Incremental Symbolic Execution for Automated Test Suite Maintenance

Sarmad Makhdoom Muhammad Adeel Khan Junaid Haroon Siddiqui

LUMS School of Science and Engineering
Lahore, Pakistan

ASE 2014
September 2014

Introduction

- System failures are expensive
- Systematic testing is effective at finding bugs
- Symbolic execution is a popular technique for systematic testing based on path exploration
- Supplement it with automatic testing
- Focus on the effects of program changes

Overview

- We present a novel approach to apply symbolic execution on increments
- Out key insight is that we can eliminate constraint solving for unchanged code
  - Used previously generated test suite
  - Checking constraints are faster than solving
- Use **dynamic analysis** to find affected code by change

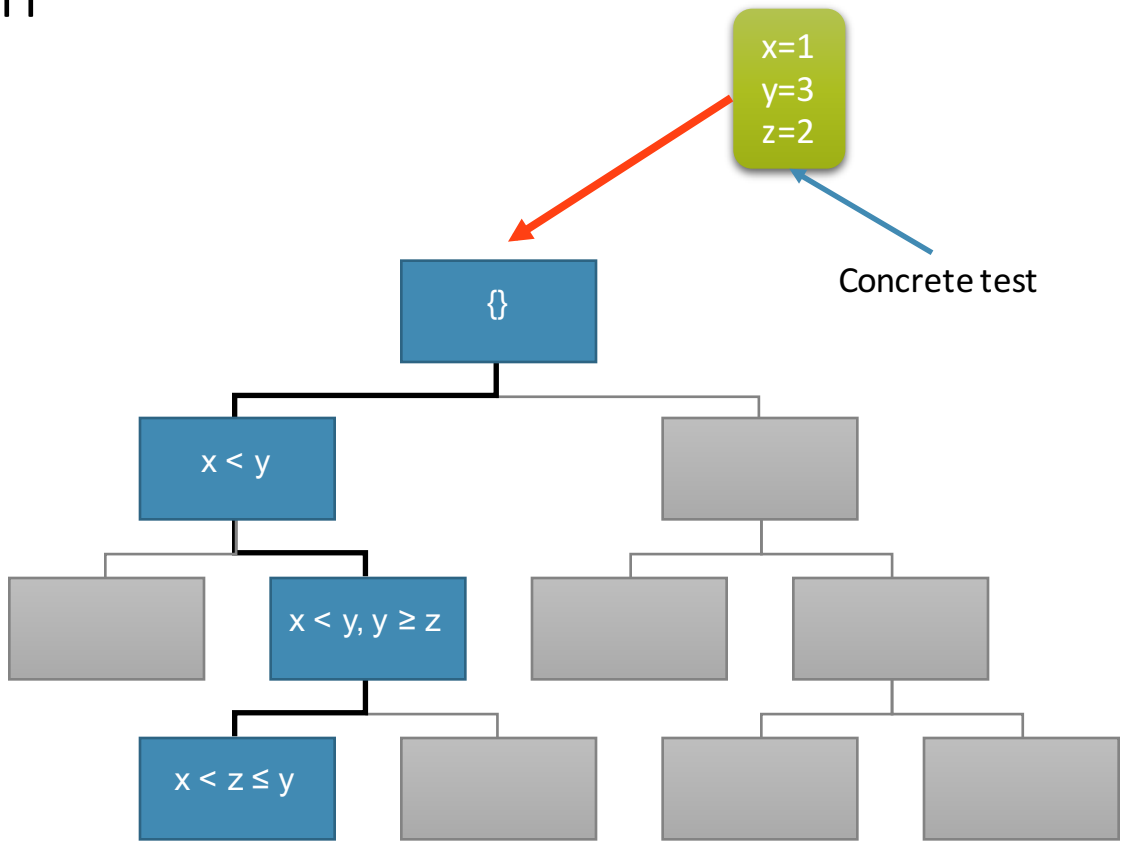
Outline

- Background
- Incremental Symbolic Execution
- Experiments
- Related Work
- Conclusion

Background

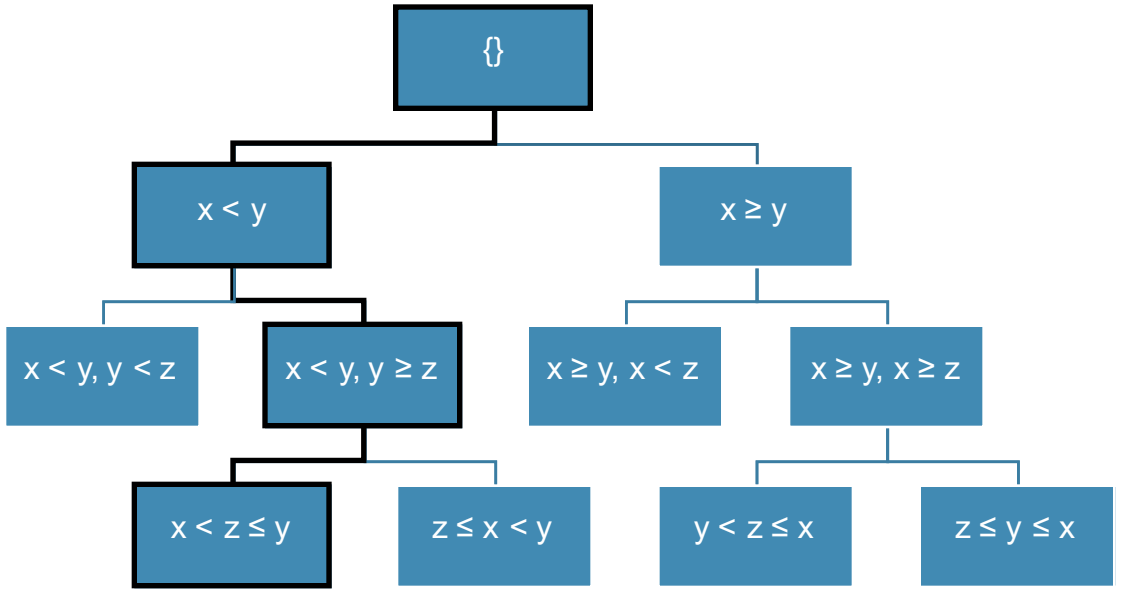
Concrete Execution

```
int mid(int x, int y, int z) {
  if (x<y){
    if (y<z){
      return y;
    }else{
      if (x<z)
        return z;
      else
        return x;
    }
  }else{
    if (x<z){
      return x;
    }else{
      if (y<z)
        return z;
      else
        return y;
    }
  }
}
```



