a constraint-driven approach for systematic and integrated testing and error recovery

sarfraz khurshid
university of texas at austin
khurshid@ece.utexas.edu

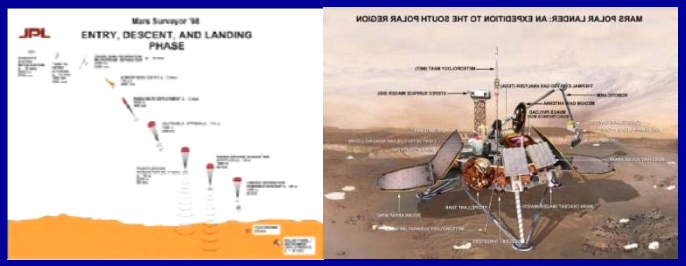
AFRL/RBCC's S5
6.16.10

software failures are expensive

software failures are expensive

- the cost is particularly high for mission-critical systems

Photos: *ESA/CNES*



Photos: *JPL/NASA*

Photo: *navsource.org*

NIST 2002: software flaws cost the US economy $60B/yr

- better testing infrastructure could save $20B/yr

software checking

existing techniques are expensive, ad hoc, and ineffective

- limited to checking weak properties
- do not scale to real-world applications

testing is the most commonly used technique for validating quality of software

- accounts for more than a half of the development cost
- requires manual generation of test inputs
- fails to find bugs

there is an urgent need to develop new methodologies

our approach [funded in part by AFOSR, NSF]

provides automated, systematic checking using **constraints**

applies during various phases of software development

- checks properties of designs
- checks properties of implementations
- monitors and repairs executions of deployed systems
  - enables error recovery

supports checking of rich correctness properties

- e.g., conformance of code to designs

uses static and dynamic analyses, in synergy

- leverages state-of-the-art constraint solvers

2010: ASE, ABZ, ECOOP, ICSE (distinguished paper), TSE

2009: ASE, ASWEC (best paper), FM, FSE, ICFEM, ICST

2008: APLAS, ASE, AST, ATGSE, FM, ICSE, ISSTA, ISSRE

2007: ACL2, ASE, ECOOP, FMCAD, FSE, ICSE, ICSM, iFM, LCSD, OOPSLA, STEP, TACAS, TAIC-PART

core technologies

constraints and solving them lie at the heart of our approach

we have developed several new, efficient algorithms

- korat
  - used at microsoft, google
- abstract symbolic execution
  - abstracts from details of library implementations
  - treats a handful of fields symbolically

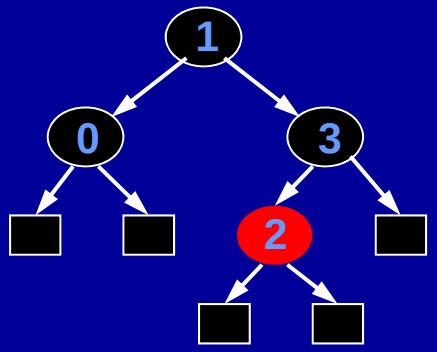
to further increase efficiency, we have developed parallel algorithms for dynamic checking

