

assertion-based repair of structurally complex data

bassem elkarablieh and sarfraz khurshid (ut)
with ivan garcia and yuk lai suen

DaCapo meeting

1.8.7

assertion-based repair

programmers have long used assertions

an assertion violation indicates a corruption in program state

traditional approach to handle a violation:

1. terminate the program
2. debug the program (if possible) and re-execute it

at times however, terminate/debug/re-boot may not be feasible

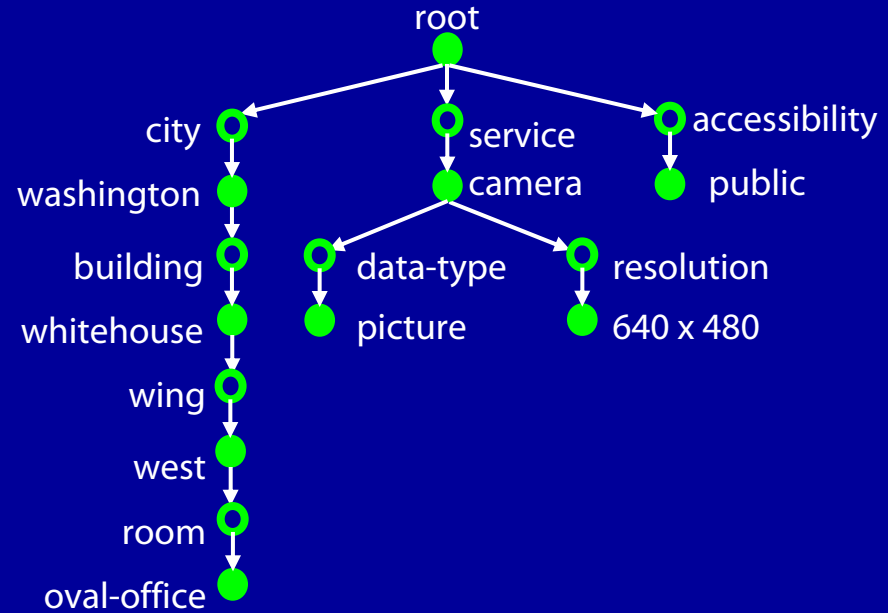
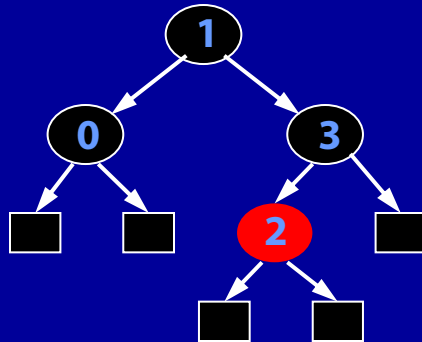
- e.g., when persistent data is corrupted

our approach to handle a violation:

1. **repair** the state of the program
2. let it continue to execute

repair tries to bring the system/data in an acceptable state
(possibly without re-booting) to continue execution

examples of structurally complex data



```
module meta_spec
sig Signature
sig Test

static sig S1 extends Test
static sig S0 extends Signature

fun Main() {}
run Main for 3
```

structural integrity constraints

violation of integrity constraints is a likely form of corruption

we can use assertions to express complex constraints

- e.g., a graph traversal that checks for acyclicity

in OO programs, **repOk** checks express class invariants

- good programming practice advocates writing repOk's

enable automated checking, e.g., via test generation

can be synthesized, even for complex structures [TACAS'07]

what does repair mean?

given a structure s and class invariant repOk where $\neg s.\text{repOk}()$,
generate a structure s' such that $s'.\text{repOk}()$ and s' is *similar* to s

- similarity is a heuristic notion
 - worst-case repair may generate a structure very different from the original one

repair does **not** aim to generate a structure that a hypothetical correct program would have computed

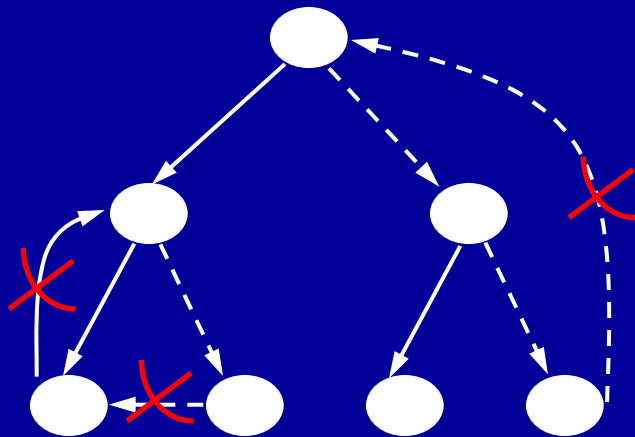
repair aims to generate a structure that is within an acceptable envelope of computation

however, repair can be precisely specified using a specification akin to post-conditions that relate pre-state with post-state