Symbolic Execution Demo

```
int mid(int x, int y, int z) {
  if (x<y){
    if (y<z){
      return y;
    }else{
      if (x<z)
        return z;
      else
        return x;
    }
  }else{
    if (x<z){
      return x;
    }else{
      if (y<z)
        return z;
      else
        return y;
    }
  }
}
```



Path condition (feasible)

Solve

x=1
y=3
z=2

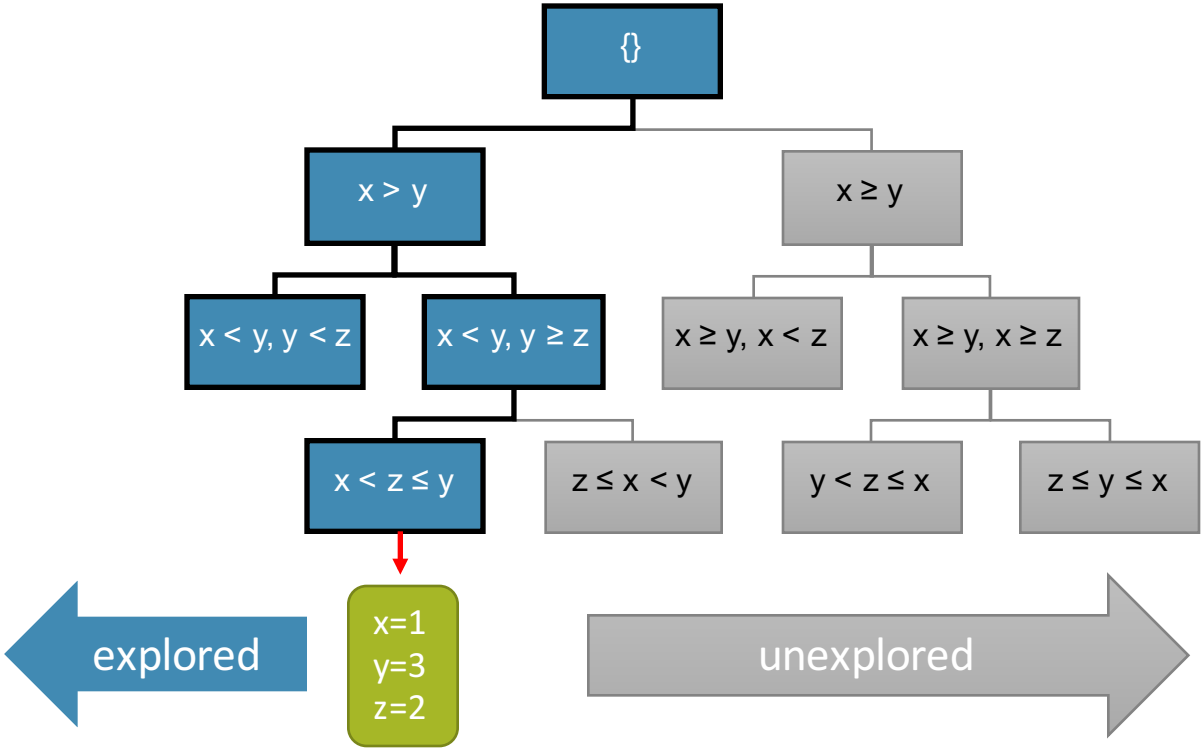
Concrete Test

Symbolic Execution [CACM'76]

- Concrete execution fixes input variables and exercises one path per input
- Symbolic execution uses symbols with no restrictions other than type
- Both branches of every condition in the program are explored
- A path condition is built for every path and contains the constraints required to take this path