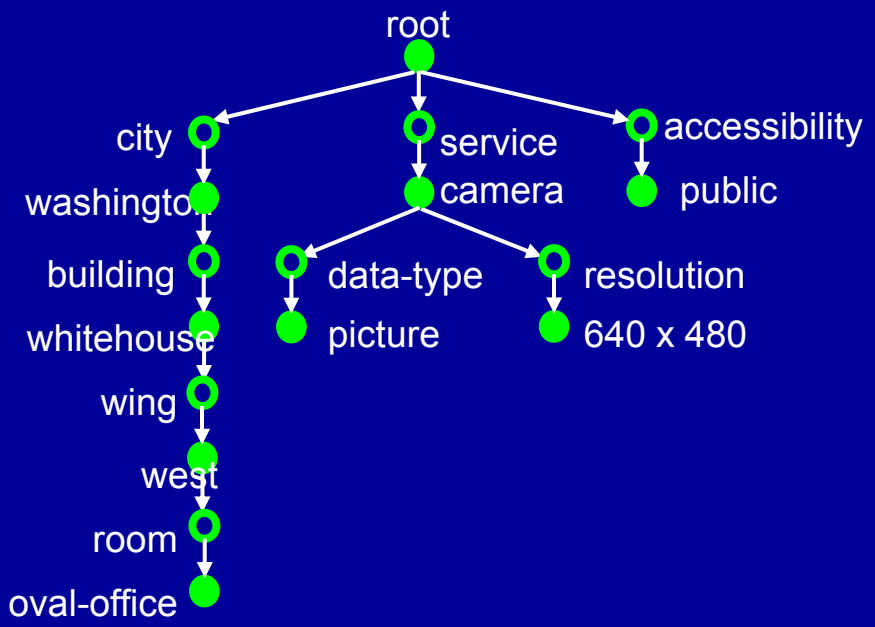
the algorithms enable test automation as well as error recovery using the same constraints

structurally complex data

a focus of our approach is data intensive properties

- e.g., structural constraints of heap-allocated data





outline

overview

example

constraint-driven testing

constraint-driven error recovery

conclusion

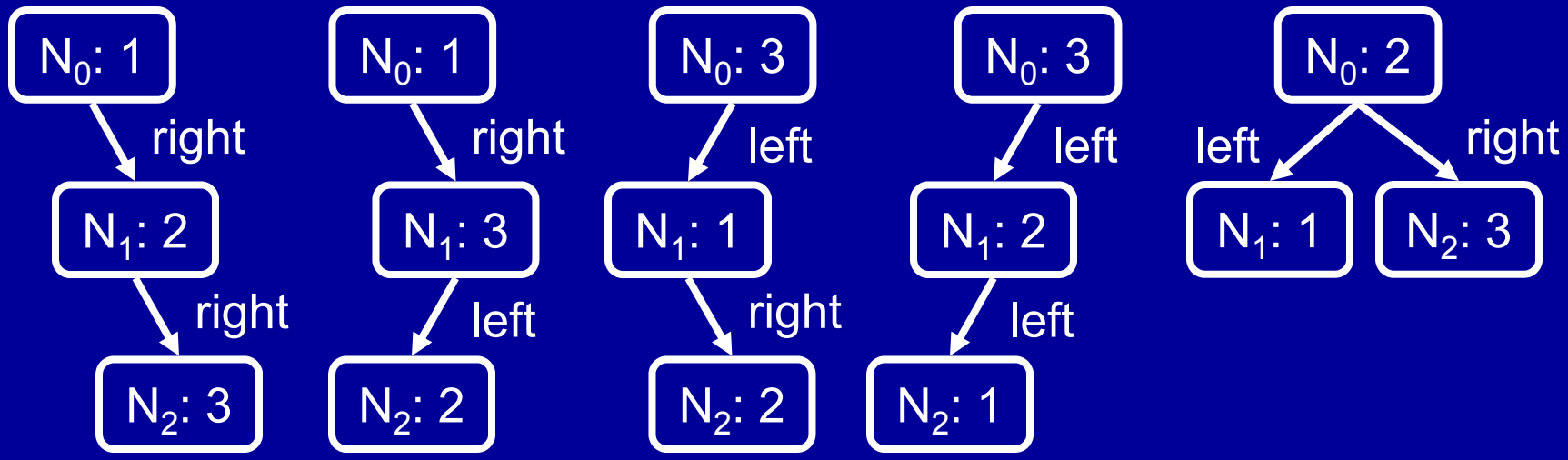
example: binary search tree

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

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

  void remove(int i) { … }
}
```

input constraint: isTree() && isOrdered()

oracle constraint: isTree() && isOrdered() && “removes only i”

non-isomorphic trees with 3 nodes



example: declarative constraints

input constraint