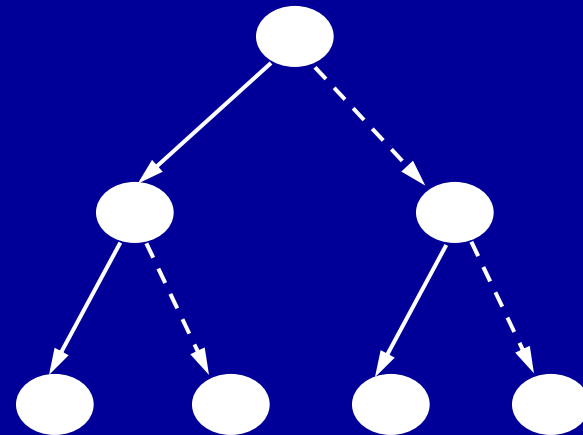
- e.g., the repaired structure must contain all data elements reachable from the root of the corrupted structure

repair example: binary tree

corrupt structure



repaired structure



our repair algorithm

uses the violated assertion as a **basis** of performing repair

- by executing repOk and monitoring the execution, we can isolate a component that is *necessarily* corrupt

systematically searches a neighborhood of the corrupt structure

is based on a **hybrid** form of symbolic execution

- treats symbolically only a dynamic subset of all object fields---the remaining fields have concrete values

performs efficient and effective repair

outline

another example

background: symbolic execution

our approach

discussion

doubly-linked circular list

```
class LinkedList {
    Entry header;           // sentinel header entry
    int size;               // number of non-sentinel entries
    static class Entry {
        Object element;    // data value
        Entry next;        // forward pointer
        Entry previous;    // backward pointer
    }

    boolean repOk() { ... }
}
```

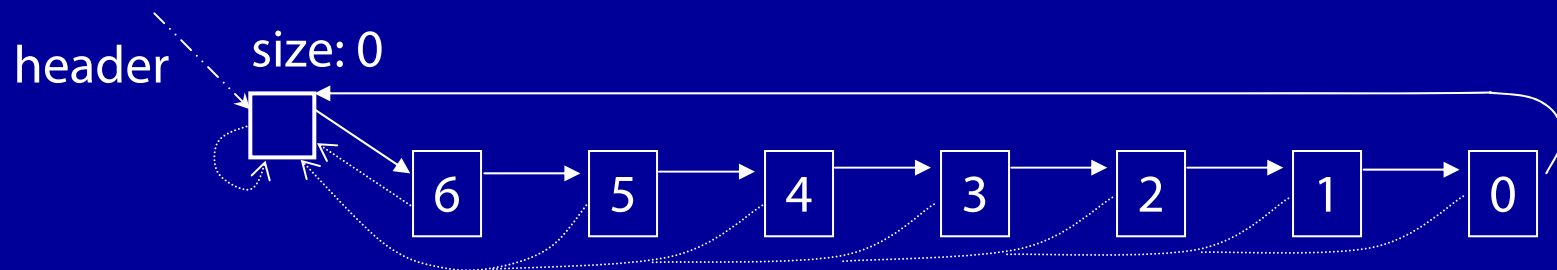
an erroneous method to add elements

```
void addFirst(Object o) {  
    Entry t = header.next;  
    Entry e = new Entry();  
    e.element = o;  
    header.next = e;  
    e.previous = header;  
    e.next = t;  
    t.previous = header; // error here  
    // error here  
}
```

