Benchmarking Symbolic Execution Using Constraint Problems - Initial Results

Sahil Verma, IIT Kanpur
Roland Yap, National University of Singapore

November 4, 2019

Overview

Motivation

Constraint Satisfaction Problems (CSP)

Transforming CSP to C

Experiments

Conclusion

Motivation

- Symbolic execution engines have played phenomenal role in program analysis
- We aim to improve symbolic execution via standardized benchmarking
- We are motivated by how “hard benchmarks” from SAT competition has driven the development of SAT solvers.

High-level idea

- We propose synthetic benchmarks which exercise the core reasoning techniques
- We propose to use discrete combinatorial problems in form of constraint satisfaction problems (CSP)
- We transform the CSPs' into C code using various possible transformations to construct the benchmarks

CSP background

- A Constraint Satisfaction Problem(CSP) P is a pair (X,C) where X is a set of n variables $x_1, \ldots, x_n$ and C a set of e constraints $c_1, \ldots, c_e$
- In a finite domain CSP, variables $x \in X$ take values from its domain D(x) which is a finite set of values
- The general form of a finite domain CSP is NP-complete

Constraint forms

- Constraints can either be defined implicitly or in form of explicit values
- When constraints are implicitly defined in form of linear arithmetic relations over variables in X, they are termed as intensional constraints

$$< constraint > equal([\%0],\ \ sum([\%1],[\%2])) < constraint >$$
$$< args > x[0]\ \ x[0]\ \ x[1]\ \ \ < /args >$$
$$< args > x[1]\ \ x[1]\ \ x[2]\ \ \ < /args >$$

Intensional Constraint

Constraint forms

- Constraints can be defined explicitly by providing a list of values some set of variables satisfy (positive tables) or do not satisfy (negative tables)

$$< positive > (0,0,0)\ \ (0,1,0) < /positive >$$
$$< args > x[0]\ \ x[1]\ \ x[2]\ \ < /args >$$
$$< args > x[3]\ \ x[4]\ \ x[5]\ \ < /args >$$

Extensional Constraint

Transforming CSP to C

- The finite domain variables of the CSP correspond to integer variables in the program P'
- The C variables are made symbolic
- The constraint relations are encoded into conditional statements in the program (if, else) or as 'assume' statement

Transforming CSP to C

- We generate several equisatisfiable transformations for a CSP
- The transformations differ in the C constructs, the constraint grouping aggressiveness and the operators used for grouping

Type	Versions	Construct	Operator	Grouped
Extensional	1	`if`	*logical*	*no*
	2	`if`	*logical*	*yes*
	3	`if`	*bitwise*	*no*
	4	*assume*	*bitwise*	*yes*
	...			
Intensional	1	`if`	*NOP*	*no*
	2	*assume*	*logical*	*yes*
	3	*assume*	*bitwise*	*no*
	4	*assume*	*bitwise*	*yes*
	...			

Distinguishing SAT and UNSAT CSPs

- When all constraints have been transformed, we place an ‘assert(0)’ at the end of the program to trigger a failure flag
- If the CSP P is satisfiable, the program terminates with an assertion failure, and if it is unsatisfiable, the statement is unreachable.
- We did not find any contradictions in our experiments

Experimental Setup

- We generate 12 extensional and 10 intensional versions
- We experiment with 6 classes, 3 from intensional and 3 from extensional category totalling 119 programs
- We use KLEE (with both STP and Z3 as solvers), Tracer-X, LLBMC as execution engines. Our baseline was AbsCon, a common tool for solving CSPs

Experimental Setup

- We use the symbolic execution engine's runtime for a transformed program P' as a measure of the tool's robustness over transformations
- We use the number of timeouts for benchmarks with a size parameter as a measure of tool's scalability with increasing size of programs

Results

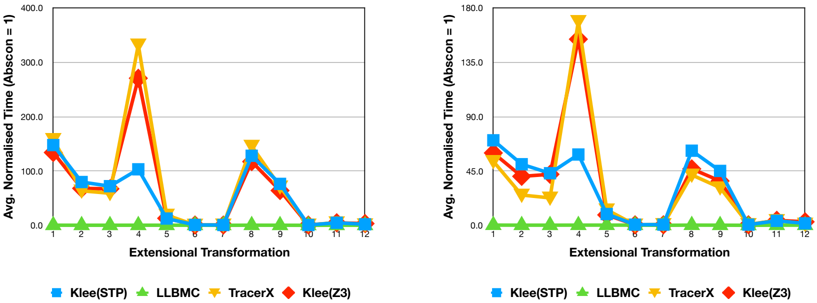

Figure: Robustness of different tools for an extensional class benchmark (AIM-100 satisfiable and unsatisfiable)

Results

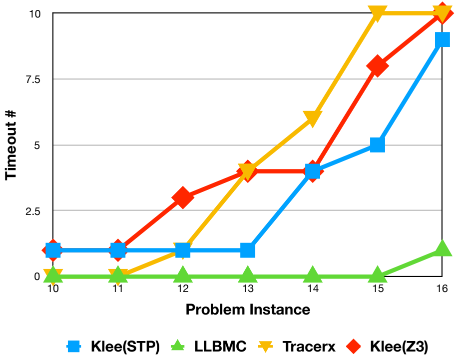

Figure: Scalability of different tools for an intensional class benchmark (CostasArrays)

Conclusions

- The results show that their certainly is a significant amount of fragility in the tools which could be addressed
- The results also show that different solvers behave differently when used in KLEE, but in an unpredictable fashion
- Tracer-X which uses interpolation to remove execution paths was faster for some benchmarks and slower for some, which points towards prediction techniques which would help direct its usage