```
boolean isTree() {
  # root.*(left + right) = size   // consistency of size
  all n: root.*(left + right) {
    n !in n.^(left + right)       // no directed cycles
    sole n.~(left + right)        // at most one parent
    no n.left & n.right } }       // left and right child not the same node

boolean isOrdered() { ... }       // binary search
```

oracle constraint

...

example: imperative constraints

```
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();
        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;
}
```

outline

overview

example

constraint-driven testing

constraint-driven error recovery

conclusion

constraint-driven test generation

constraints define properties of desired inputs

- can characterize test purposes, selection criteria etc.
- easier to formulate than a high quality test suite

korat/SAT solvers provide automatic input generation

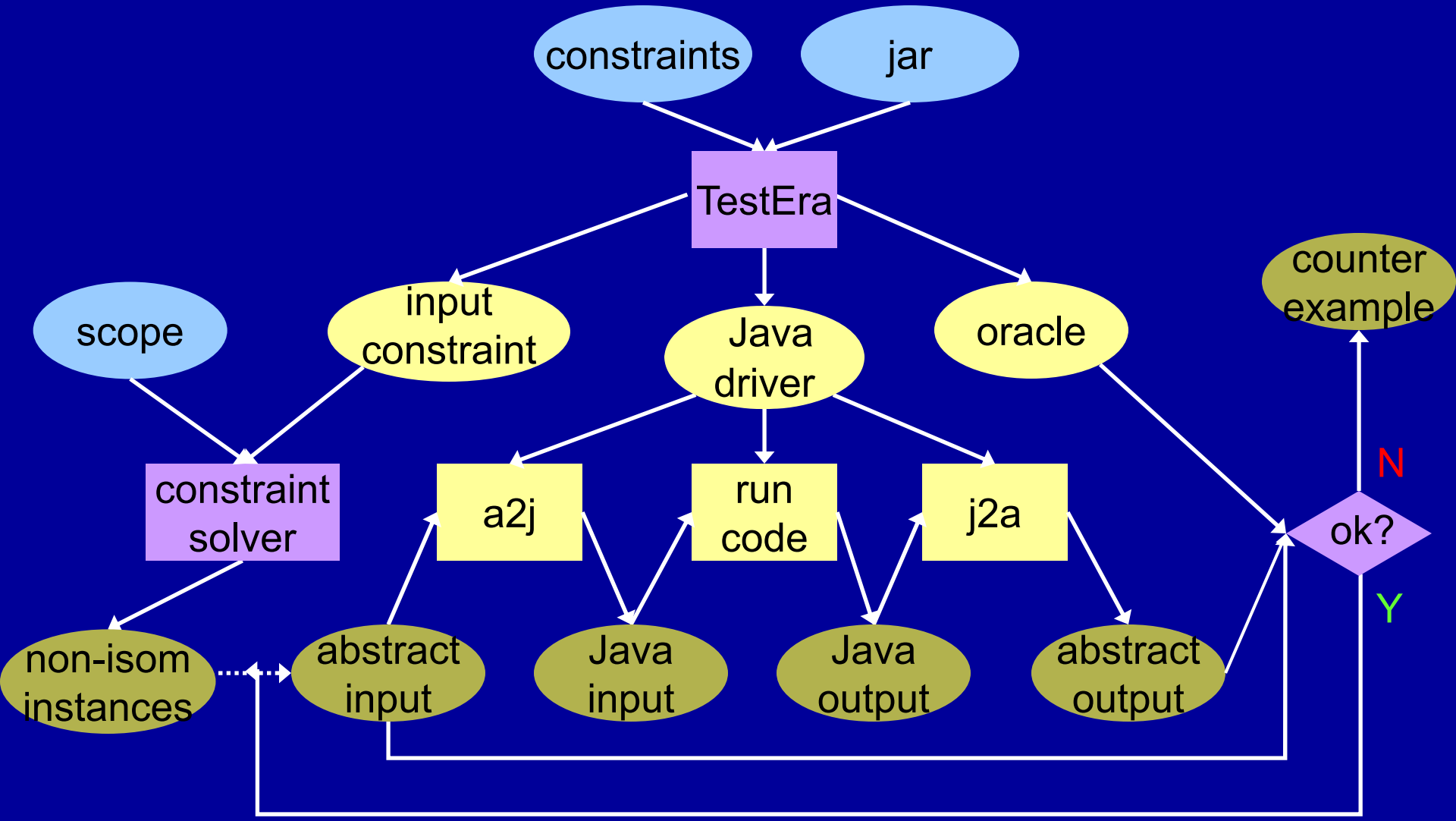
test suites have no redundancy – inputs are non-equivalent

testing is systematic (aka “scope bounded”)

