assertion-driven analyses

from compile-time checking to runtime error recovery

sarfraz khurshid
the university of texas at austin

state of the art in software testing and analysis day, 2008
rutgers university

overview

programmers have long used assertions

- runtime checks
- documentation

assertions **are** lightweight specifications

- written using the underlying programming language

we envision a much broader use of assertions

- developers assert designs
- static analyses check conformance to designs
- systematic approaches test executable code
- runtime checks monitor for erroneous executions
- error recovery repairs as desired

assertion-based repair [elkarablieh et al ASE'07]

an assertion violation indicates a corrupt program state

traditional approach to handle an assertion violation:

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

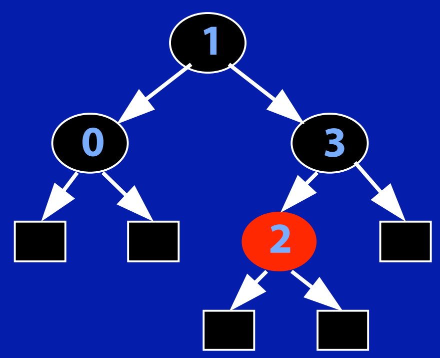
at times however, terminate/debug/re-boot may not be feasible, e.g., when persistent data is corrupted

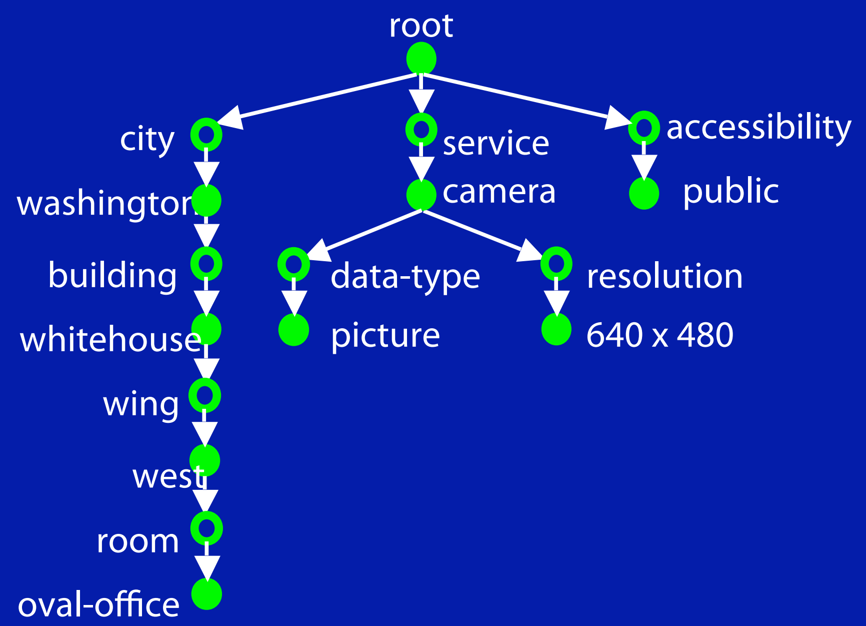
our approach to handle a violation:

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





structural integrity constraints

violation of integrity constraints is a likely form of corruption

assertions readily express complex constraints

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

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

- good programming practice advocates writing repOk's

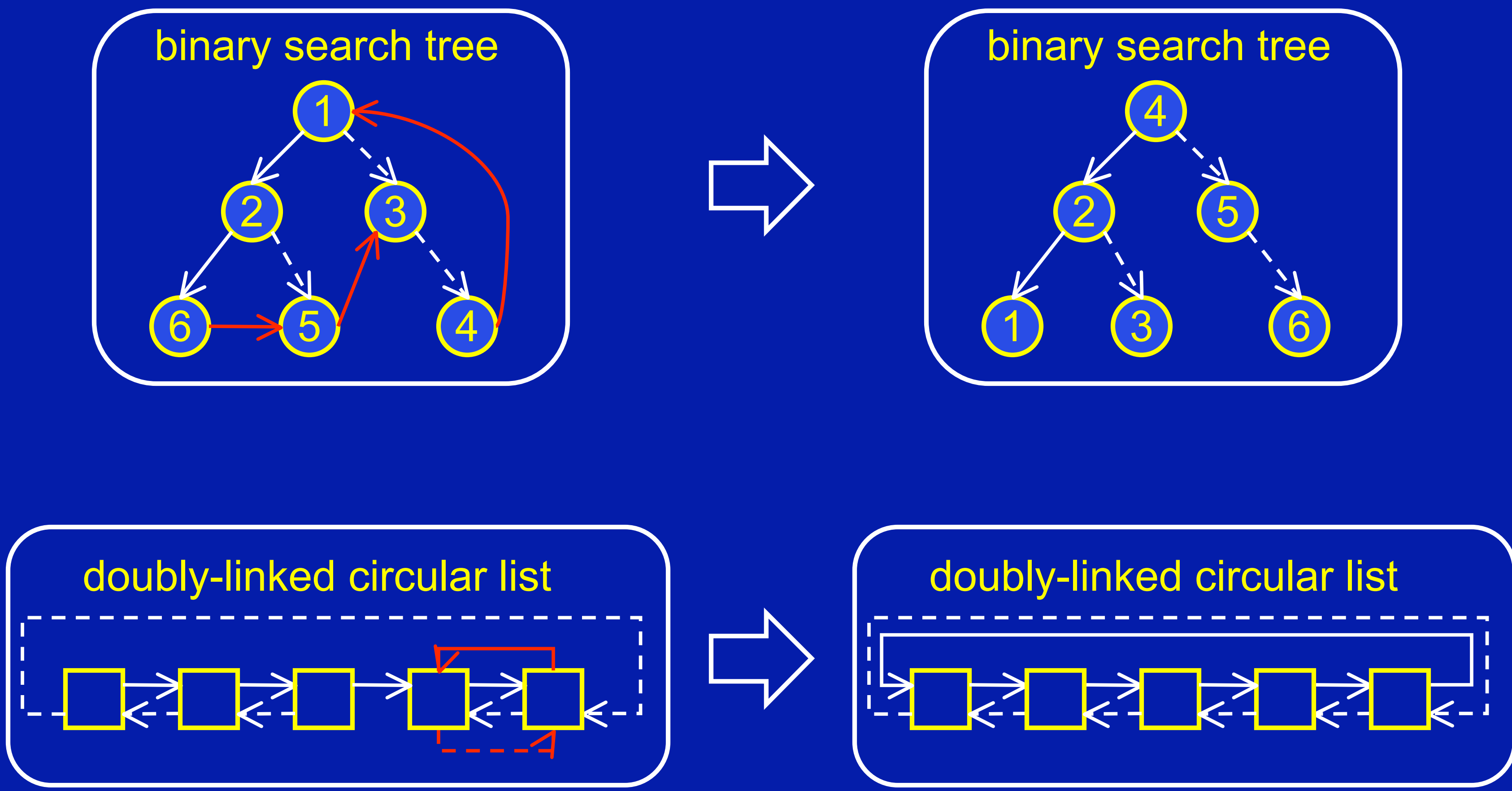
enable automated checking, e.g., via test generation

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

repair examples

corrupt

repaired



what does repair mean?

given a structure s and a repOk where !s.repOk(), generate s' such that s'.repOk() and s' is *similar* to s

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

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

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

can be specified using a specification (cf. postconditions)

- e.g., the repaired structure contains all data elements reachable from the root of the corrupt structure

overview of our repair algorithm

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

- executes repOk and monitors its execution to isolate a component that is *necessarily* corrupt

systematically searches a neighborhood of the corrupt structure

uses 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

overview

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

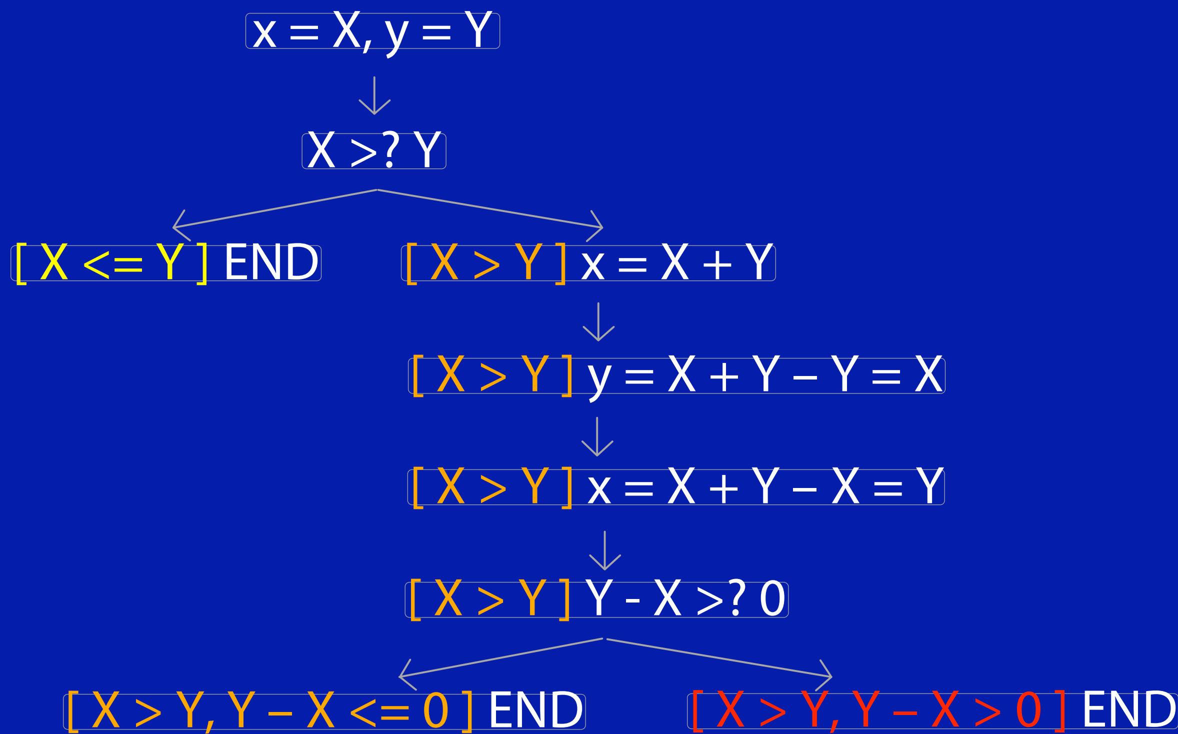
recent generalizations handle more general java/C++ code [khurshid+03, pasareanu+04, visser+04, xie+04, csallner+05, godefroid+05, cadar+05, sen+05]

