# specification-based program repair using SAT

sarfraz khurshid
joint work with: divya gopinath and muhammad zubair malik
university of texas at austin
khurshid@ece.utexas.edu

AFRL’s S5
6.16.11

# overview

removing bugs in code is tedious and error-prone – even when location of a fault is known

- particularly hard for programs that perform destructive updates on complex, dynamically-allocated structures

this talk presents a novel specification-based approach for automated debugging

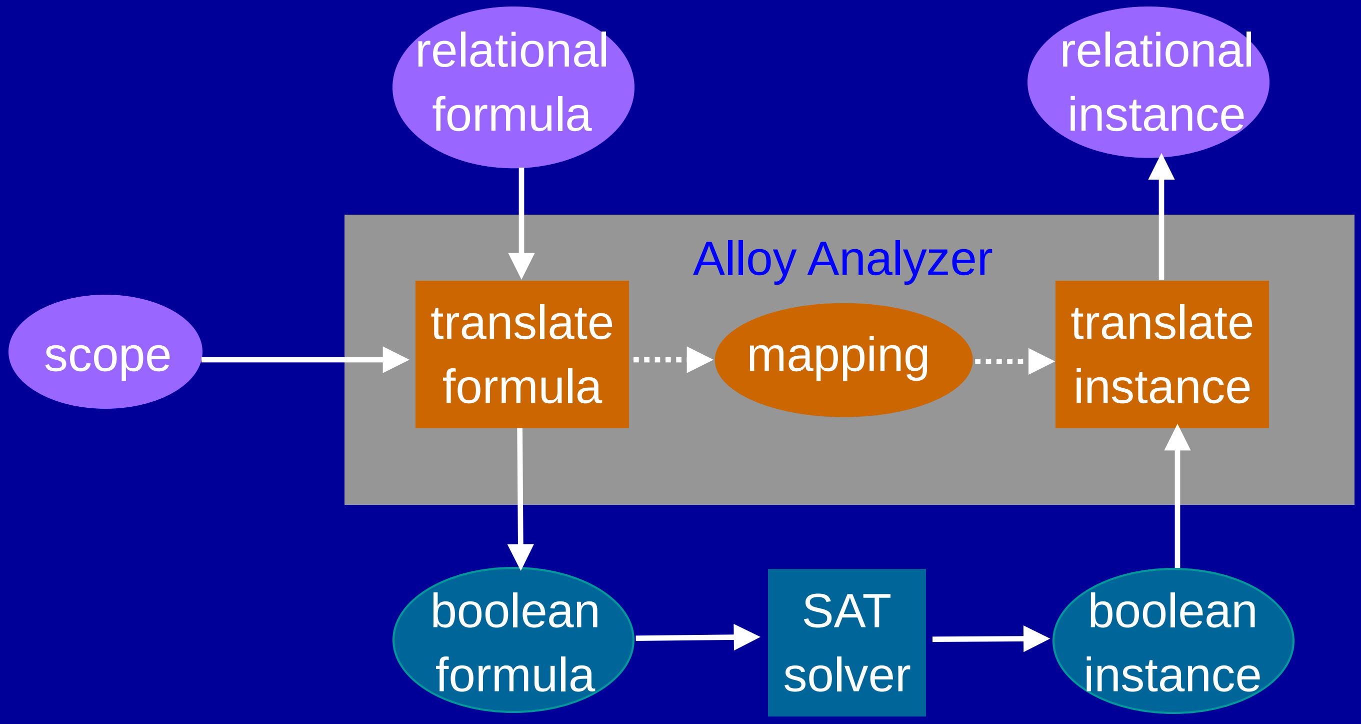
- the alloy tool-set provides an enabling technology
  - pre/post-conditions in alloy describe expected behavior
  - SAT provides an analysis engine

our insight is to replace a faulty, deterministic statement with a non-deterministic one that represents a class of similar operations