- test against **all** inputs within a bound on the input size
- inspired by model checking (“small scope hypothesis”)

has been applied successfully in both academia and industry

TestEra architecture



test generation performance: SAT/mChaff

benchmark	size	structures* generated	time (sec)	state space
BinarySearchTree	7	429	7	2^{186}
	9	4862	618	2^{292}
HeapArray	6	13139	6	2^{72}
	8	1005075	1172	2^{110}
java.util.LinkedList	7	877	1	2^{191}
	10	115975	304	2^{362}
java.util.TreeMap	7	35	111	2^{263}
	9	122	742	2^{407}
java.util.HashSet	7	1716	32	2^{119}
	9	24310	512	2^{215}

* *Sloane's online encyclopedia of integer sequences*

outline

overview

example

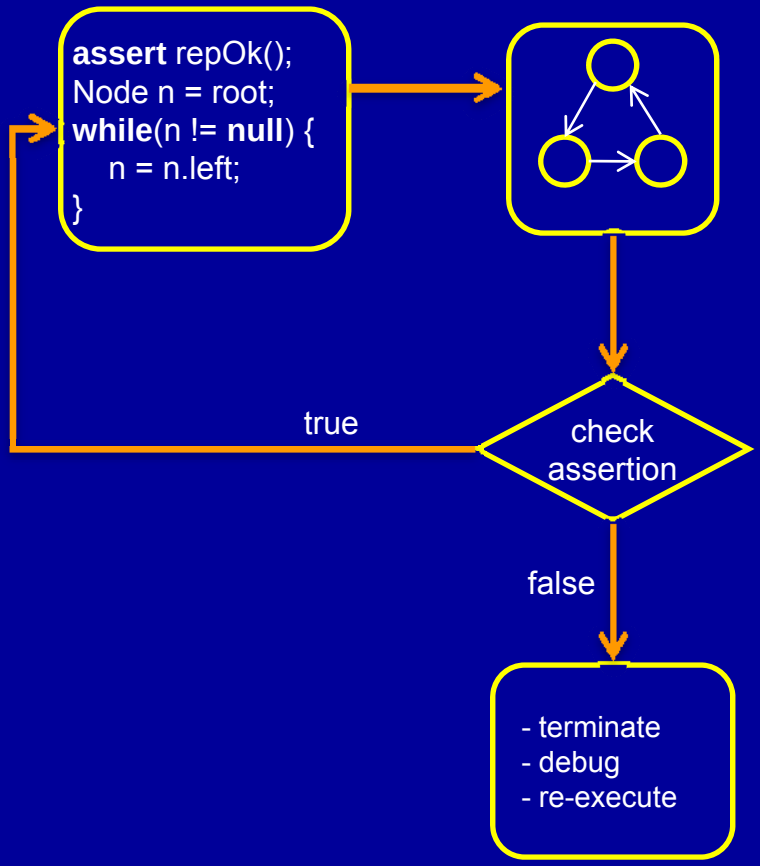
constraint-driven testing

constraint-driven error recovery

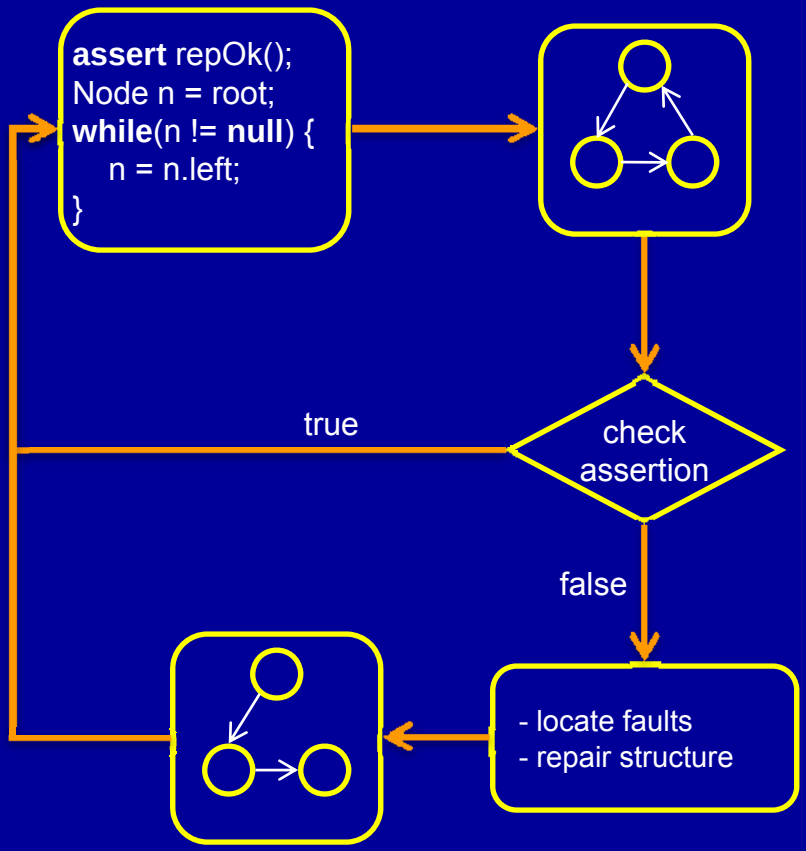
conclusion

error recovery using data structure repair

traditional approach



repair-based approach



constraint-driven error recovery

constraints that define rich behavioral specifications, e.g., Alloy or JML specs, precisely describe expected behavior