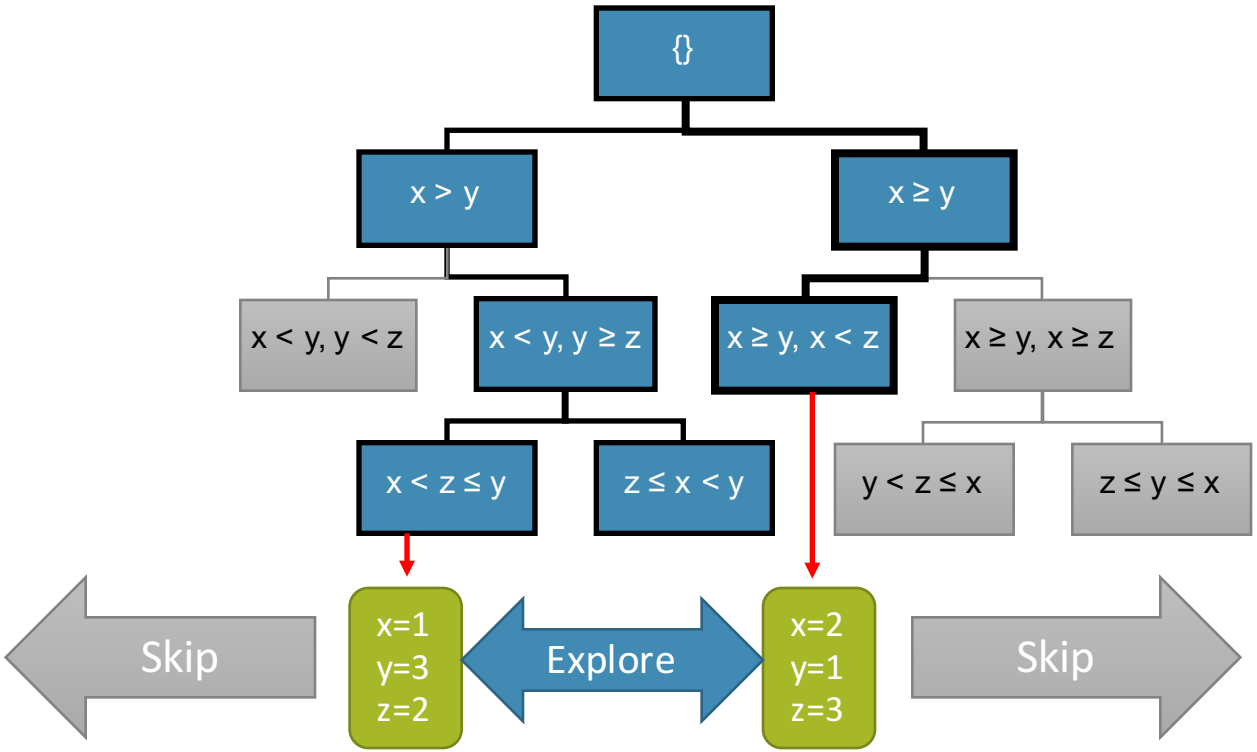
"Symbolic Execution and Program Testing" – CACM 1976

Ranged Symbolic Execution [OOPSLA'12]



Scaling symbolic execution using ranged analysis. ACM-OOPSLA 2012

Ranged Symbolic Execution [OOPSLA'12]



Scaling symbolic execution using ranged analysis. ACM-OOPSLA 2012

Motivation

Motivation

- All execution paths may be very large and may not be of interest
- Only execution paths that differ between two versions are of interest
- Test only program changes not the whole program
- Incremental symbolic execution is useful in bug finding and regression testing

Incremental Symbolic Execution

Key Ideas

- Majority of search space is invalid
- Solving path conditions is expensive
- Comparing and validating of path conditions is cheap
- One way is to compare both CFGs *[DiSE'11]*
  - Static analysis
  - Inexact (in-depth node changes are problematic)
  - Scalability issues

Technique

- Full symbolic execution with initial version:
  - Generates inputs for each distinct path
- Incremental symbolic execution on subsequent versions
  - On exploration divide tests based on each branch condition
  - Compare and validate tests
    - If test is valid, don't use solver
    - If test is invalid, explore the program for new states

