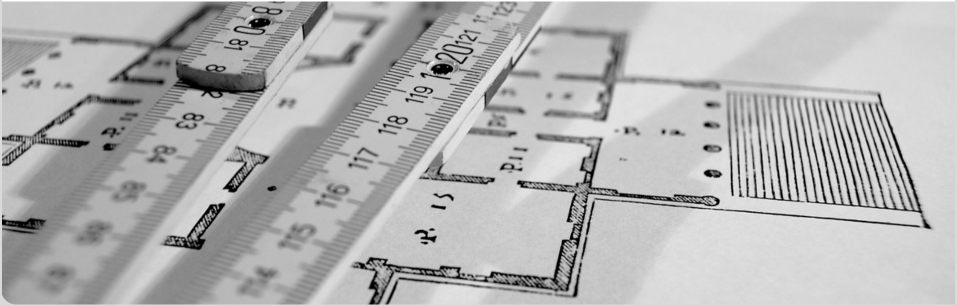
A Comparative Study of Incremental Constraint Solving Approaches in Symbolic Execution

Presentation at HVC2014, Haifa.

Tianhai Liu[1], Mateus Araújo[2], Marcelo d'Amorim[2], and Mana Taghdiri[1] | November 18, 2014

[1]KARLSRUHE INSTITUTE OF TECHNOLOGY, GERMANY || [2]FEDERAL UNIVERSITY OF PERNAMBUCO, BRAZIL



Introduction

Karlsruhe Institute of Technology

Symbolic Execution

- introduced 40 years ago.
- automatically generate high-coverage tests for complex software.
- popular and practical in recent years.
- various tools exist, e.g., KLEE, SPF, PEX.

Bottlenecks

- space cost: space exploration in path exploration.
- time cost: constraint solving in the solver.

We focus on reducing the **time cost**.

Introduction by an Example

Code

```
int biggest(int a,
    int b,int c){
  int res = 0;
  if (a < b)
    res = b;
  if (res < c)
    res = c;
  return res;
}
biggestTest(){
  x,y,z:=sym_input();
  w:=biggest(x,y,z);
  assert(w>=x);
}
```

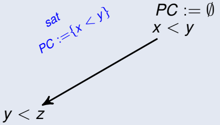
Symbolic Execution Tree

Introduction by an Example

Code

```
int biggest(int a,
    int b,int c){
  int res = 0;
  if (a < b)
    res = b;
  if (res < c)
    res = c;
  return res;
}
biggestTest(){
  x,y,z:=sym_input();
  w:=biggest(x,y,z);
  assert(w>=x);
}
```

Symbolic Execution Tree

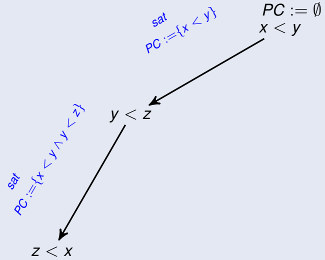


Introduction by an Example

Code

```
int biggest(int a,
    int b,int c){
  int res = 0;
  if (a < b)
    res = b;
  if (res < c)
    res = c;
  return res;
}
biggestTest(){
  x,y,z:=sym_input();
  w:=biggest(x,y,z);
  assert(w>=x);
}
```

Symbolic Execution Tree

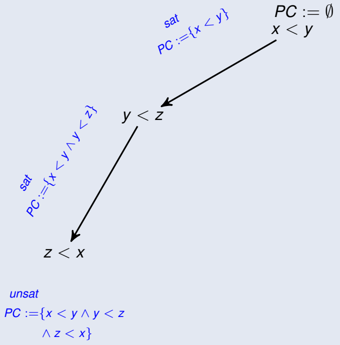


Introduction by an Example

Code

```
int biggest(int a,
    int b,int c){
  int res = 0;
  if (a < b)
    res = b;
  if (res < c)
    res = c;
  return res;
}
biggestTest(){
  x,y,z:=sym_input();
  w:=biggest(x,y,z);
  assert(w>=x);
}
```