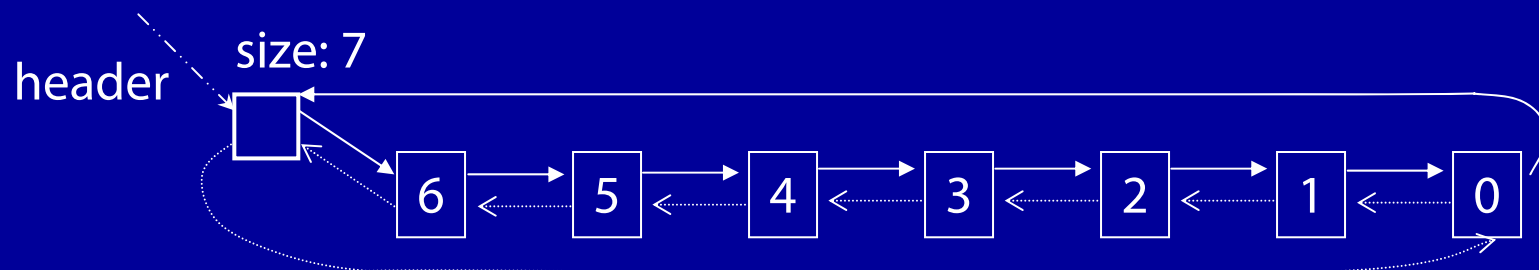
repairing to correct program behavior

consider inserting seven integer objects 0, ..., 6 into an empty list using the method `addFirst`

doing so generates:



repair corrects it to:



outline

example

background: symbolic execution

our approach

discussion

forward symbolic execution

technique for executing a program on symbolic input values

- pioneered three decades ago [boyer+75, king76]

explore program paths

- for each path, build a *path condition*
- check satisfiability of path condition

various applications

- test generation and program verification

traditional use focused on programs with fixed number of integer variables

recent generalizations work with arbitrary java/C++ programs [khurshid+03, pasareanu+04, visser+04, xie+04, csallner+05, godefroid+05, cadar+05, sen+05]

concrete execution path (example)

int x, y;	$x = 1, y = 0$
if (x > y) {	$1 >? 0$
x = x + y;	$x = 1 + 0 = 1$
y = x - y;	$y = 1 - 0 = 1$
x = x - y;	$x = 1 - 1 = 0$
if (x - y > 0)	$0 - 1 >? 0$
assert(false);	
}	

symbolic execution tree (example)

```
int x, y;
```

```
if (x > y) {
```

```
    x = x + y;
```

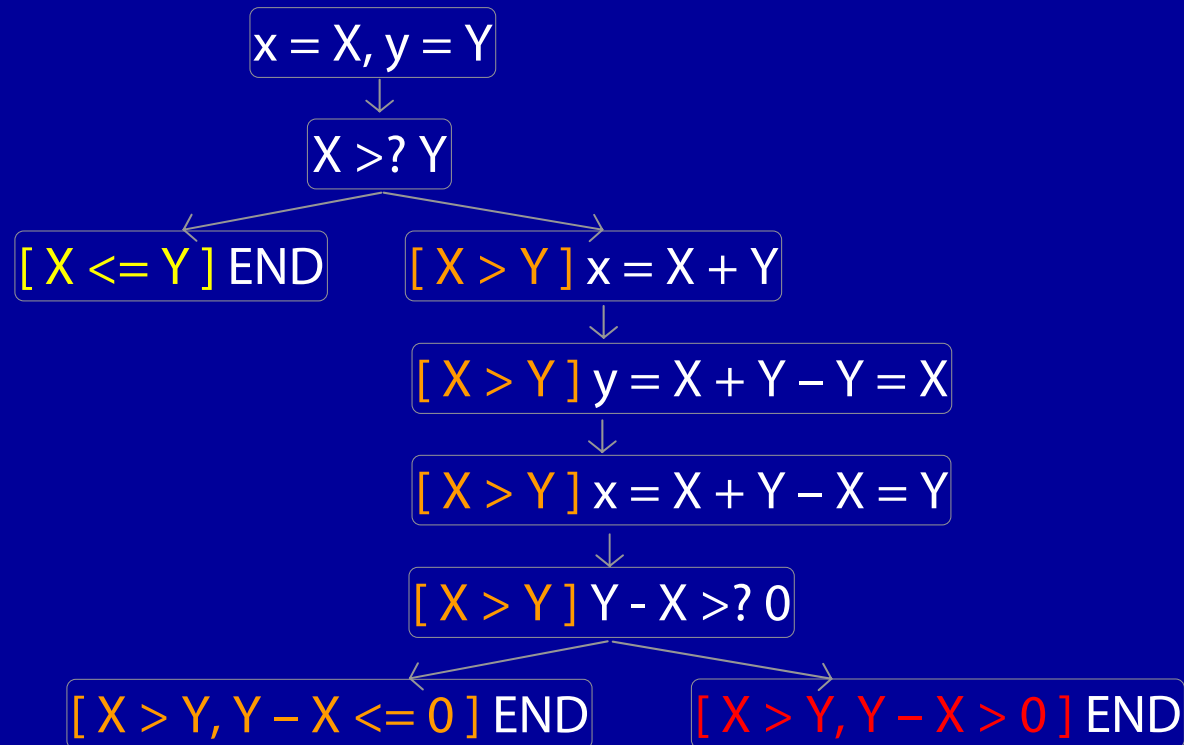
```
    y = x - y;
```

```
    x = x - y;
```

```
    if (x - y > 0)
```

```
        assert(false);
```

```
}
```