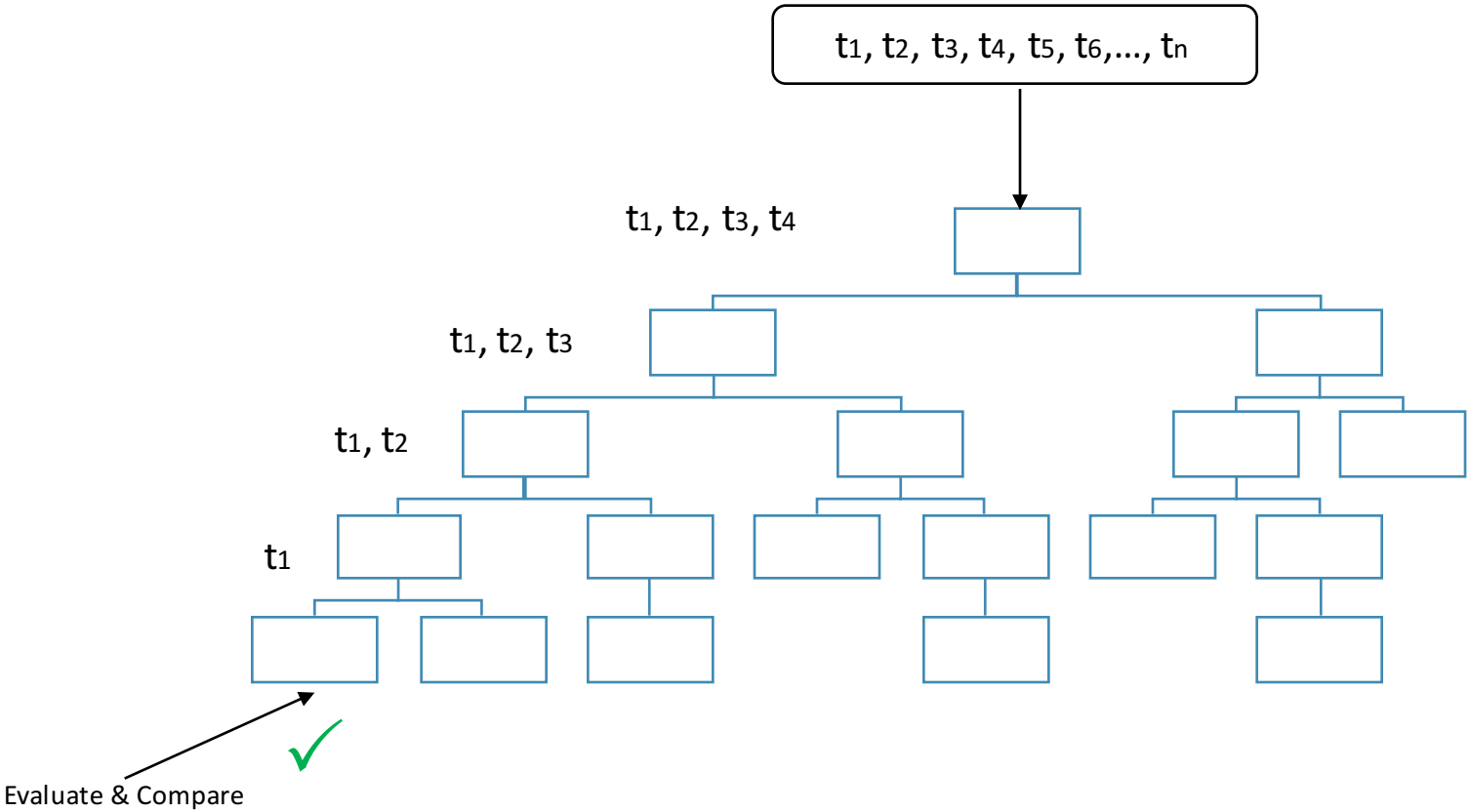
Algorithm

Input: A finite set $TestSuite = \{t_1, t_2, \ldots, t_n\}$ of test cases

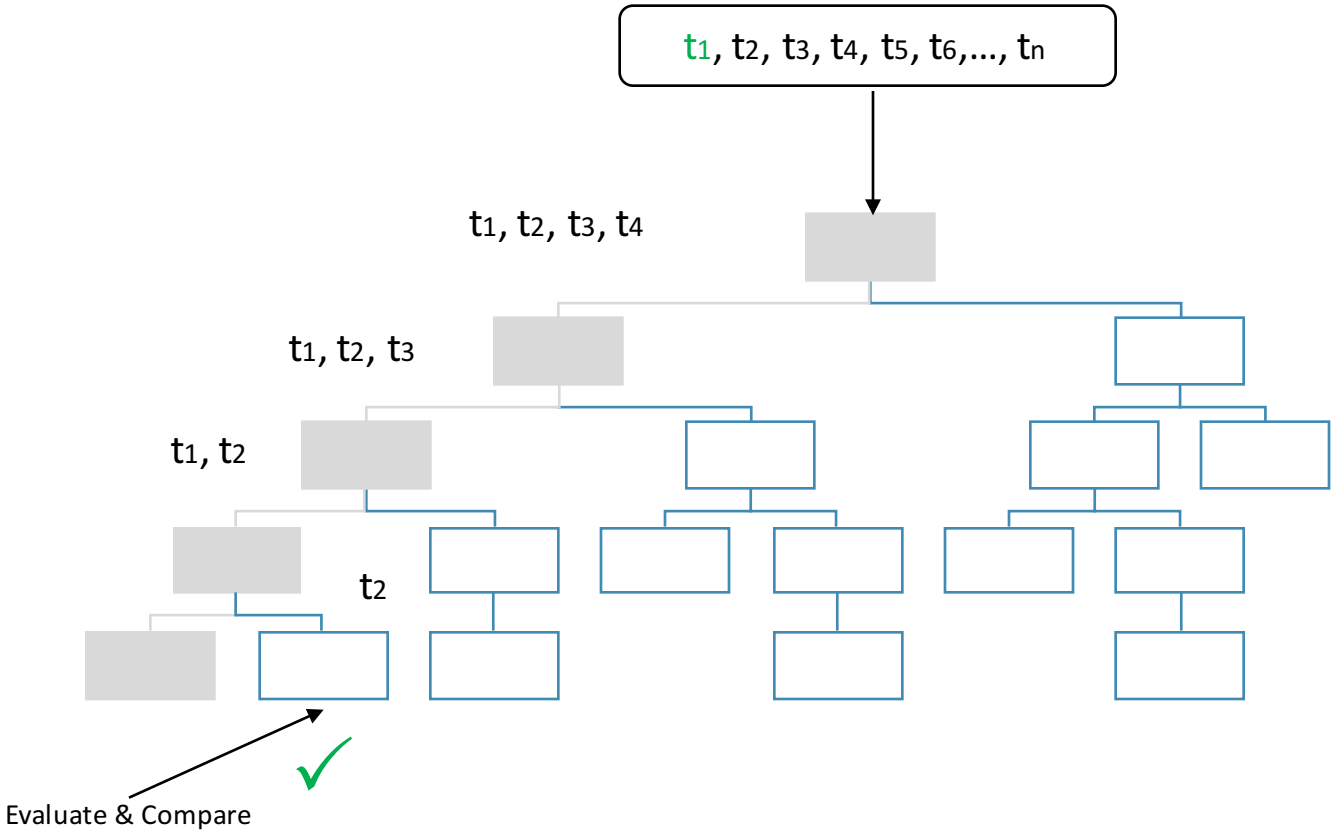
```
1  for each b in BranchCondition do
2      T_true ← split(TestSuite, b); T_false ← split(TestSuite, ¬b)
3      T_invalid ← TestSuite − (T_true ∪ T_false)
4      if T_invalid ≠ ∅ then
5          if T_true = ∅ then
6              exploreAndSolve(T_true)
7              IncrementalExplore(T_false)
8          if T_false = ∅ then
9              exploreAndSolve(T_false)
10             IncrementalExplore(T_true)
11     else
12         IncrementalExplore(T_true)
13         IncrementalExplore(T_false)
```