Symbolic Execution Tree



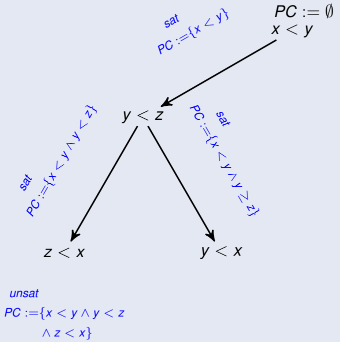
Introduction by an Example

Karlsruhe Institute of Technology

Code

```
int biggest(int a,
    int b,int c){
  int res = 0;
  if (a < b)
    res = b;
  if (res < c)
    res = c;
  return res;
}
biggestTest(){
  x,y,z:=sym_input();
  w:=biggest(x,y,z);
  assert(w>=x);
}
```

Symbolic Execution Tree

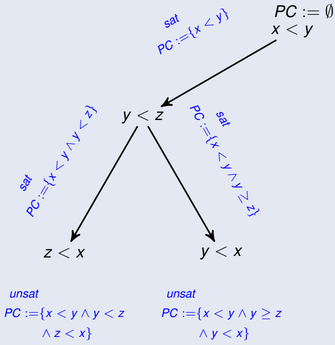


Introduction by an Example

Code

```
int biggest(int a,
    int b,int c){
  int res = 0;
  if (a < b)
    res = b;
  if (res < c)
    res = c;
  return res;
}
biggestTest(){
  x,y,z:=sym_input();
  w:=biggest(x,y,z);
  assert(w>=x);
}
```

Symbolic Execution Tree

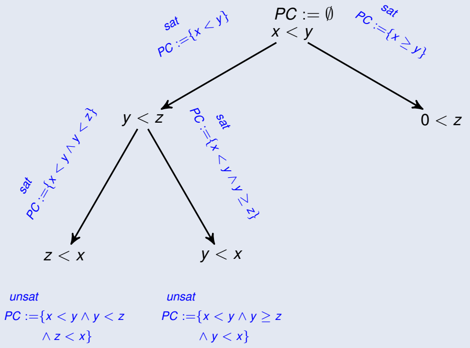


Introduction by an Example

Code

```
int biggest(int a,
    int b,int c){
  int res = 0;
  if (a < b)
    res = b;
  if (res < c)
    res = c;
  return res;
}
biggestTest(){
  x,y,z:=sym_input();
  w:=biggest(x,y,z);
  assert(w>=x);
}
```

Symbolic Execution Tree

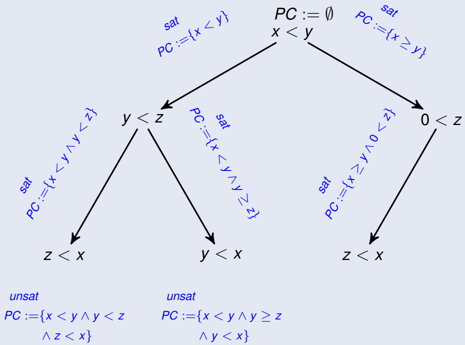


Introduction by an Example

Code

```
int biggest(int a,
    int b,int c){
  int res = 0;
  if (a < b)
    res = b;
  if (res < c)
    res = c;
  return res;
}
biggestTest(){
  x,y,z:=sym_input();
  w:=biggest(x,y,z);
  assert(w>=x);
}
```

Symbolic Execution Tree

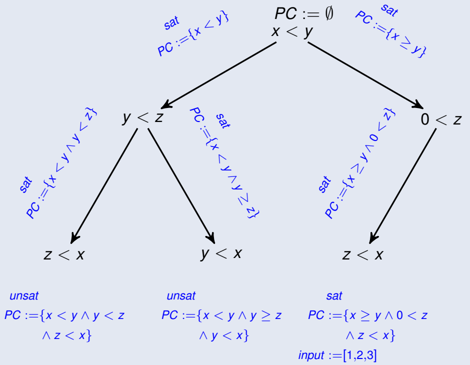


Introduction by an Example

Code

```
int biggest(int a,
    int b,int c){
  int res = 0;
  if (a < b)
    res = b;
  if (res < c)
    res = c;
  return res;
}
biggestTest(){
  x,y,z:=sym_input();
  w:=biggest(x,y,z);
  assert(w>=x);
}
```

