Enhancing Symbolic Execution Using Test Ranges

Sarfraz Khurshid
University of Texas at Austin
khurshid@ece.utexas.edu

1st International KLEE Workshop on Symbolic Execution
London, UK
20 April 2018

Work funded in part by the US National Science Foundation
In short, what is this talk about?
A tale of two techniques

Ranging for two systematic analysis techniques

• A symbolic execution technique
• A constraint solving technique

The two techniques look quite different but have commonalities

• Ranging to enhance them shares a common spirit – it applies even to other techniques
• Moreover, the two techniques have an intricate relation
  • Symbolic execution requires constraint solving
  • But it also enables constraint solving – for a class of constraints using a solver for another class!
    • E.g., symbolic execution can solve structural constraints using a solver for linear arithmetic

• Understanding this relation can help scale better

Khurshid: Enhancing Systematic Analyses Using Test Ranges
So what exactly is this talk about?

Basics of systematic constraint-driven testing

- Logical constraints describe inputs, outputs, paths, etc.
  - Programs with structurally complex inputs

Basics of test ranges and ranged analysis

- Enhance systematic techniques
  - Resumeable – pause and resume analysis; resume analysis after it fails (hits resource bound)
  - Parallel – distribute the analysis among different workers with minimal overhead
  - Incremental – re-use (some) analysis results after a change
- Apply to a range of techniques
Foundations
Systematic constraint-driven testing

Black-box view

• TestEra – based on Alloy/SAT [ASE’01]
  • ASE Most Influential Paper Award 2015
• Korat – imperative constraints [ISSTA’02]
  • ACM SIGSOFT Impact Paper Award 2012

White/gray-box view

• Symbolic execution for object-oriented code
  • Generalized symbolic execution [TACAS’03]
• Input generation using JPF [ISSTA’04]
  • ISSTA Retrospective Impact Paper Award 2018*

* Announced. To be awarded at ISSTA in July 2018

Khurshid: Enhancing Systematic Analyses Using Test Ranges
Structurally complex data

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

(Khurshid: Enhancing Systematic Analyses Using Test Ranges)
Outline

Overview
Basics of systematic constraint-driven testing
Basics of ranged analysis
A bit of history
Conclusions
Example: Binary search tree
How to systematically test remove?

class SearchTree {
    Node root;
    int size;

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

    // method under test
    void remove(int x) { ... }
}

input constraint: isTree() && isOrdered()
oracle constraint: isTree() && isOrdered() && “removes only x”
Systematic constraint-based test generation
Black-box view

Input constraints define properties of desired inputs
• Can characterize test purpose etc.
• Constraint solving problem only about properties of inputs, not program behaviors

Efficient solvers provide automatic test generation
• Alloy/SAT for declarative constraints [alloy.mit.edu]
• Korat for imperative constraints [korat.sourceforge.net]

Inputs are non-equivalent, i.e., tests have no redundancy
Test suites are dense, i.e., cover entire bounded input space

Oracle constraints automate test oracles

Khurshid: Enhancing Systematic Analyses Using Test Ranges
Example: Declarative constraints
Based on Alloy/SAT

Input constraint

```
# 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
```

```
...  // binary search
```

Oracle constraint

```
root.*(left + right).info = root`.*(left` + right`).info` - x  // remove method
```

Khurshid: Enhancing Systematic Analyses Using Test Ranges
Example: Imperative constraints

```java
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
    // check binary search properties
    return true;
}
```
How to solve an imperative constraint?  
A simple approach: Use $repOk$ as a filter

The constraint is executable. So, execute it – over and over again – to solve it!

- Create many candidate inputs, run $repOk$ to filter

E.g., consider trees with $\leq 3$ nodes

- $4 \times 4 \times (3 \times 4 \times 4)^3 > 1.7$M candidates; but only 15 are valid and non-isomorphic!
Using *repOk* as a filter: Example

Search tree with ≤ 3 nodes, 3 int values

\[(t.\text{root}, t.\text{size}, n_0.\text{left}, n_0.\text{right}, n_0.\text{info}, n_1.\text{left}, n_1.\text{right}, n_1.\text{info}, n_2.\text{left}, n_2.\text{right}, n_2.\text{info}]\]

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 2
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 3
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 2
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 3
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 2
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 3
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 1

Valid: 249,984
Invalid: 1,519,488

Khurshid: Enhancing Systematic Analyses Using Test Ranges
Korat solver for imperative constraints
[ISSTA’02: Boyapati, Khurshid, Marinov]