Algorithm to explore new ranges and not solving the present path conditions in the new program

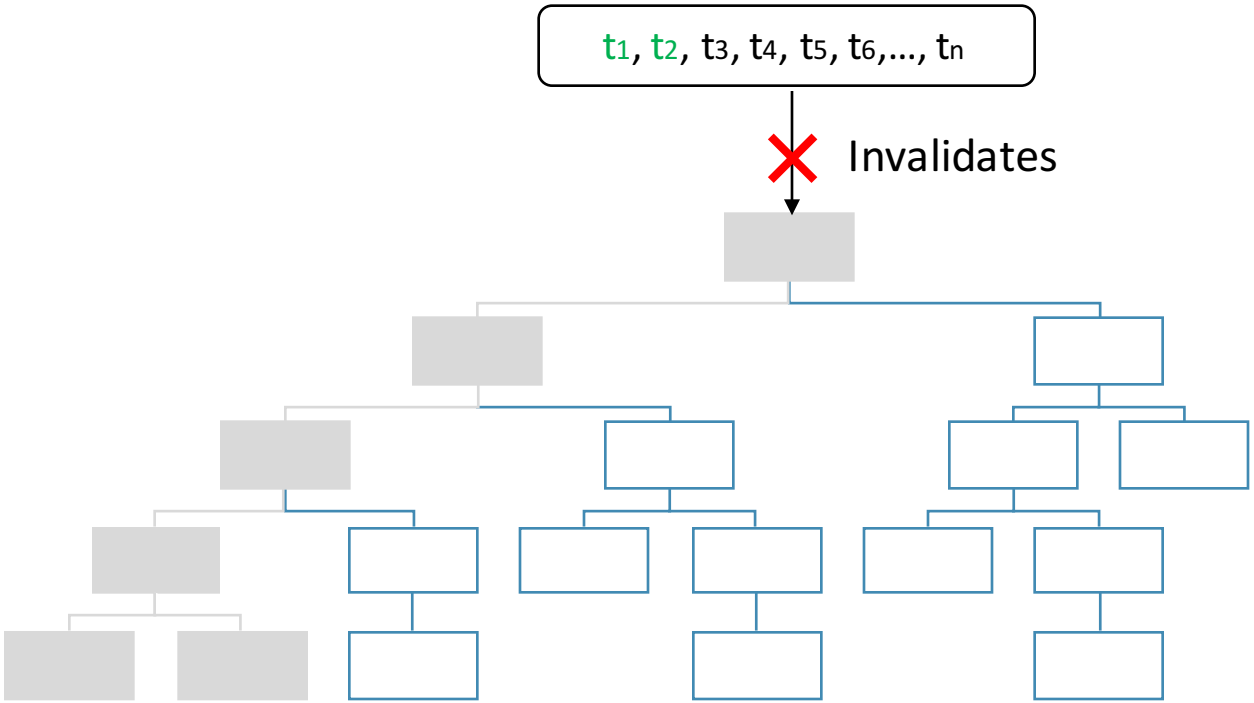
Technique



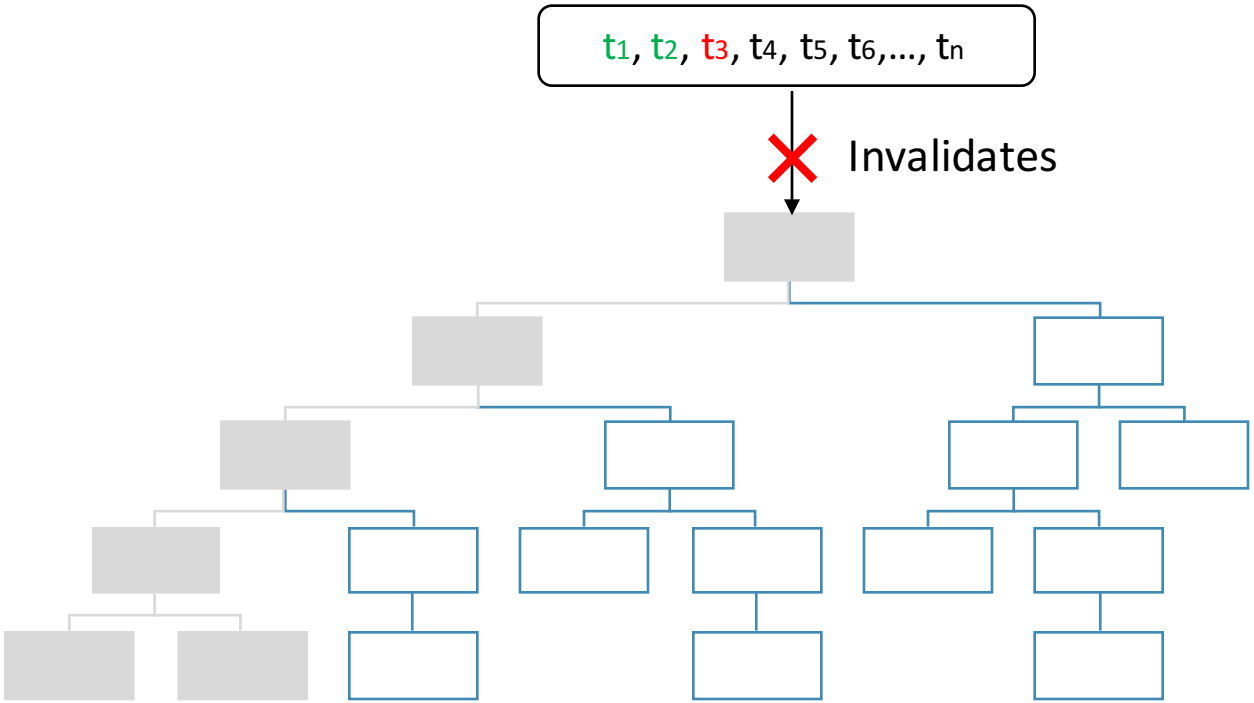
Technique



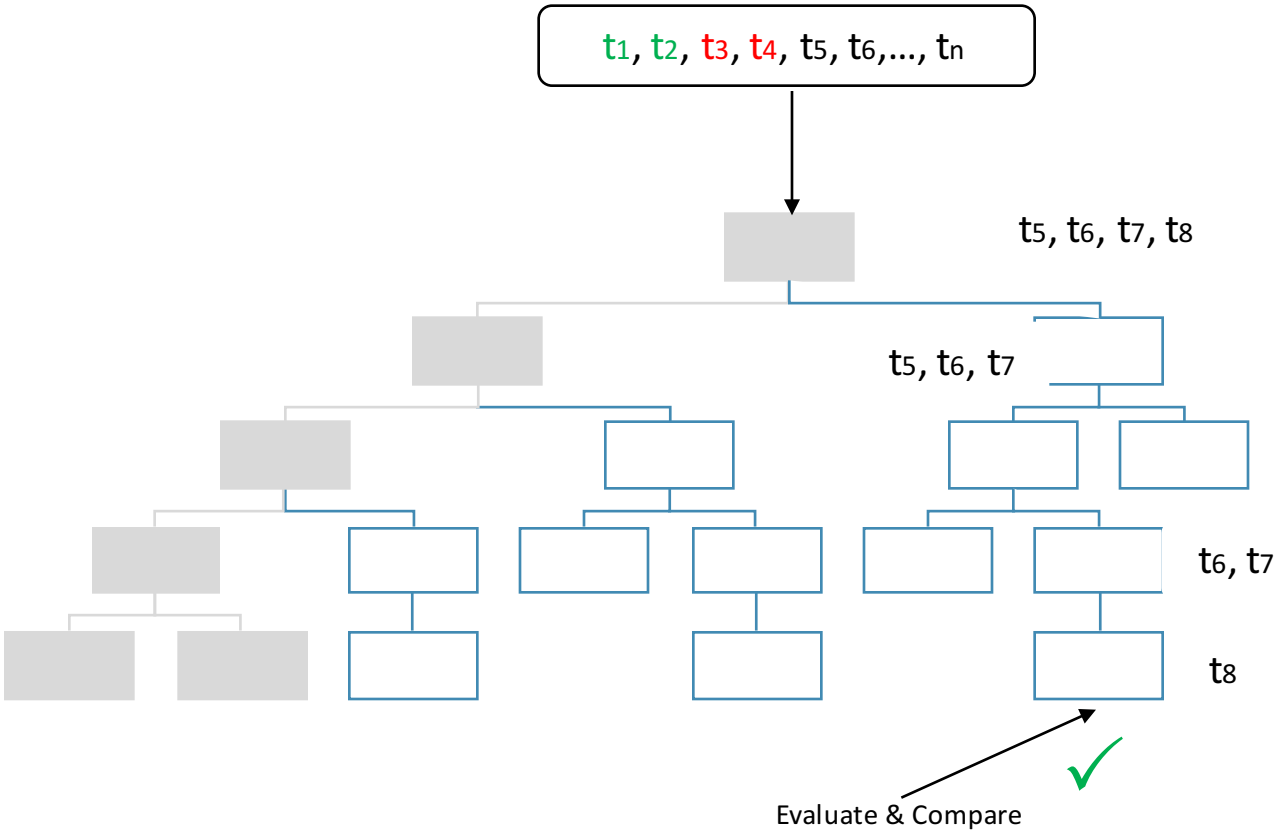
Technique