such a specification defines a whole class of implementations

in fact, a spec itself often represents an implementation, albeit one that has a high degree of non-determinism

but directly executing a spec is likely to be highly inefficient

however, a partially correct implementation enables pruning some of the spec's non-determinism

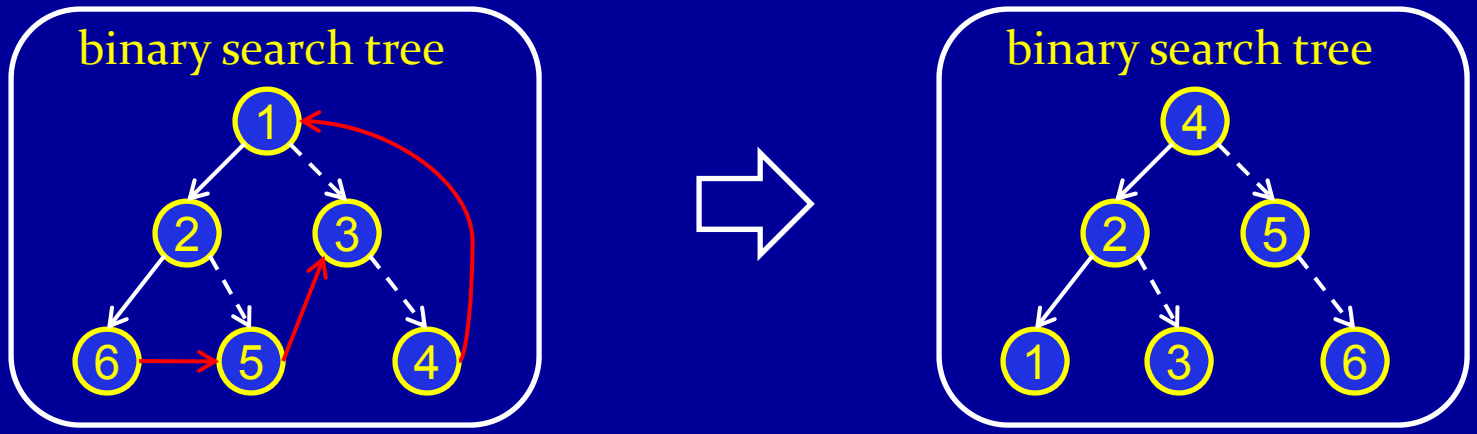
- preserve correct program actions (state mutations)
- repair incorrect actions with respect to the spec

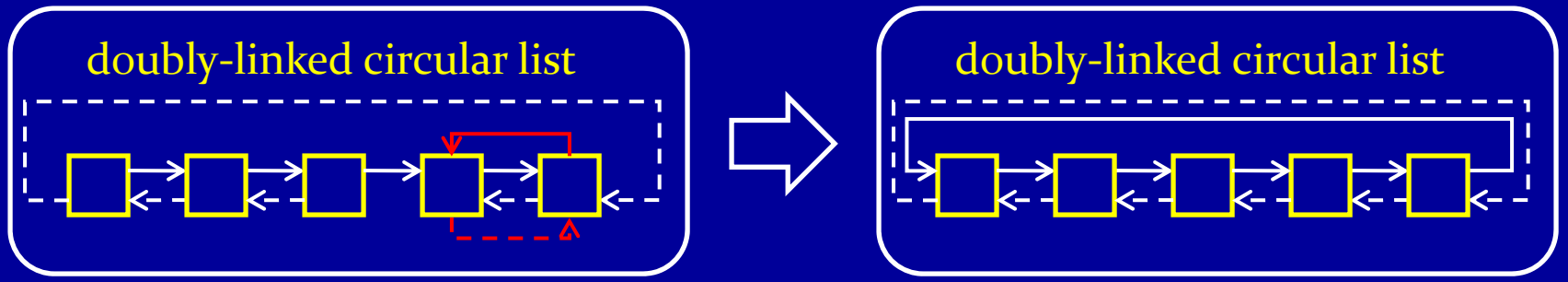
thus, specs serve as a basis for runtime error recovery

repair examples

corrupt

repaired





our initial work on data structure repair

our initial work focused on assertions – lightweight specs

- an assertion states expected properties at a control point
- does not (directly) relate properties at different points

we designed and built a framework for data structure repair

- designed core algorithms
- developed optimizations

we performed experimental evaluation

