodes, and apply it to an erroneous list with 500 nodes
Experimental Results for DREAM + Arreh

Figure: Performance: repairing singly-linked lists. Cobbler (C), DREAM (D).

- Time measurements
  1. DREAM repair time includes concretization and application
  2. Check after repair time is to check contracts on the result of DREAM repair

- Improved Cobbler performance in two ways
  1. Cost of initial check (Arreh), also for error-free executions
  2. Cost of repair (DREAM)

- However, repair abstractions do not always work
  - They can be too tailored to a specific problem instance, e.g., error 2
Repair Abstractions and Repair Challenges

- **Efficiency and scalability**
  - DREAM reuses repair actions without searching the space
  - Amortizes the cost of repair from cases that do require a search
  - Arreh improves the efficiency and scalability of checking

- **Effectiveness**
  - Accuracy: DREAM is as accurate as the underlying repair
    - Exact same result as Cobbler when they both worked

- **Usability**
  - Summarizes repair actions into intuitive descriptions for debugging faulty code
Outline

1. Introduction
   - Motivation
   - Problem Definition and Approach
   - Related Work

2. Contract-Driven Data Structure Repair
   - Example
   - Background
   - History-Aware Data Structure Repair
   - Repair Abstractions

3. Structure Generation with Dynamic Programming
   - Problem Definition and Approach
   - Example
   - Test Input Generation Using Dynamic Programming

4. Conclusion and Future Work
   - Ideas on Using Dynamic Programming for Repair
   - Conclusion
   - Contributions
Test Input Generation and Repair

- Testing is the most commonly used methodology to find faults
  - Web browsers, etc. need structurally complex test inputs
- Constraint-based testing is a well-known automation technique
  1. Logical constraints define desired inputs
  2. Constraints are solved for all small instances
  3. Solutions are refined as test inputs
- The problems of contract-based repair and constraint-based test input generation are similar
  - Repair finds a solution to $\phi(pre, post)$ when $pre$ is held constant
  - Input generation enumerates solutions to $\rho(pre)$
- We present a test input generation technique [ZaeemKhurshidFSE12]
- We discuss future ideas to adapt it for repair
Our Test Input Generation Approach

- We present a constraint-based test input generation technique that
  - Leverages recursive structures of complex inputs
    - To enable more natural formulation of constraints
    - To provide faster generation of inputs and better scalability
  - Uses dynamic programming
    - To build larger inputs using smaller ones with the same structure
    - Dynamic programming is a problem solving methodology designed to exploit common subproblems

![Diagram](image)

**Figure:** Combining smaller inputs to build larger ones.
Example: Generating HTML Files

- Many data structures are inherently recursive
- Example: model HTML files as Trees
- Recursive predicates
- Assuming acyclicity
  - The framework is aware that there might be cycles

```java
class BinaryTree {
    BinaryTree left, right;
    int size;
    boolean repOK () {
        int rightsize;
        if (right == null) rightsize = 0;
        else {
            if (!right.repOK()) return false;
            rightsize = right.size;
        }
        ...
        return (size == rightsize + leftsize + 1);
    }
}
```

Listing 5: A recursive binary tree in Java.

Figure: A tree representation of an HTML input.
User writes constraints as recursive predicates

We break the problem into building smaller similar data structures and solve it efficiently using **dynamic programming**
- Non-conventional application

We present three algorithms for test input generation
1. DP: Core algorithm to use dynamic programming
2. LazyDP: Lazy initialization strategy for efficiency
3. SymboLazyDP: Combination with symbolic execution
Three Algorithms: DP Finding Binary Trees I

(Size $\leq 2$)

<table>
<thead>
<tr>
<th>Iteration 1</th>
<th>size = 0</th>
<th>size = 1</th>
<th>size = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>pool</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lastRoundTests</td>
<td>null</td>
<td>size = 0, #0</td>
<td></td>
</tr>
<tr>
<td>thisRoundTests</td>
<td></td>
<td>size = 1, #1</td>
<td></td>
</tr>
</tbody>
</table>