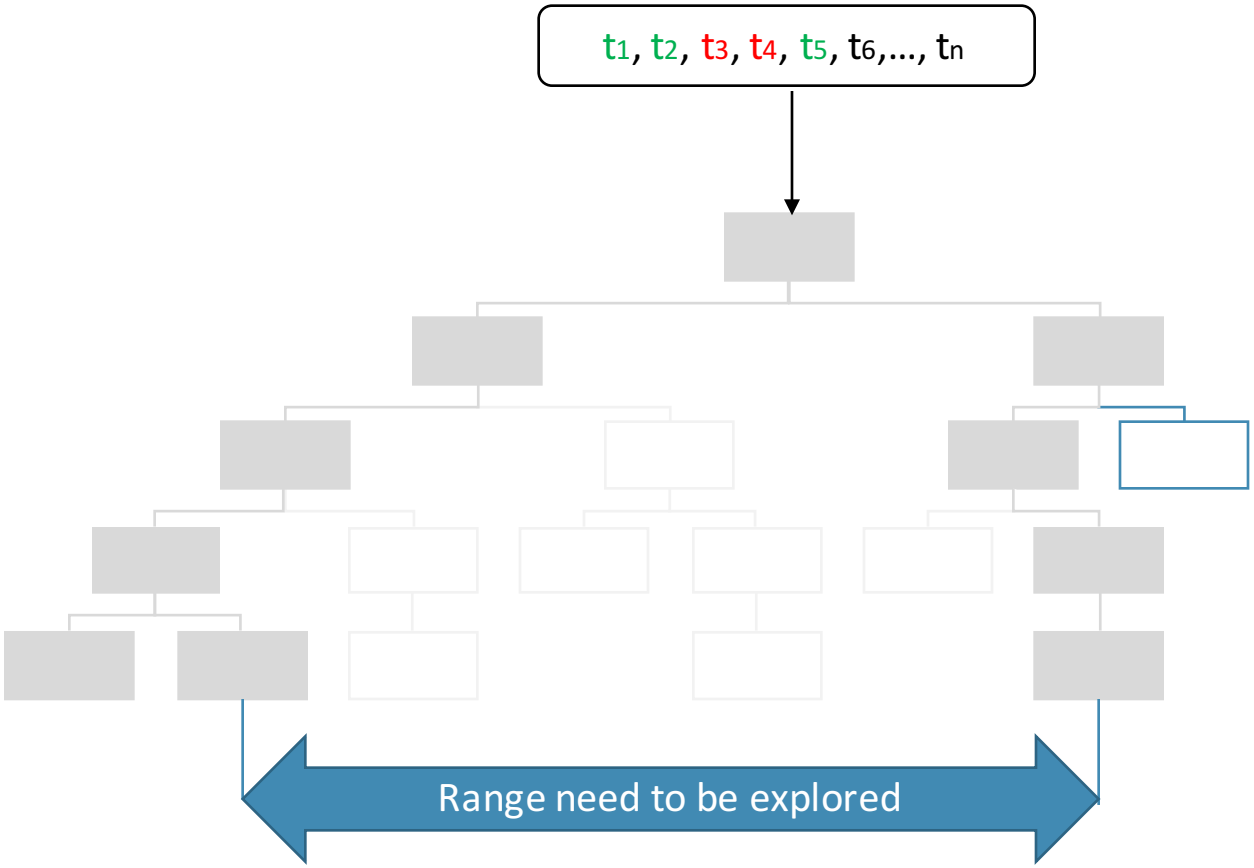
Technique



Technique



Technique



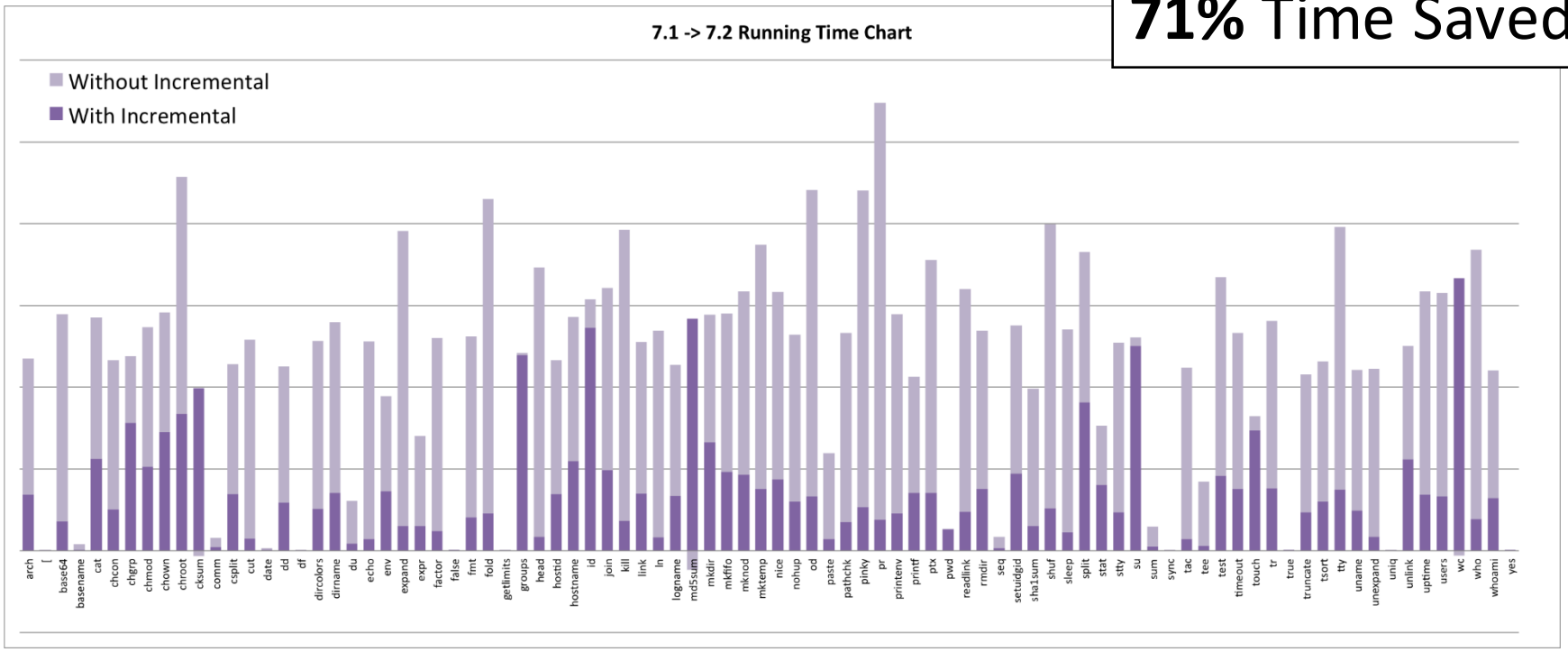
Evaluation

Evaluation

- Incremental testing on two different GNU Coreutils Suite
  - Minor update release (v7.1 → v7.2)
  - Major update release (v7.2 → v8.1)
- 85 stand-alone (i.e. excluding wrappers) apps
  - File system management: ls, mkdir, chmod, etc.
  - Management of system properties: hostname, printenv, etc.
  - Text file processing : sort, wc, od, etc.
  - …
- Tests performed on Lonestar Linux cluster at TACC (http://tacc.utexas.edu/)

