Pruning the Search Space in Path-based Test Generation

Sébastien Bardin
sebastien.bardin@cea.fr
CEA-LIST, Software Security Labs

(joint work with Philippe Herrmann)

Context

Automatic test data generation from source code (STDG)

The test suite must achieve a global structural coverage objective

- all instructions, all branches, etc.

Do not consider the oracle generation issue : assume an external automatic oracle

- perfect oracle (back-to-back testing)
- partial oracle (assertions / contracts)

Symbolic Execution

Symbolic Execution (SE) is a very fruitful approach for STDG

- efficiency
- robustness

SE in a nutshell

Constraint-based reasoning : translate a part of the program into a logical formula φ, such that a solution of φ is a relevant TD

Path-based approach : focus on a single path at once + enumerate (bounded) paths

- simple formulas, only conjunctions (no quantifier / fixpoint)

Concolic paradigm : combination of symbolic and dynamic execution

- robustness to “difficult-to-model” programming features

A few prototypes

PathCrawler (CEA)	2004
Dart (Bell Labs), Cute (Uni. of Illinois / Berkeley)	2005
Exe (Stanford)	2006
Jpf (NASA)	2007
Osmose (CEA), Sage (Microsoft), Pex (Microsoft)	2008

Main Limitations

Two major bottlenecks for Symbolic Execution

1. constraint solving (along a single path)
2. # paths

Path explosion phenomenon

- nesting loops and conditional instructions
- inlining of function calls

Moreover : SE require a user-defined path-bound *k*

- things get worse if *k* is over-estimated
- sometimes, very long paths to exhibit specific behaviours

Our goal : lower the path explosion in SE

Not all Paths are Relevant for STDG

Irrelevant paths

- In practice, SE enumerates all k-paths
- But the true goal is to cover "items" (instr., branches)
- Some paths are very unlikely to improve the current coverage

Idea : detect a priori irrelevant paths to discard them and lower the path explosion

Our results

1. three complementary heuristics to prune likely redundant paths
2. implementation in the OSMOSE tool and experiments

Outline

Motivation
SE
Heuristics
Experiments
Conclusion

- Context
- Symbolic Execution
- Heuristics
- Experiments
- Conclusion

Path Predicate

π a finite path of the program P
D the input space of P
$V \in D$ an input vector

Path predicate

A path predicate for π is a formula φ_π interpreted on D s.t. if $V \models \varphi_\pi$ then the execution of P on V exercices π at runtime.

More formally : let $\pi = \xrightarrow{t_1} \xrightarrow{t_2} \ldots \xrightarrow{t_n}$

- the greatest path predicate

  $\bar{\varphi_\pi} = wpre(t_1, wpre(t_2, \ldots wpre(t_n, \top)))$

- a path predicate

  φ_π such that $\varphi_\pi \Rightarrow \bar{\varphi_\pi}$

A path predicate is typically computed via strongest postcondition

Framework of Symbolic Execution

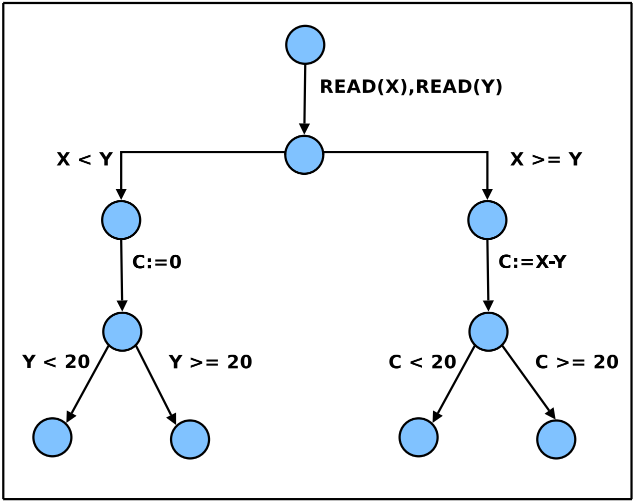
Path-based test data generation

1. choose an uncovered (k-bounded) path π
2. compute one of its path predicates φ_π
3. solve φ_π : solution = TD exercising path π
4. update coverage, if still something to cover then goto 1