- **pool**: all structures that we have already used to build bigger ones
- **lastRoundTests**: all previously generated structures that we have not yet used
- **thisRoundTests**: all structures that we are generating at this iteration
Three Algorithms: DP Finding Binary Trees II

<table>
<thead>
<tr>
<th>Iteration 2</th>
<th>size = 0</th>
<th>size = 1</th>
<th>size = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>pool</td>
<td>null</td>
<td>size = 0, #0</td>
<td></td>
</tr>
<tr>
<td>lastRoundTests</td>
<td>size = 1, #1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>null</td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>thisRoundTests</td>
<td>size = 2, #2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>repOK</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>null</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>null</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>null</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>null</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>no repOK call</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Non-isomorphism: 
≥ 1 structures from lastRoundTests

Pruning: No structure with > 2 nodes
Three Algorithms: LazyDP

- Data structures are kept in a concise format to preserve memory
- When repOK is called, we need to expand them
- Expand them lazily (only when needed)

Figure: Lazy initialization.
Three Algorithms: SymboLazyDP

- SymboLazyDP avoids a systematic search for *non-recursive* (data) fields
- Symbolic execution of repOK
  - Uses symbols instead of values for non-recursive fields
  - Builds a path condition along the execution path
  - Uses a constraint solver to solve for non-recursive fields
- Saves path conditions (not solutions) with valid substructures
  - After combining the substructures, solves for the entire candidate

![Binary Tree with symbolic values](image)

**Figure:** Binary Tree with symbolic values.
Experimental Evaluation of Dynamic Programming for Test Generation

- RQ1: How efficient and scalable are our algorithms, compared to
  - Korat: an open-source tool for solving (Java) repOK’s
    - Performs execution-driven pruning and isomorphism breaking
  - Pex: a symbolic execution tool from Microsoft
    - We run Pex on (C#) repOK’s (but not full application)
  - Six small subjects: sorted lists, binary trees, red-black trees, Fibonacci heaps, binary heaps, hash tables
  - Evaluate for both systematic and random test generation

- RQ2: How effective are the generated tests in finding real bugs?
  - Two web browsers: Google Chrome and Apple Safari
Experimental Results for Test Generation I

Figure: Systematic generation performance: sorted linked lists.
Experimental Results for Test Generation II

Table: Random generation of ten tests with $90 \leq \text{size} \leq 100$. TO represents a time out of 1000s.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Generation Time (s)</th>
<th>State Space</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Korat</td>
<td>SymboLazyDP</td>
</tr>
<tr>
<td>Linked List</td>
<td>0.137</td>
<td>0.136</td>
</tr>
<tr>
<td>Binary Tree</td>
<td>0.266</td>
<td>0.174</td>
</tr>
<tr>
<td>Red-Black Tree</td>
<td>TO</td>
<td>20.256</td>
</tr>
<tr>
<td>Fibonacci Heap</td>
<td>0.114</td>
<td>0.567</td>
</tr>
<tr>
<td>Binary Heap</td>
<td>7.617</td>
<td>2.823</td>
</tr>
<tr>
<td>Hash Table</td>
<td>TO</td>
<td>3.947</td>
</tr>
</tbody>
</table>
Testing Safari and Chrome: Overview

- We tested the support for rendering CSS3 3D effects
  - CSS allows separating presentation from content
- Applied our approach to directly test the two browsers
- Wrote repOK’s and a test harness and oracle
  - An instance of an HTML file as a tree
  - A CSS rule as a linked list of properties
    - Each property has a linked list of values
- Used staged generation: first generate CSS, then HTML
  - 2 declarations inside a CSS block with 5 properties each
  - 8 tags inside an HTML file
- Used differential testing: image differencing algorithm
Testing Safari and Chrome: Results

Table: Chrome and Safari test input generation results.