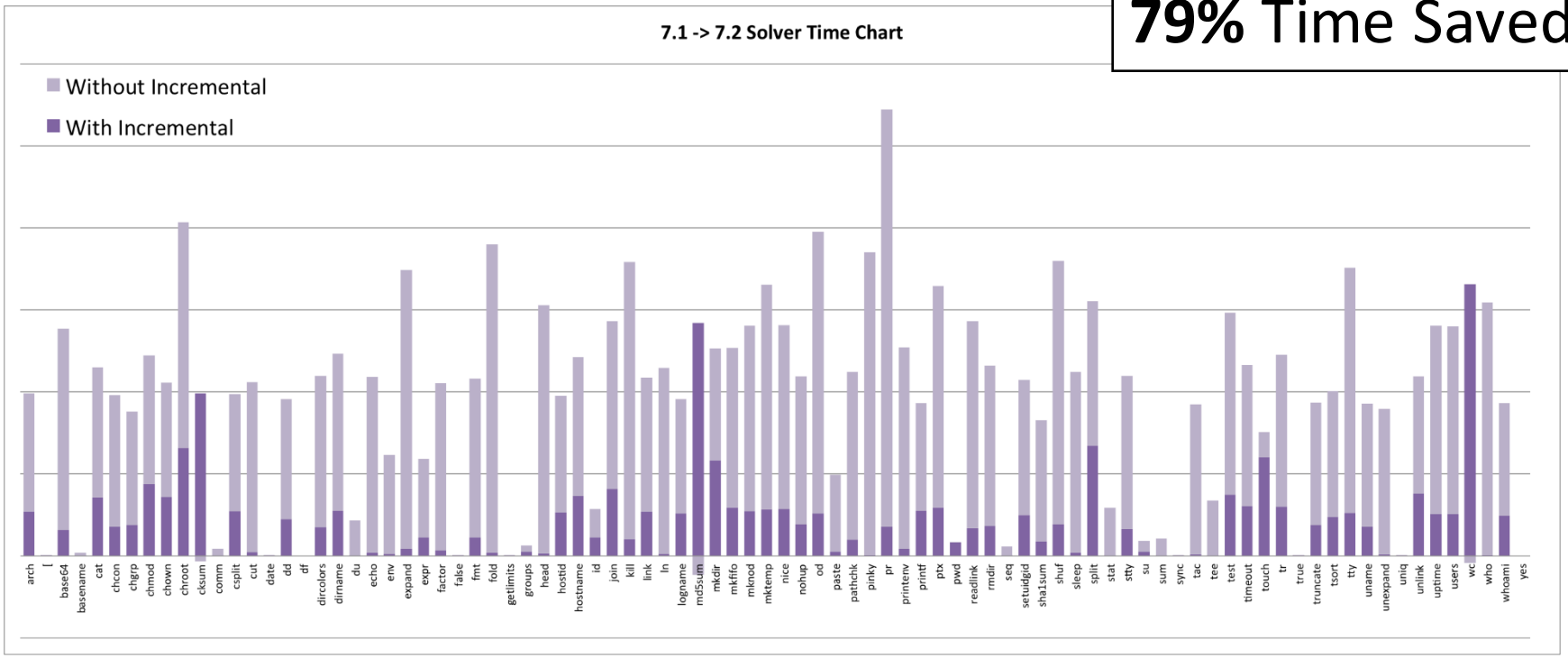
Results of Evaluation (v7.1→v7.2 – Running Time)

71% Time Saved



Results of Evaluation (v7.1→v7.2 – Solver Time)

79% Time Saved



Related Work

Directed Incremental Symbolic Execution [PLDI'11]

- Compute affected location by comparing CFGs
- Their technique is based on static analysis and reachability analysis
- Perform symbolic execution on only modified CFG

- Out technique perform dynamic analysis
- Generate test suite for whole new program not only for modified area

Directed incremental symbolic execution. PLDI 2011

KATCH [FSE'13]

- Combines static and dynamic analysis for increased coverage
- Katch select tests from manual generated test suite
- Perform heuristics based dynamic analysis to increase chance of hitting modified code
- Execute modified code symbolically for test suite generation

- We don't use static analysis or manual test suite for incremental testing
- Our technique can also explore in deeper depth in less time

KATCH: High-Coverage Testing of Software Patches. FSE 2013

Memoized Symbolic Execution [ISSTA'12]

- Cache based symbolic execution technique
- Store the results in trie-based data structure
- Re-use results from previous run by maintaining and updating trie
- Saving relies on position on change

- Our proposed technique based on ranged analysis is more effective
- Low cost of storing tests

Green: Reducing, Reusing and Recycling Constraints [FSE'12]

- An interface between analyzer and solver
- Benefits
  - Reuse within an analysis run
  - Reuse Across Programs, Analyses, Solvers
- Procedure
  - Path Conditioning Slicing
  - Canonization
  - Storage
  - The Green Solver Interface

Green: Reduce, reuse and recycle constraints in program analysis, in FSE. ACM, 2012

Conclusion

Conclusion

- Symbolic execution allows us to reason about multiple concrete executions
- Ranged symbolic execution allows dividing the problem of symbolic execution
- The proposed Incremental technique enables more *effective* and *efficient* testing of code using symbolic execution
  - Fully dynamic approach to find changes
  - Incremental testing in much less time
- Results show time saving for the same programs running incrementally on their new versions