- prune non-determinism using alloy/SAT

experiments show our approach holds promise

# enabling technology: alloy [jackson'00]

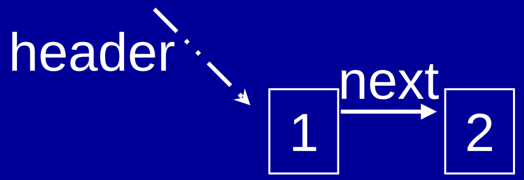


# outline

- overview
- example
- approach
- experiments
- conclusion

# example: singly-linked, sorted list

```
class SLList {
    Node header;
    static class Node {
        Node next;
        int elem;
    }
```

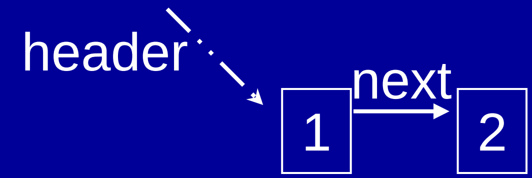



```
    // class invariant
    // acyclic
    all n: header.*next | n !in n.^next
    // sorted
    all n: header.*next | some n.next => n.elem < n.next.elem
```

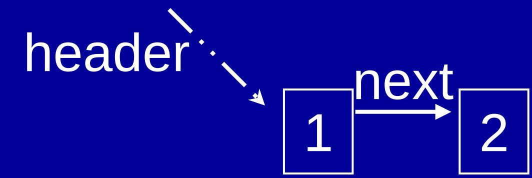
# example: (faulty) delete method

```
// post-condition
header.*next.elem – v = header'.*next'.elem'
```

```
void delete(int v) {
    Node prev = null;
    Node l = header;
    while (l != null) {
        if (l.elem == v) {
            if (prev != null) {
                prev.next = l; // Error
            } else
                header = header.next;
            return;
        }
        else {
            prev = l;
            l = l.next;
        }
    }
}
```