key findings

- assertion-based repair is feasible for structures of moderate size, e.g., 1000 nodes, with a few faults
- assertion-based repair is effective at fixing structural faults

our ongoing work on data structure repair [funded in part by AFOSR]

development of the theoretical underpinnings of a framework for specification-based error recovery

- definition of the repair problem
  - let φ be a method post-condition that relates pre- and post-states. given a pre-state *u*, and an erroneous post-state *s*, i.e., !φ(*u*, *s*), mutate *s* into *t* so that φ(*u*, *t*)
- design of a veneer on a spec language for repair
  - support directed repair, e.g., make a field unmodifiable

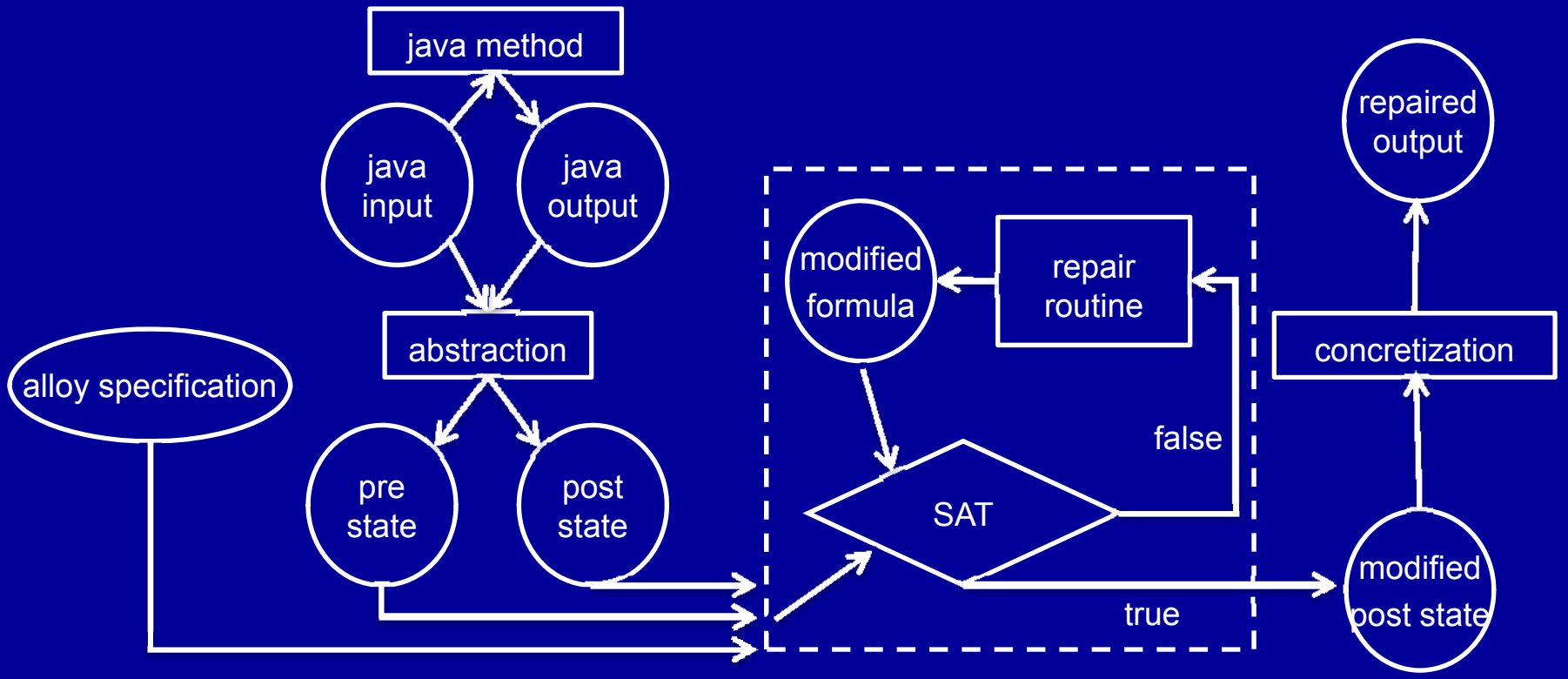
design and analysis of effective and efficient repair algorithms

support for enhanced usability

- infrastructure to assist the user with writing specs
- repair feedback mechanism that assists with debugging

experimental evaluation

tarmeem architecture



outline

overview

example

constraint-driven testing

constraint-driven error recovery

conclusion

some of our recent/ongoing projects

testing: how to generate test cases?

- use of constraints
- staged generation
- parallel generation
- white-box generation
- abstract symbolic generation

error recovery: how to repair erroneous program states?

design: how to analyze designs?

- sequential encoding for relational analysis
- declarative slicing for constraint prioritization

development: how to statically check code as it is developed?

summary

towards a unified framework for compile-time checking and runtime error recovery

- systematic testing before deployment
- systematic repair once deployed

efficient constraint solving lies at the heart of our work

- abstract symbolic execution

a novel view of constraints

- assert design constraints
- use assertions as a basis for test generation and checking of conformance of code to designs
- use violated assertions as basis for repair to ensure design constraints are not violated at runtime

khurshid@ece.utexas.edu