Key insight: repOk executions can help prune input space
- Monitor accesses of object fields

Algorithm
- Explores bounded input space defined by a finitization
- Represents structures using candidate vectors, e.g.,

\[
\begin{array}{ccc}
\text{BinarySearchTree} & N_0 & N_1 \\text{root} & \text{size} & \text{info} \\text{left} & \text{right} & \text{info} \\text{left} & \text{right} \\text{info} \\text{left} & \text{right}
\end{array}
\]

- For size ≤ 3, #candidates > 1.7M
- Executes repOk on a candidate to check its validity and to determine which candidate to check next
- Provides isomorph-free generation

Khurshid: Enhancing Systematic Analyses Using Test Ranges
Example: Monitoring field accesses

```java
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
    // check binary search properties
    return true;
}
```

Example: Korat search step

Backtrack using field access list
[ T₀.root, N₀.left, N₀.right ]

Generate the next candidate
- which satisfies repOk

Prune from the search all $3^3 \cdot 4^4 = 6,912$ candidates of the form
Korat search example: Dyn. backtracking
Search tree with ≤ 3 nodes, 3 int values

\[ t.\text{root}, t.\text{size}, n_0.\text{left}, n_0.\text{right}, n_0.\text{info}, n_1.\text{left}, n_1.\text{right}, n_1.\text{info}, n_2.\text{left}, n_2.\text{right}, n_2.\text{info} \]

000000000000 *** 100200000000 130200000000
010000000000 110200000000 100200100000
020000000000 120200000000 100200200000
030000000000 120200010000 *** 100200300000
100000000000 120200020000 *** 110200300000
110000000000 *** 120210000000 120200300000
110010000000 *** 120210010000 130200300000
110020000000 *** 120210020000 *** 130200310000
120000000000 120220000000 130200310010
130000000000 120220010000 13020031002 ***
100100000000 120220020000 ...

Khurshid: Enhancing Systematic Analyses Using Test Ranges
Korat search example: Many invalid cands. Search tree with ≤ 3 nodes, 3 int values

\[ t.\text{root}, t.\text{size}, n_0.\text{left}, n_0.\text{right}, n_0.\text{info}, n_1.\text{left}, n_1.\text{right}, n_1.\text{info}, n_2.\text{left}, n_2.\text{right}, n_2.\text{info} \]

<table>
<thead>
<tr>
<th>0 0 0 0 0 0 0 0 0 0 0 0 0 ***</th>
<th>1 0 0 2 0 0 0 0 0 0 0 0 0 1 3 0 2 0 0 0 0 0 0 0 0 0 0</th>
<th>1 3 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 2 0 0 0 0 0 0 0 0 1 0 0 2 0 0 1 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 2 0 0 0 0 0 0 0 0 0 0 0 0 1 2 0 2 0 0 0 0 0 0 0 0 1 0 0 2 0 0 2 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 3 0 0 0 0 0 0 0 0 0 0 0 0 1 2 0 2 0 0 0 1 0 0 0 0 0 0 0 1 0 0 2 0 0 3 0 0 0 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 2 0 2 0 0 0 2 0 0 0 0 0 0 0 1 1 0 2 0 0 3 0 0 0 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 2 0 2 1 0 0 0 0 0 0 0 1 2 0 2 0 0 3 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 0 0 1 0 0 0 0 0 0 0 0 0 1 2 0 2 1 0 0 1 0 0 0 0 1 3 0 2 0 0 3 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 0 0 2 0 0 0 0 0 0 0 0 0 1 2 0 2 1 0 0 2 0 0 0 0 1 3 0 2 0 0 3 1 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 2 0 0 0 0 0 0 0 0 0 0 0 0 1 2 0 2 2 0 0 0 0 0 0 0 1 3 0 2 0 0 3 1 0 0 1 0 0 0 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 3 0 0 0 0 0 0 0 0 0 0 0 0 1 2 0 2 2 0 0 1 0 0 0 0 1 3 0 2 0 0 3 1 0 0 2 0 0 0 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 0 1 0 0 0 0 0 0 0 0 0 0 1 2 0 2 2 0 0 2 0 0 0 0 1 3 0 2 0 0 3 1 0 0 2 0 0 0 0 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **#explored** = 178; **#valid found** = 15; **#candidates** > 1.7M