Symbolic Execution Tree



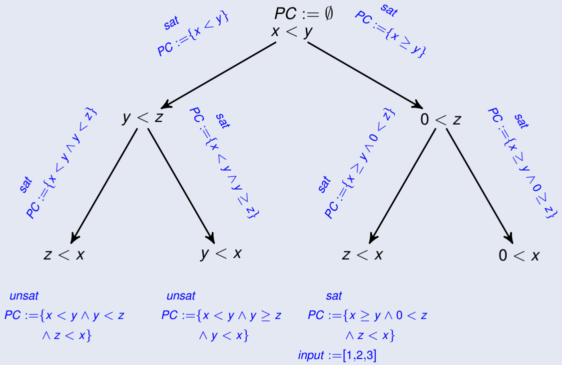
Introduction by an Example

Karlsruhe Institute of Technology

Code

```
int biggest(int a,
    int b,int c){
  int res = 0;
  if (a < b)
    res = b;
  if (res < c)
    res = c;
  return res;
}
biggestTest(){
  x,y,z:=sym_input();
  w:=biggest(x,y,z);
  assert(w>=x);
}
```

Symbolic Execution Tree

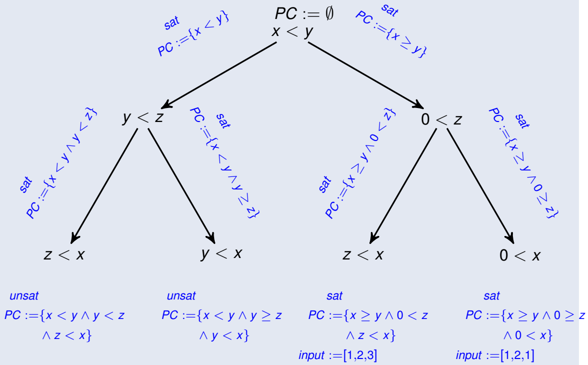


Introduction by an Example

Code

```
int biggest(int a,
    int b,int c){
  int res = 0;
  if (a < b)
    res = b;
  if (res < c)
    res = c;
  return res;
}
biggestTest(){
  x,y,z:=sym_input();
  w:=biggest(x,y,z);
  assert(w>=x);
}
```

Symbolic Execution Tree



Motivation

Karlsruhe Institute of Technology

Reducing the cost of constraint solving

- consecutively generated PCs are similar.
- they are suitable for incremental SMT solvers.

$\Longrightarrow$ compare various incremental constraint solving approaches.

Reducing the cost of path exploration

- symbolic states are destroyed on decision points.
- new PC is constructed on each decision point.

$\Longrightarrow$ How about constructing path-conditions before path exploration?

Cache-based Techniques

***caching*: reuse solutions of solved constraints.**

- simplify new constraints with independent clauses optimization.
- introduce cost of partitioning constraints and operating cache.
- popular used in tools, e.g., KLEE, SPF, PEX

***cachingOpt*: same to *caching* but building partitions incrementally.**

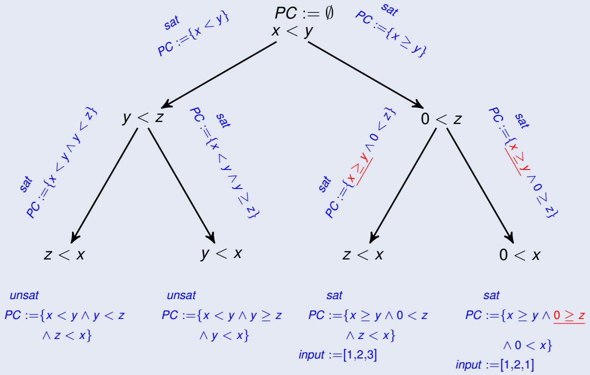
- build new partition by combining new constraint with dependent partitions.
- introduce overhead in combination.