**delete(2);**



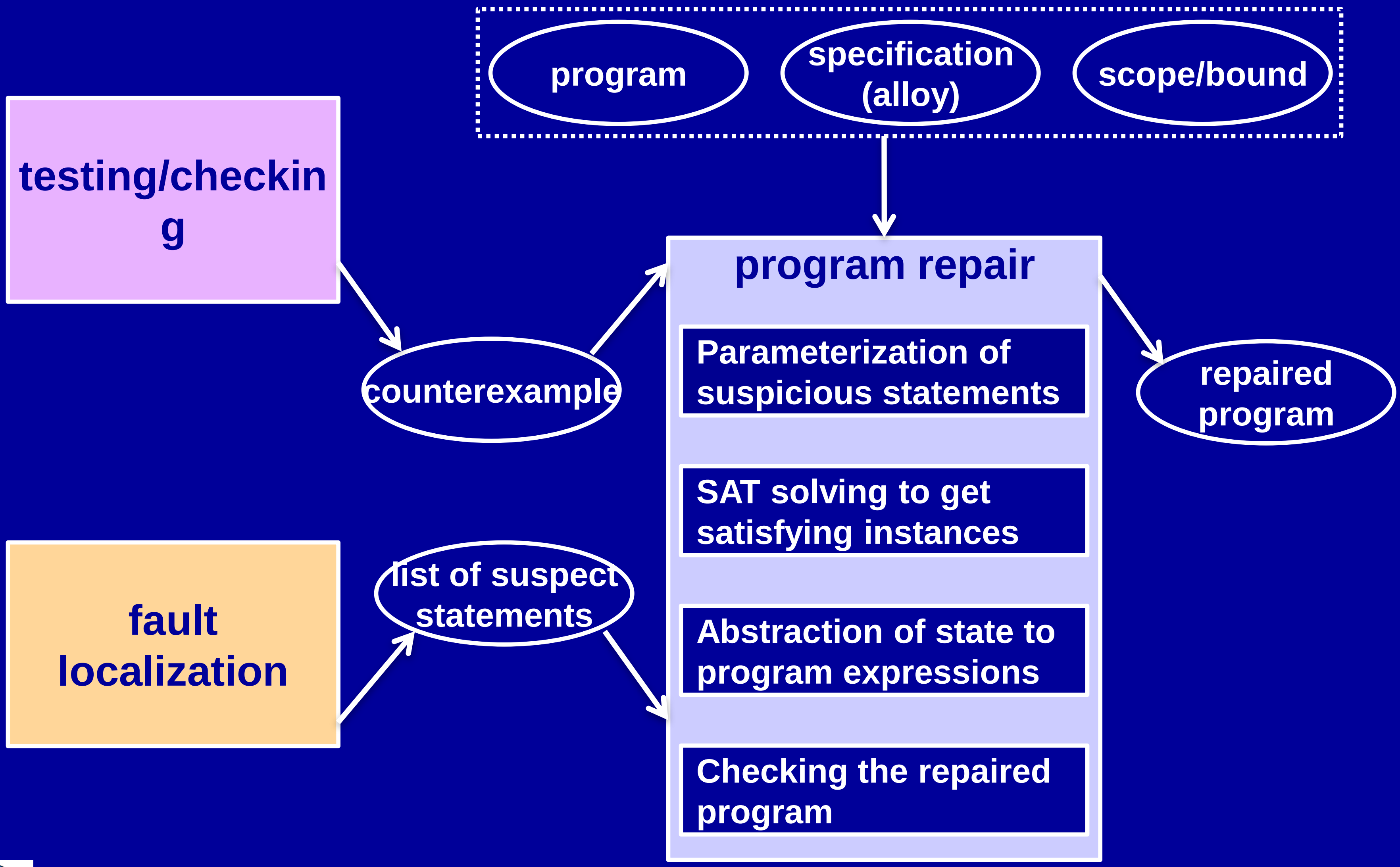

**prev.next = l.next; // OK**

**delete(2);**

# outline

- overview
- example
- approach
- experiments
- conclusion

# our framework: overview



# our framework: inputs

counterexample

- generated using SAT-based bounded exhaustive checking
  - relational encoding of the formula: *pre && code && !post*


SLList = { $L_0$ }, Node = { $N_0$, $N_1$ }, Integer = { -4, ..., 3 },
this = { $L_0$ },
header = { $\langle L_0, N_0 \rangle$ }, next = { $\langle N_0, N_1 \rangle$ },
elem = { $\langle N_0, 1 \rangle$, $\langle N_1, 2 \rangle$ },
header’ = header, next’ = next, elem’ = elem

list of suspect faulty statements

- identified using a fault localization tool or manually
  - statement 6: “prev.next = l;”

# our framework: parameterization of suspicious statements

replace expressions with fresh variables that take non-deterministic values from appropriate domains

- “$e_1$.f = $e_2$;” replaced with “$v_1$ in D; $v_2$ in D; $v_1$.f = $v_2$;”
- “v = e;” replaced with “$v_1$ in D; v = $v_1$;”
- “if (x op y) ...” replaced with “$v_1$ in D; $v_2$ in D; if ($v_1$ op $v_2$) ...”
  - similarly for conditions in other statements

in our example, “prev.next = l;” replaced with “$v_1$ in { null, $N_0$, $N_1$ }; $v_2$ in { null, $N_0$, $N_1$ }; $v_1$.next = $v_2$;”