Khurshid: Enhancing Systematic Analyses Using Test Ranges
Systematic constraint-based test generation
White/gray-box view: Generalized Symbolic Execution

[TACAS’03: Khurshid, Pasareanu, Visser]
Symbolic execution for primitives a la 70’s style
Concrete execution for references using lazy initialization on access, e.g., consider “t.next”

- Enabled handling complex libraries, e.g., Java Collections
- Included in UC-KLEE [Ramos+CAV’11]

Abstract symbolic execution for library class java.util.String
- Build and solve constraints on strings
Outline

Overview
Basics of systematic constraint-driven testing
Basics of ranged analysis
A bit of history
Conclusions
Ranged analysis: Intuition

“What’s in a test?!”

• A test input encodes the state of an analysis run
  • Partitions the state space: explored, unexplored
  • Enables resumeable analysis (pause, continue later)
    • May resume on a different machine (faster or with more memory)
• Allows quick recovery if analysis crashes
• Examples
  • A candidate vector encodes the state of Korat search
  • A test input encodes the state of symbolic execution

“What’s in 2 tests?!”
Ranged analysis: Basic concept

A test pair \([t_1, t_2]\) defines an analysis range

- The analysis only explores the subset of state space defined by the range

Ranging applies to several analyses

- Parallel Korat [FSE’07]
  - Parallel workers explore non-overlapping ranges
- Ranged symbolic execution [OOPSLA’12]
  - Work stealing for load balancing
- Ranged model checking [JPF’12]
  - Stateful model checker
- Ranged Alloy [ASE’13]
  - Black-box back-end search based on SAT
Ranged analysis: Forming ranges

Korat – 2 candidate vectors \(<v, w>\) where \(v\) is lexicographically smaller than \(w\), i.e., Korat search explores \(v\) before \(w\)

Symbolic execution – 2 test inputs \(<x, y>\) where \(\text{path}(x)\) is lexicographically smaller than \(\text{path}(y)\)
  - Symbolic execution explores \(\text{path}(x)\) before \(\text{path}(y)\)
Illustration: Triangle classification

```java
// Jeff Offutt -- Java version Feb 2003
...
// The main triangle classification method
static int triang(int Side1, int Side2, int Side3) {
    int tri_out;
    // tri_out is output from the routine:
    // Triang = 1 if triangle is scalene
    // Triang = 2 if triangle is isosceles
    // Triang = 3 if triangle is equilateral
    // Triang = 4 if not a triangle
    // After a quick confirmation that it's a legal
    // triangle, detect any sides of equal length
    if (Side1 <= 0 || Side2 <= 0 || Side3 <= 0) {
        tri_out = 4;
        return (tri_out);
    }
    tri_out = 0;
    if (Side1 == Side2) tri_out = tri_out + 1;
    if (Side1 == Side3) tri_out = tri_out + 2;
    if (Side2 == Side3) tri_out = tri_out + 3; ...
```
Illustration: Symbolic execution results

PC<sub>1</sub>: (S1 > 0), (S2 > 0), (S3 > 0), (S1 != S2), (S1 != S3), (S2 != S3),
((S1 + S2) > S3), ((S2 + S3) > S1), ((S1 + S3) > S2)

• Solution: S1 = 3, S2 = 4, S3 = 2; Output: 1

PC<sub>2</sub>: (S1 > 0), (S2 > 0), (S3 > 0), (S1 != S2), (S1 != S3), (S2 != S3),
((S1 + S2) > S3), ((S2 + S3) > S1), ((S1 + S3) <= S2)

• Solution: S1 = 2, S2 = 3, S3 = 1; Output: 4

PC<sub>3</sub>: (S1 > 0), (S2 > 0), (S3 > 0), (S1 != S2), (S1 != S3), (S2 != S3),
((S1 + S2) > S3), ((S2 + S3) <= S1)

• Solution: S1 = 3, S2 = 2, S3 = 1; Output: 4

PC<sub>4</sub>: (S1 > 0), (S2 > 0), (S3 > 0), (S1 != S2), (S1 != S3), (S2 != S3),
((S1 + S2) <= S3)

• Solution: S1 = 1, S2 = 2, S3 = 3; Output: 4

PC<sub>5</sub>: (S1 > 0), (S2 > 0), (S3 > 0), (S1 != S2), (S1 != S3), (S2 == S3),
((S2 + S3) <= S1)