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

overview

background: symbolic execution

our approach

discussion

algorithm: outline

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 the 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

- assume field f originally has value v
- assign f a symbolic value S
- add to path condition the constraint S != v

reference field

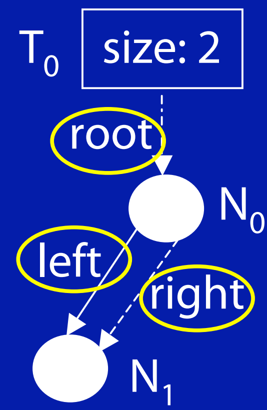
- non-deterministically assign
  - null (if original value is non-null)
  - an object of a compatible type already encountered during the current execution (if the field was not originally pointing to this object)
  - a new object (if the field was not originally pointing to an object different from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

illustration: binary tree

```
class BinaryTree {
   int size;
   Node root;

   static class Node {
      int info;
      Node left, right;
   }

   boolean repOk() { ... }

   void add(int e) {
      assert repOk();
      ...
   }
}
```

example execution

```
boolean repOk() {
  if (root == null) return size == 0; // empty tree

  Set visited = new HashSet();
  LinkedList workList = new LinkedList();
  visited.add(root);
  workList.add(root);
  while (!workList.isEmpty()) {
    Node current = (Node)workList.removeFirst();
    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;
}
```

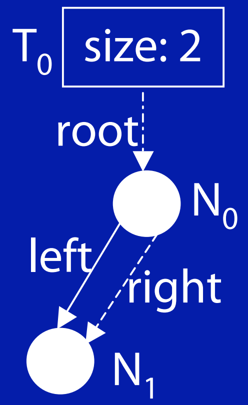


field accesses:
[T_0.root, N_0.left, N_0.right]

repair action

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

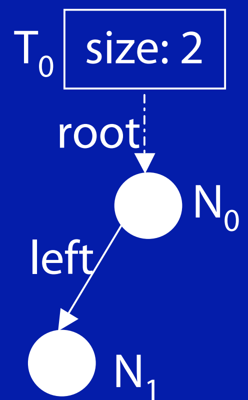
T_0 root	T_0 size	N_0 left	N_0 right	N_1 left	N_1 right
N_0	2	N_1	N_1	null	null



produces next candidate structure

T_0 root	T_0 size	N_0 left	N_0 right	N_1 left	N_1 right
N_0	2	N_1	null	null	null

- 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

effectiveness

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

usefulness

- abstract repair log

performance

evaluated on a suite of text-book data structures

- singly/doubly-linked lists, binary search trees, etc.

for a small number of faults (<= 10), algorithm can repair structures with a few hundred nodes in less than 10 sec

does not scale to large data structures

- but we are working on several optimizations

outline

overview

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 but efficient repair routines may not be
  - e.g., red-black tree invariants are well-known but 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]

our recent work

static analysis for repair [OOPSLA 2007]

- uses cahoon and mckinley's recurrent field analysis
- prioritizes repair actions based on whether a field recurs
- enables repair of larger structures

constraint-based generation of large test inputs [ECOOP 2007]

- repairs randomly generated object graphs of a desired size
- enables efficient generation of larger test inputs

repairing programs [UT-TR 2006]

- translates 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]

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

summary of assertion-based repair

a novel view of assertions

- use violated assertions as basis for repair

an algorithm for repair using symbolic execution

- a non-conventional application of backtracking search

still not practical for very large structures in deployed systems

opens a promising direction for future work

- a unified framework for verification and error recovery
  - systematic testing before deployment
  - systematic repair once deployed

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