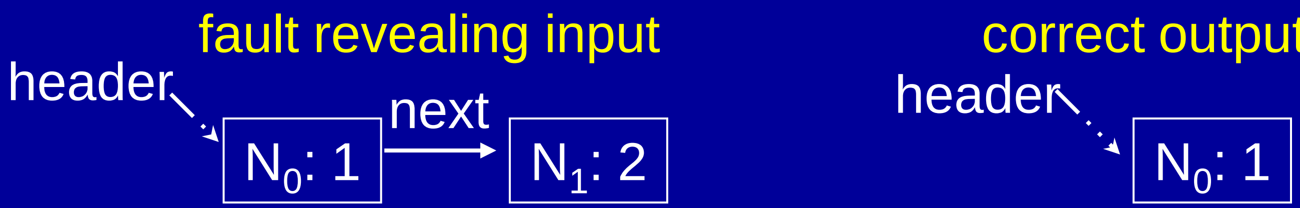
we ignore errors of omission and errors in operators or constants

# our framework:
# SAT solving to get satisfying instances

use the input from the counterexample and the program with parameterized statements to generate a correct execution

- solve the formula: *$input_{fault\text{-}revealing}$ && $code_{parameterized}$ && post*

in our example, we get:




and the valuation for correct execution has $v_1 = \{ N_0 \}$, $v_2 = \{ null \}$ for assignment statement "$v_1.next = v_2$;"

# our framework: abstraction of state to program expressions

abstract the values of fresh variables to program expressions

- "*v*" or "$v.f_1. \ldots . f_n$", where "*v*" is a program variable and $f_1, \ldots, f_n$ are fields

in our example, at statement 6:

$N_0$ abstracts to one of { prev, this.header }

null abstracts to one of { l.next, prev.next.next,
this.header.next.next }

# our framework: checking the repaired program

systematically replace fresh variables with appropriate expressions and perform bounded exhaustive checking (not just for one input)

in our example, “prev.next = l;” can be transformed to “header.next = l.next;”

- but bounded verification finds counterexample with 3 nodes

>1 transformations may generate correct repair

- e.g., “prev.next = l.next;” and “prev.next = prev.next.next;” are both correct

# outline

overview

example

approach

experiments

conclusion

# setup

our prototype uses the forge framework [Dennis+ISSTA'06] of the alloy tool-set for program repair as well as bounded checking

- code and spec encoded in forge intermediate language
- miniSAT
- works with program's bounded computation graph

subject programs

- "insert" method of binary search tree
- "addChild" method of ANother Tool for Language Recognition

manually seeded faults in these methods

manually set bounds on input size and loop unrollings

manually provided list of suspect statements – different scenarios

metrics

- efficiency – total repair time, # of SAT calls
- accuracy – fix quality (semantic equivalence to correct code)

# scenarios

fault injection

- #faults <= 4
- commission
  - field assignment statement
  - local variable update
  - “if-else” condition/body
  - “while” condition/body
- omission

fault localization

- initial suspect list equals the list of faulty statements (Scr#1)
- suspect list additionally contains non-faulty statements (Scr#2)
- initial suspect list does not contain all faulty statements and may contain some non-faulty statements (Scr#3)

# scenarios: example [Error#10 in BST.insert]

fault injection

```
...
if (x == null) //FIX: if (y == null)
   t.root = x;
else {
   if (k < y.key)
      y.left = y; //FIX: y.left = x;
   else
      y.right = y; //FIX: y.right = x;
}
y.parent = x; //FIX: x.parent = y;
...
```

fault localization

- initial counterexample is empty tree which leads to list of one suspicious statement { “if (x == null)” }
- next counterexample leads to { “y.right = y;”, “y.parent = x;” }
- next counterexample leads to { “y.left = y;” }