• Solution: S1 = 2, S2 = 1, S3 = 1; Output: 4
Illustration: Symbolic execution tree

Khurshid: Enhancing Systematic Analyses Using Test Ranges
Illustration: Ranging

Consider 2 tests: $t_6 = <1, 2, 2>$ and $t_9 = <1, 1, 2>$
- Range $[t_6, t_9]$ includes 5 paths
  - 4 are feasible
  - 1 is infeasible
- 3 ranges $[-, t_6)$, $[t_6, t_9)$, $[t_9, -]$ partition the exploration

Khurshid: Enhancing Systematic Analyses Using Test Ranges
Ranged analysis: Characteristics
“What’s in a range?!"

Ranges have succinct representations
Ranging provides a natural way to distribute the search
  • However, forming “equi-distant” ranges requires care

Ranges encode a variety of useful analysis results
  • Enable memoization and incremental analysis
Ranges define (and are defined by) test input orderings
  • Provide a basis for test prioritization, minimization, …
    • E.g., “pick a test that is further away from this test”

[FSE’07, OOPSLA’12, Siddiqui-UT-PhD’12, Qiu-UT-PhD’16,
  Dini-UT-MS’16, ICSE_poster’17, SPIN’17, NFM’18]
Specializing ranges: Re-execution

Infeasible ranges – summarize infeasibility results

E.g., for Korat, all candidates in the range are invalid, but still must be checked explicitly by the search one by one.

Future search – for the same problem – can skip them

- E.g. previously tested “if (repOk()) m();” and now test “if (repOk()) p();”

![Diagram showing ranges]

2 largest invalid ranges: \([cv_0, cv_{366}]\) and \([cv_{739}, cv_{829}]\)
- Represent 47% of the candidates explored
Specializing ranges: Constraint caching

Feasible ranges – summarize feasibility results
E.g., for symbolic execution, all paths in the range are feasible
  • $[t_1, t_2, d]$ – all paths in range $[t_1, t_2]$ up to depth $d$

Distributed workers can share constraint feasibility results using lightweight communication based on feasible ranges
  • Re-create results by solver-free symbolic exploration

A sequence of feasible ranges can encode the entire program’s constraint feasibility database – including infeasibility results
[\[t_1 = <3, 4, 2>, t_8 = <2, 1, 2>, \infty\]\]

is a feasible range

- It includes 8 paths
[\[t_1 = <3, 4, 2>, t_8 = <2, 1, 2>, -\]} encodes that each of the following path conditions is feasible:

PC_1: (S1 > 0), (S2 > 0), (S3 > 0), (S1 != S2), (S1 != S3), (S2 != S3),
((S1 + S2) > S3), ((S2 + S3) > S1), ((S1 + S3) > S2)

PC_2: (S1 > 0), (S2 > 0), (S3 > 0), (S1 != S2), (S1 != S3), (S2 != S3),
((S1 + S2) > S3), ((S2 + S3) > S1), ((S1 + S3) <= S2)

PC_3: (S1 > 0), (S2 > 0), (S3 > 0), (S1 != S2), (S1 != S3), (S2 != S3),
((S1 + S2) > S3), ((S2 + S3) <= S1)

PC_4: (S1 > 0), (S2 > 0), (S3 > 0), (S1 != S2), (S1 != S3), (S2 != S3),
((S1 + S2) <= S3)

PC_5: (S1 > 0), (S2 > 0), (S3 > 0), (S1 != S2), (S1 != S3), (S2 == S3),
((S2 + S3) <= S1)

PC_6: (S1 > 0), (S2 > 0), (S3 > 0), (S1 != S2), (S1 != S3), (S2 == S3),
((S2 + S3) > S1)

PC_7: (S1 > 0), (S2 > 0), (S3 > 0), (S1 != S2), (S1 == S3), (S2 != S3),
((S1 + S3) <= S2)

PC_8: (S1 > 0), (S2 > 0), (S3 > 0), (S1 != S2), (S1 == S3), (S2 != S3),
((S1 + S3) > S2)
Feasible ranges: Illustration

3 feasible ranges encode all constraint feasibility results:

\[ t_1 = <3, 4, 2>, t_8 = <2, 1, 2>, - \]
\[ t_9 = <1, 1, 2>, t_{10} = <2, 2, 1>, - \]
\[ t_{11} = <1, 1, 3>, -, - \]
Specializing ranges: Continuation