outline

example

background: symbolic execution

our approach

discussion

algorithm: outline [SPIN'05]

to repair structure s

- execute $s.repOk()$ and monitor the execution
 - note the order in which object fields in s are accessed
- when execution evaluates to false, backtrack and modify value of the **last** field accessed
 - modify the value to a new (symbolic) value that is not equal to the original one
- re-execute $repOk$

algorithm based on korat [ISSTA'02] and generalized symbolic execution [TACAS'03]

algorithm: field value update

primitive field access

- assume field f originally has value v
- assign f a symbolic value S
- add to path condition the constraint $S \neq v$

reference field access

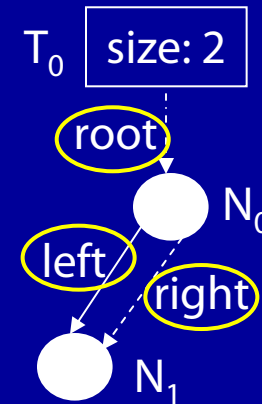
- non-deterministically assign
 - null (if original value is non-null)
 - an object of a compatible type already encountered during the current execution (if the field was not originally pointing to this object)
 - a new object (if the field was not originally pointing to an object different from those previously encountered)

illustration: binary tree

```
class BinaryTree {  
    int size;  
    Node root;  
  
    static class Node {  
        int info;  
        Node left, right;  
    }  
  
    boolean repOk { ... }  
  
}
```

example execution

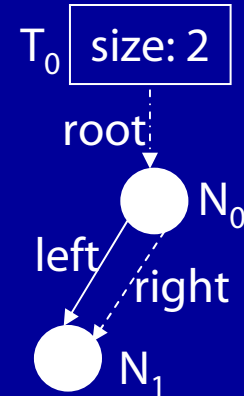
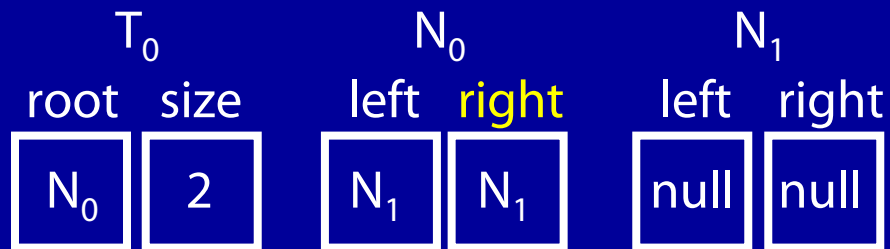
```
boolean repOk() {  
  if (root == null) return size == 0; // empty tree  
  Set visited = new HashSet();  
  List workList = new LinkedList();  
  visited.add(root);  
  workList.add(root);  
  while (!workList.isEmpty()) {  
    Node current = (Node)workList.removeFirst();  
    workList.pop_front();  
    if (current.left != null) {  
      if (!visited.add(current.left)) return false; // sharing  
      workList.add(current.left);  
    }  
    if (current.right != null) {  
      if (!visited.add(current.right)) return false; // sharing  
      workList.add(current.right);  
    }  
  }  
  if (visited.size() != size) return false; // inconsistent size  
  return true;  
}
```



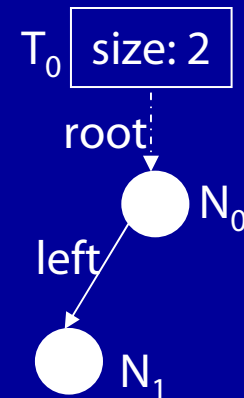
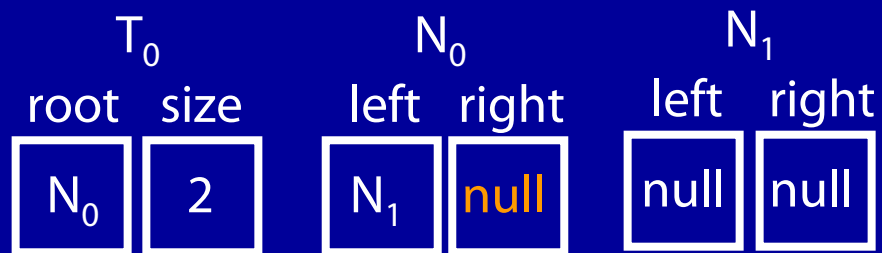
field accesses:
[T₀.root, N₀.left, N₀.right]