# results

| Name | Scr# | Error# | FL Scheme Output | | Type of Stmts | Repair Time(secs) | # SAT Calls | Accuracy |
|---|---|---|---|---|---|---|---|---|
| | | | # Faulty | # Correct | | | | |
| BST | 1 | 1 | 1 | 0 | Assign Stmt | 3 | 2 | √, Same |
| | | 2a | 1 | 0 | Branch stmt | 34 | 114 | √, Diff |
| | | 2b | 1 | 0 | Branch stmt | 4 | 2 | √, Same |
| | | 3a | 1 | 0 | Assign stmt | 5 | 2 | √, Diff |
| | | 3b | 1 | 0 | Assign stmt | 5 | 4 | √, Same |
| | | 4a | 1 | 0 | Branch stmt | 12 | 96 | √, Diff |
| | | 4b | 1 | 0 | Branch stmt | 4 | 2 | √, Same |
| | | 4c | 1 | 0 | Loop condition | 1 | 2 | √, Same |
| | | 5 | 2 | 0 | Branch, Assign stmts | 7 | 5 | √, Same |
| | | 6 | 2 | 0 | Assign stmts | 5 | 3 | √, Same |
| | 2 | 7 | 1 | 2 | Branch, Assign stmts | 15 | 21 | √, Same |
| | | 8 | 2 | 1 | Branch, Assign stmts | 6 | 2 | √, Same |
| | | 9 | 1 | 1 | Assign stmts | 11 | 2 | √, Same |
| | 3 | 10 | 4 | 0 | Branch, Assign stmts | 6 | 8 | √, Same |
| | | 11 | 2 | 0 | Branch, Assign stmts | 26 | 9 | √, Same |
| | | 12 | 2 | 1 | Branch, Assign stmts | 33 | 14 | √, Same |
| | | 13 | 2 | 1 | Assign, Branch stmts | 14 | 24 | √, Same |
| | | 14 | 0 | 2 | Omission error | NA | NA | NA |
| ANTLR | 1 | 1 | 2 | 0 | Assign Stmt | 71 | 2 | √, Diff |
| | 2 | 2 | 2 | 2 | Branch, Assign stmts | 1 | 5 | √, Same |

# outline

overview

example

approach

experiments

conclusion

# related work

program repair as a game
[jobstmann+CAV'05]

sketching for program synthesis using SAT
[solar-lezamaAPLAS'09]

machine learning-based tool for fixing bugs
[jeffrey+ICPC'09]

genetic programming for finding patches
[weimer+ICSE'09]

program repair using mutation
[debroy+ICST'10]

fixing of programs with contracts
[wei+ISSTA'10]

# our previous work on repair

program repair using data structure repair [**UT-MS’06**, **ASE’09**, **ICST’11**]

- generate java statements that abstract concrete repair actions
  - <$N_0$, previous, $N_1$> → “newEntry.next.previous = newEntry;”

data structure repair using systematic constraint solving

- assertion-based repair [**SPIN’05**, **ASE’07**, **OOPSLA’07**, **ECOOP’07**, **ICSE$_d$’08**, **ISSTA’08**, **UT-PhD’09**]
  - assertion describes expected properties at a control point, e.g., class invariant, such as “assert repOk();”
  - systematic search of a bounded neighborhood of the erroneous state generates a repaired state
- contract-based repair [**ABZ’10**, **ECOOP’10**, **UT-MS’10**]
  - alloy post-conditions relate pre-state and post-state
  - repair algorithms iteratively modify field values

# our ongoing work

further develop core algorithms for spec-based program repair

- handle more general errors of commission, e.g., incorrect operators or method invocations
- reduce burden of writing specs
  - our insight: enable writing specs using *mixed* constraints
- handle errors of omission
  - our insight: synthesize code from violated parts of spec

use program repair to optimize on-the-fly data structure repair

- our insight: abstract concrete repair actions into "program statements" that are "executed" to repair future errors

# ? & //

this talk presents a novel specification-based approach to program repair using alloy/SAT [Gopinath+TACAS'11]

- transform faulty statement into a non-deterministic statement and use SAT to prune non-determinism

our project lays the foundation for using rich behavioral specs as a basis of program repair

it forms a part of our wider effort on constraint-based development and analyses

- specs are one form of constraints – at implementation level
- constraints may be at a higher level e.g., to describe requirements, architecture, design, or even tests/analyses

it provides a basis for new reliability methodologies that apply traditionally different approaches in synergy

khurshid@ece.utexas.edu