<table>
<thead>
<tr>
<th>Gen.</th>
<th>Time (s)</th>
<th>#Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CSS</td>
<td>HTML</td>
</tr>
<tr>
<td>DP</td>
<td>0.140</td>
<td>10.851</td>
</tr>
<tr>
<td>LazyDP</td>
<td>0.140</td>
<td>10.850</td>
</tr>
<tr>
<td>SymboLazyDP</td>
<td>N/A</td>
<td>1.628</td>
</tr>
</tbody>
</table>

- 818 out of 3081 tests failed
- 148 false positives due to the inaccuracy of our test oracle
- Manually classified the remaining failing tests
  - At least three distinct bugs in Chrome
Example Bug Found in Chrome

CSS style sheet

```
.ClassName4{
  -webkit-transform: rotateY(180deg);
}
.ClassName12{
  -webkit-perspective: 800;
  -webkit-backface-visibility: hidden;
}
```

HTML file using the CSS

```
<html>
<head>
<link rel="stylesheet" type="text/css" href="file.css">
</head>
<body>
<div class="ClassName4">
  <h1>This is some text</h1>
</div>
<div class="ClassName12">
  <h1>This is some text</h1>
</div>
</body>
</html>
```

Chrome (second line should be hidden because it is inside ClassName12)

Safari
Outline

1. Introduction
   - Motivation
   - Problem Definition and Approach
   - Related Work

2. Contract-Driven Data Structure Repair
   - Example
   - Background
   - History-Aware Data Structure Repair
   - Repair Abstractions

3. Structure Generation with Dynamic Programming
   - Problem Definition and Approach
   - Example
   - Test Input Generation Using Dynamic Programming

4. Conclusion and Future Work
   - Ideas on Using Dynamic Programming for Repair
   - Conclusion
   - Contributions
Using Dynamic Programming for Repair

1. Memoized checks (Ditto [ShankarBodikPLDI07]) *locate the error*
   - Ditto incrementally checks recursive repOK
   - We improve Ditto with support for
     - Cycles
     - Some pre- and post-conditions, according to the user

2. We use a set of small patches built using SymboLazyDP
   - We solve for symbolic values using the values of the location of error
   - We check if patches repair the data structure using contracts
Example: Ideas on Using Dynamic Programming for Repair

(a) input for remove(7)

root → 2

btSize = 5

repOK():true

1

null

6

null

5

null

7

repOK():true

null

null

repOK():true

(b) faulty output of remove(7)

root → 2

btSize = 4

repOK():false, postCondition(): false

1

null

6

null

5

null

7

repOK():false, postCondition(): false

null

null

repOK():true, postCondition(): true

(c) patches generated with SymboLazyDP

#1

x

null

null

#5, x < y < z

x

null

null

y

null

null

z

null

null

(d) repair result of remove(7)

root → 2

btSize = 4

1

null

6

null

5

null

null

null

null
Dynamic Programming and Repair Challenges

- Efficiency and effectiveness
  - Memoized checks locate the error more accurately
  - This repair technique does not alter correct parts of the state

- Usability
  - Provides an alternative way of describing contracts recursively
  - Amortizes the overhead of writing and maintaining contracts between test input generation and repair
Conclusion

- Contract-driven data structure repair improves our ability to develop reliable programs.
- For programs with contracts, error recovery comes at a low cost.
- The same contracts can be used for testing code before deployment.
  - Using existing techniques and our new technique for input generation.
  - Amortizing the cost of writing and maintaining contracts between testing and repair.
- Contract-driven data structure repair can potentially improve software quality.
Publications

2013

2012

2010
<table>
<thead>
<tr>
<th>Reference</th>
<th>Title</th>
</tr>
</thead>
</table>
References II


References III


[ZaeemMS10] Razieh Nokhbeh Zaeem. 
Contract-based data structure repair using Alloy. 
Master’s thesis, Department of Electrical and Computer Engineering, University of Texas at Austin, May 2010.

Repair abstractions for more efficient data structure repair. 

History-aware data structure repair using SAT. 
Outline

5 Supplemental Slides
Each Alloy relation has its instance with respect to two bounds
- Lower bound: all tuples that a relation \textit{must} have
- Upper bound: all tuples that a relation \textit{may} have
- \( \text{LB}(R) \subset \text{instance}(R) \subset \text{UB}(R) \)
- A lower or upper bound is a function of type \( R \to 2^T \)
Summary of Four Heuristics of Tarmeem

Table: Comparison of the four repair algorithms of Tarmeem.