repair action

backtracking on [T_0 .root, N_0 .left, N_0 .right]



produces next candidate structure



- which **satisfies** repOk

implementation

written in java

has three main components

- search
 - implements systematic backtracking
- symbolic execution
 - implements library classes for hybrid symbolic execution
 - uses CVC-lite for constraint solving
- program instrumentation
 - translates java bytecode using BCEL and javassist

can handle complex structures

optimizations

efficiency

- heuristics
- static analysis
- dedicated solver

effectiveness

- preserve reachability of data values
- abstraction functions to compare pre/post repair structures

usefulness

- abstract repair log

performance

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	30				
	1000	43	not applicable	not applicable	not applicable	not applicable
	10000	138				
	100000	1466				
Doubly-linked circular list	100	18	21	24	26	25
	1000	37	39	40	42	45
	10000	101	129	165	193	220
	100000	1231	1714	2341	2944	3425
Binary tree	127	19	25	29	39	39
	1023	40	45	49	58	63
	16383	235	371	563	755	937
	131071	2141	3786	5681	7757	9660
Red-black tree	127	25	35	37	40	42
	1023	64	66	72	68	71
	16383	422	530	674	836	984
	131071	3282	4720	6197	7824	9647
Relational database	100	10	10	20	20	20
	1000	30	40	50	50	60
	10000	180	273	374	495	588
	100000	2473	4550	5730	7370	8734
Intentional name	121	8	13	22	23	29
	1093	24	30	30	33	32
	29524	258	409	581	741	913
	265720	2498	4088	5775	7849	9709
File system	156	7	10	14	20	22
	3906	28	32	40	55	61
	19531	167	284	426	571	700
	488281	3247	5822	9010	12200	15326

Table 2: Results. Times tabulated are in milliseconds. For all subjects, repairing ≤ 10 corrupt fields takes on average: < 1 second for ≤ 10000 nodes; < 10 seconds for ≤ 100000 nodes.

outline

example

background: symbolic execution

our approach

discussion

applicability: how hard is it to write assertions?

any technique for repair has a cost, e.g., the cost of writing a repair routine correctly

assertion-based repair has minimal cost

- assertions are written in the programming language
- assertion describes *what*; repair routine describes *how*
- properties are known at time of implementation; efficient repair routines are not
 - e.g., red-black tree invariants are well-known, while there are no text-book algorithms to repair them
- assertions may already be present in code
 - e.g., due to systematic testing or defensive programming

scalability: how efficient can repair be?

repair considers the problem of generating one (large) structure
korat [ISSTA'02], TestEra [ASE'01] show feasibility of exhaustive
generation of a large number of small structures

results from analogous SAT problems indicate repair should be
easier than exhaustive generation

- finding one solution is easier than model counting [Wei+05]
- moreover, w.h.p. we expect the repaired structure to lie
in a close neighborhood of the corrupt structure
 - repair is therefore analogous to finding one solution
to a SAT formula that is satisfiable w.h.p.
 - local search is expected to work well [Hoos99]

cost of repair seems to scale linearly with structure size

novel applications of repair

generating large test inputs

- idea: repair a randomly generated large graph
- generates structures 100x larger than those with korat

repairing programs [ghori'06]

- idea: translate repair actions in code that performs repair

related work

fault-tolerance and error recovery have featured in software systems for a long time

most of the past work has been on specialized repair routines

- file system utilities, such as fsck
- commercial systems, such as IBM MVS operating system and Lucent 5ESS switch

demsky and rinard's constraint-based framework [OOPSLA'03]

- declarative constraints define desired structures
- mapping defines data translations
- repair is ad hoc
- requires users to provide mappings and learn a new constraint language

conclusion

a new view of assertions

- a non-conventional application of backtracking search

a unified framework for verification and resilience

- systematic testing before deployment
- systematic repair once deployed

assert-first programming

- assertions are immensely popular in hardware verification; the time has also come that we realize the potential benefits assertions have long offered in software

?/!

khurshid@ece.utexas.edu
<http://www.ece.utexas.edu/~khurshid>