Cache-based Techniques

Code

```
int biggest(...){
  int res = 0;
  if (a < b)
    res = b;
  if (res < c)
    res = c;
  return res;
}
  ...
```

Symbolic Execution Tree



Stack-based Techniques

stack

- use incremental SMT solving.
- create new frame on the assertion stack of the solver for each query.
- build new path-conditions in path exploration.

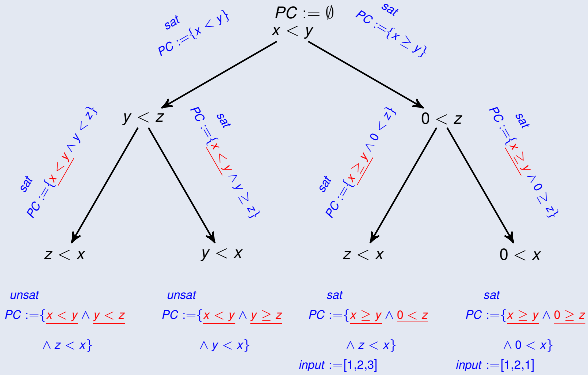
Technique *stack*

Karlsruhe Institute of Technology

Code

```
int biggest(...){
  int res = 0;
  if (a < b)
    res = b;
  if (res < c)
    res = c;
  return res;
}
  ...
```

Symbolic Execution Tree



Stack-based Techniques

Optimization: *stackOpt*

- same to *stack* but with common sub-expression elimination(CSE).
- find shared expressions outside of constraint solvers.
- introduce new variables referring to shared expressions.

Evaluation

Research Questions

- How cache-based and stack-based approaches compare?
- What is the benefit of using common sub-expressions elimination?
- Where each technique spends most time?

Benchmarks

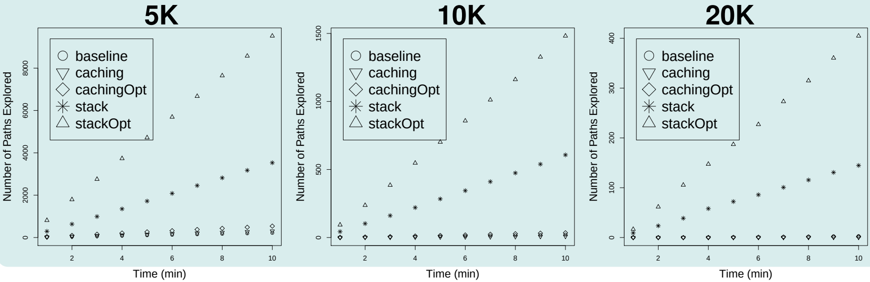
- **RUGRAT**: 300 programs of 5, 10, 20 KLOC, 100 programs for each program size. ($\sim$4300 KLOC together).
- **KLEE**: 96 Unix programs, e.g., cat, sed. ($\sim$4.5 KLOC together)

Environment

- Intel CPU with 2.60 GHz running 64-bit openSUSE
- 8GB max heap size for each running

RQ1: Cache* VS Stack* on RUGRAT

Number of paths explored in 10min



Observations

- incremental SMT solving is beneficial.
- fewer paths are generated when exploring longer programs.
- linear x-y relationship in the average plots.

RQ1: Cache* VS Stack* on KLEE

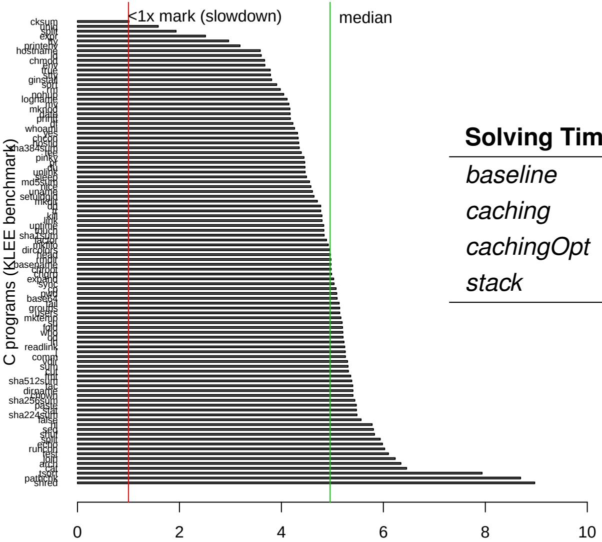
Constraint Generation of KLEE

- use KLEE's default configuration[a]
- run KLEE on 96 programs from KLEE Coreutils benchmark (total 101 programs).
- KLEE spent ∼27s ($90\% = 27s/30s$) in constraint solving.
- *stack* spends average $7.53s$. (best case $0.14s$, worst case $72.36s$, median $6.3s$)
- all $91/96$ programs had been solved under $10s$.