<table>
<thead>
<tr>
<th></th>
<th>Uses post-state</th>
<th>Uses post-condition to optimize</th>
<th>Uses annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic approach</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Iterative relaxation</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error localization</td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>Guided error localization</td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
</tbody>
</table>
Edit Distance and Similarity

- $r$: repaired data structure
- $e$: expected data structure
- We measure the edit distance in set difference operations using the relational representation
- $\text{dist}(e, r) = \sum_{R}(|\text{inst}_e(R) - \text{inst}_r(R)| + |\text{inst}_r(R) - \text{inst}_e(R)|)$
- $\text{sim}(e, r) = (1 - \frac{\text{dist}(e, r)}{\sum_{R}|\text{inst}_e(R)|}) \times 100$
Abstracting a concrete repair action $o.f = v$ has three steps:

1. Use BFS (for pre-state and post-state) to find sequences of dereferences to reach $o$ and $v$
   - E.g., $[3].right = [4]$ is reachable through $\text{root}(\text{in post-state}).right = \text{root.right.left}(\text{in post-state})$

2. Find all abstractions
   - E.g., $\text{First}(\text{in post-state}).right = \text{First.Neighbor.Neighbor}(\text{in post-state})$, among others

3. Build abstract repair actions $\alpha.f = \beta$

A pre-defined, extensible repository of abstractions:
- Pointer-based (Null, First, Leaf, Self, Neighbor, ...)
- Value-based (Offset, Coefficient, ...)

Faulty output of `remove(5)`
Concretization

- Concretizing an abstract repair action $\alpha.f = \beta$ has two steps
  1. Traverse the data structure (in the appropriate state) to find concrete values for $\alpha$ and $\beta$
  2. Perform the concrete repair, i.e., mutate the post-state accordingly
     E.g., `[2].right = [6]`

All possible previous abstractions with all possible concretizations are tried
DP, LazyDP, SymboLazyDP

1. **DP**: Core algorithm to use dynamic programming
   - Memoizes internal repOK checks (on smaller structures)
   - Prunes based on the expected size
   - Avoids repetitions (non-isomorphic generation)
   - Stores structures as candidate vectors with pointers
   - Supports systematic and random generation

2. **LazyDP**: Lazy initialization strategy for efficiency
   - Expands structure as needed during repOK execution

3. **SymboLazyDP**: Combination with symbolic execution
   - Handles constraints on non-recursive (data) fields
SymboLazyDP

- SymboLazyDP automatically performs source to source instrumentation

```java
if (getBoolean()) {
    addCond("btSize", EQ, "rightBtSize+leftBtSize+1");
    return true;
} else {
    addCond("btSize", NOTEQ, "rightBtSize+leftBtSize+1");
    return false;
}
```

Listing 6: Instrumenting BinaryTree for symbolic execution.
Systematic Generation of Fibonacci Heaps

**Figure:** Systematic generation performance: Fibonacci heaps.
Memoized checks (Ditto [ShankarBodikPLDI07]) *locate the error*

- Ditto incrementally checks repOK
- Memoizes the result of previous checks and re-checks only where the data structure changes
- Uses write barriers to identify changes
- Optimistically assumes repOK returns true
  - Propagates the result up if it does not
- We improve Ditto with support for
  - Cycles
  - Some pre- and post-conditions, according to the user
Contributions

- Contract-based data structure repair [ZaeemKhurshidABZ10, ZaeemKhurshidECOOPT10, ZaeemMS10]
- **History-aware contract-based repair** [ZaeemTACAS12]
  - Read and write barriers to obtain program execution history
  - Unsatisfiable cores to obtain SAT solving history
- Repair abstraction for contract-based repair **using** Alloy [ZaeemRV13]
- Dynamic programming for structure generation
  - Test input generation [ZaeemKhurshidFSE12]
  - Ideas for repair
- Implementation
- Evaluation
  - Repair: microbenchmarks, open source projects (ANTLR and Kodkod)
  - Test generation: versus state-of-the-art tools Korat and Pex, tested Apple Safari and Google Chrome (found 3 bugs)