Unexplored ranges – contain some unexplored candidate(s)
  • Can be explored later, by another worker, or even another technique
E.g., for symbolic execution, different test generation techniques can apply in tandem
Tests created by another technique or manually provide the basis to define unexplored ranges
Unexplored ranges: Illustration

Assume user provides 4 tests: 
\{t_5, t_6, t_7, t_{10}\} (in any order)
This test suite leads to 3 unexplored ranges:
\([-, t_5), (t_7, t_{10}), (t_{10}, -]\)
Specializing ranges: Extension

Mixed ranges – summarize one search step
E.g., for Korat: \([v, w)\) is a mixed range, if \(v\) is valid and \(w = Korat.nextCV(v)\)
Korat search can be made incremental when \(repOk\) is extended, e.g., binary tree evolves to a binary search tree
Outline

Overview
Basics of systematic constraint-driven testing
Basics of ranged analysis
A bit of history
Conclusions
Constraints in testing

Boyer et al. [1975], Clarke [1976], Howden [1975], King [1976] pioneered core ideas – in the context of symbolic execution
- Constraints based on execution paths – path conditions
- Constraints provided by the user – assertions
  - Focus: numeric constraints

Tools have existed for over 4 decades
- SELECT – A Formal System for Testing and Debugging Programs by Symbolic Execution [Boyer+’75]
- EFFIGY
  - Symbolic Execution and Program Testing [King’76]
Constraints in SELECT [Boyer+’75]

G. User Supplied Assertions as an Adjunct to the Program Code

As another mode of operation it is possible to insert assertions, possibly in the form of programs themselves, at various points in the program including the output. These assertions can serve as

(2) constraint conditions that enable a user to define subregions of the input space from which SELECT is to generate the test data, or

(3) specifications for the intent of the program from which it is possible to verify the paths of the program. Note that this does not imply that the program itself is correct, which would require that all program paths are verified.
Path-based verification and need to support debugging [King-PhD-CMU’69]

When a verification condition is found not to be a theorem, one usually is able to exhibit a set of values for the variables which make it evaluate to ‘false’. The linear-solver in our prover should be modified to produce a counter-example set of values whenever the proof fails. These values can be used to form a particular state vector for some point in the program where the program and assertions disagree. A verifier which was able to construct such counter-examples for erroneous programs would be an extremely useful debugging aid. Other useful aids would also evolve from careful consideration of the whole process with debugging in mind.
8. Program Correctness, Proofs, and Symbolic Execution

That is, one must show, using any set of variable values which satisfy the predicate at the beginning of the path, that the values resulting from execution along the path must satisfy the predicate at the end.

One can prove the correctness of each path by executing it symbolically as follows:

1. Change the ASSERT at the beginning of the path to an ASSUME; change the ASSERT at the end of the path to a PROVE.
2. Initialize the path condition to true and all the program variables to distinct symbols say, $\alpha_1, \alpha_2, \ldots$.
3. Execute the path symbolically. Whenever an unre
symbolic testing. If one is strictly confined to symbolic execution without the use of any user introduced predicates, \( pc \) and the expressions requiring proof are syntactically and semantically determined by the programming language. However, the predicate semantics in correctness proofs derive from the \( problem \) area of the program and not the programming language.

It is this difference that convinces us that symbolic execution for \textit{testing programs} is a more exploitable technique in the short term than the more general one of program verification.
Outline

Overview
Basics of systematic constraint-driven testing
Basics of ranged analysis
A bit of history
Conclusions
Conclusions

Logical constraints have a key role in effective testing
  - Can capture a rich class of (input/oracle) properties
Systematic testing is effective at finding bugs
  - Handles programs with complex inputs
Ranging offers exciting ways to enhance systematic analyses
  - A test encodes analysis state and allows resumeability
  - A test pair forms a range that defines a search sub-space
    - Simple ranges enable parallel analysis
    - Infeasible, feasible, unexplored, and mixed ranges enables memoization and incremental analysis
Acknowledgements

Darko Marinov  Nima Dini
Chandrasekhar Boyapati  Diego Funes
Corina S. Pasareanu  Aleksandar Milicevic
Willem Visser  Sasa Misailovic

Nemanja Petrovic
Junaid Haroon Siddiqui
Rui Qiu
Guowei Yang
Cagdas Yelen
Junye Wen

Work funded in part by the US National Science Foundation

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

Khurshid: Systematic Software Testing Using Logical Constraints