[a] `http://klee.github.io/klee/klee-tools.html#klee-stats`

RQ1: Cache* VS Stack* on KLEE

Speedup (x times quicker than best other)

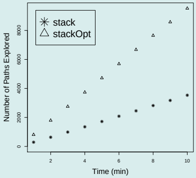


Solving Time (ms) per query	
baseline	34.559
caching	26.231
cachingOpt	25.860
stack	7.260

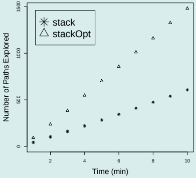
RQ2: Effects of Common Sub-expression Elimination

Number of paths explored for RUGRAT benchmark in 10min

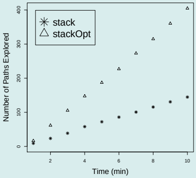
5K



10K



20K



Discussion

- *stackOpt* reuses the constraints built before path exploration, while *stack* constructs new constraint on each node.
- SMT solver finds shared expressions at SMT formulas (syntactical) level, while *stackOpt* finds them at Java (semantic) level.

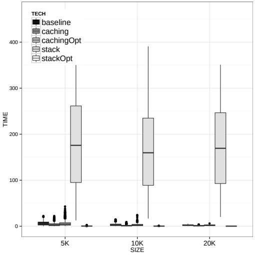
RQ3: Time-breakdown

Runtime Cost

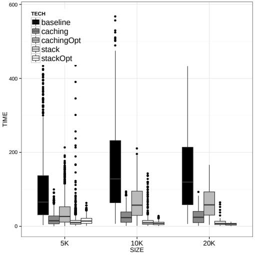
- *path exploration*: exploring paths.
- *expression construction*: creating Z3 internal expressions.
- *constraint solving*: solving and caching constraints.
- *rest*: others, e.g. code transformation.

RQ3: Time-breakdown

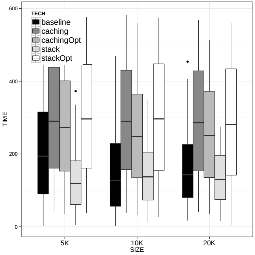
Path Exploration



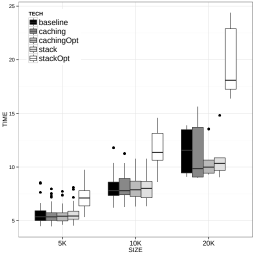
Expression Construction



Constraint Solving



Rest



Infrastructure

Implementation

- use Soot to parse Java source/binary code.
- use Jung to construct symbolic execution tree.
- use InspectJ to unroll loops and inline methods.
- use Z3 API to solve constraints.

Related Works

Karlsruhe Institute of Technology

Caching schemes

- Cadar *et al.* proposed several optimizations to simplify constraints prior to invoking the solver in KLEE. In addition, KLEE checks constraints with a potential solution.
- Visser *et al.* proposed GREEN to share results of symbolic executions across different environments. Our contributions are complementary to GREEN to speedup constraint solving.

Incremental SMT solving

- Incremental SMT solving is used in some situations. e.g., solving scheduling problems.
- Wieringa proposed a technique to enhance the clause learning by executing the solver in multiple threads.

Conclusion

Conclusion

- Incremental constraint solving is important in SE.
- Stack-based approaches provides superior results compared to cache-based approaches in our experiments.

Challenges

- Integrate syntactic and semantic approaches. E.g., how to get semantic information from SMT solvers?
- Handle the theories not supported in SMT solvers efficiently. E.g., how to represent floating-point numbers in SMT solvers?

Reproduce Experiments

http://asa.iti.kit.edu/130_392.php