Parameter 1 - Logical theory : not relevant here

Parameter 2 - Path enumeration strategy : here, standard DFS

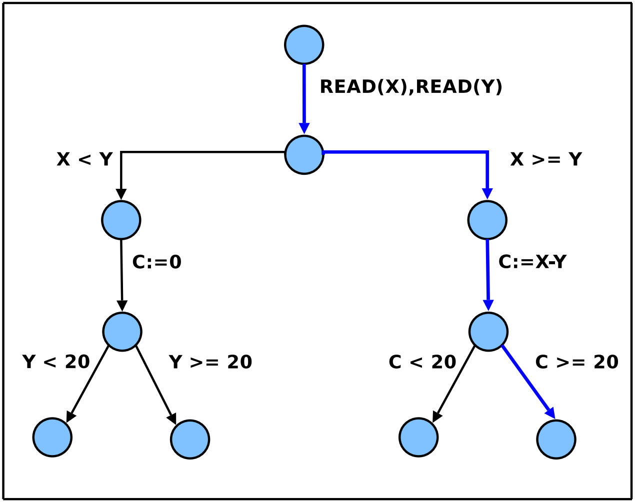
Extension - Concolic execution



choose path

compute path predicate, solve it, update cover

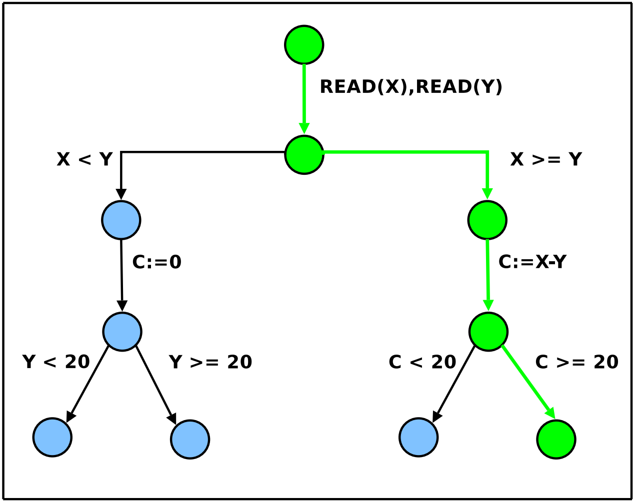
choose the next path by DFS backtracking, and so on



choose path

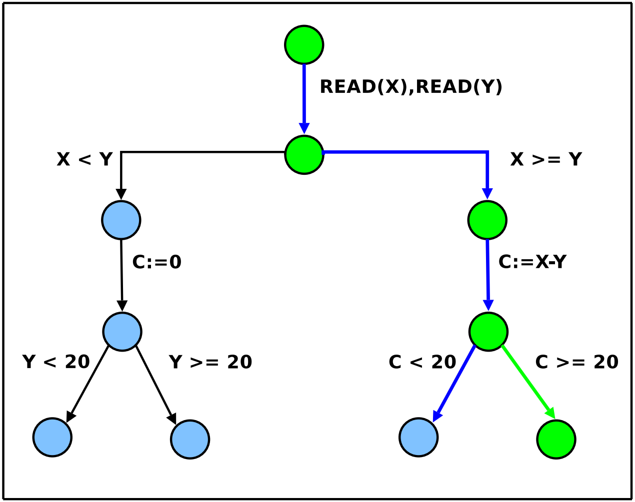
compute path predicate, solve it, update cover

choose the next path by DFS backtracking, and so on



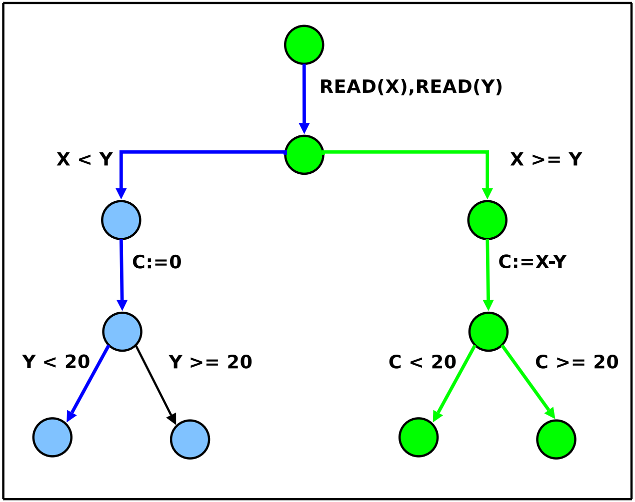
- choose path
- compute path predicate, solve it, update cover
- choose the next path by DFS backtracking, and so on

Symbolic Execution, Basic Procedure (BP)



choose path
compute path predicate, solve it, update cover
choose the next path by DFS backtracking, and so on

Symbolic Execution, Basic Procedure (BP)



- choose path
- compute path predicate, solve it, update cover
- choose the next path by DFS backtracking, and so on

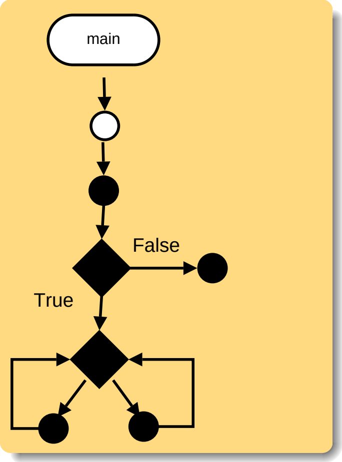
Outline

- Context
- Symbolic Execution
- Heuristics
- Experiments
- Conclusion

Procedure BP tries to cover a new path at each iteration

BUT this new path does not necessarily cover new items

- the resolution time is wasted
- more useless paths will be explored from this prefix



On the example, full coverage requires at most 3 TD, while there are $\approx 2^{k+1}$ paths of length $\leq k$

Idea

Check if uncovered items may be reached from the current instruction. If not, solve the current prefix but do not expand it

Optimistic check based on the CFG abstraction of the program

The Look-Ahead heuristic enjoys nice properties

- soundness : discard only redundant paths
- relative completeness : BP+LA achieves always the same coverage than BP
- path reduction : BP+LA explores always less path than BP

Difficulty : efficient computation of the (CFG) reachability set

Reachability Set Computation

Procedure `ReachSet : node` $\rightarrow$ `Set(node)`

Standard worklist algorithm has the following problems in our context

- all reachability sets are computed at the same time, even if BP will not use all of them
- not designed for interprocedural or context-sensitive analysis

Efficient interprocedural analysis

Efficient computation

- lazy computation
- computation cache

Interprocedural analysis

- compact representation of sets of nodes : manipulate CFG nodes and Call Graph (CG) nodes
- function summaries : propagate reachable CG nodes (from CG)
- lazy computation and computation cache extend to CG

Context-sensitive analysis

the current stack is passed as an argument, if the current node can reach a `ret` instruction, then the procedure is recursively launched on the top of the stack (return site)

`ReachSet-context(node,stack, rset)` :

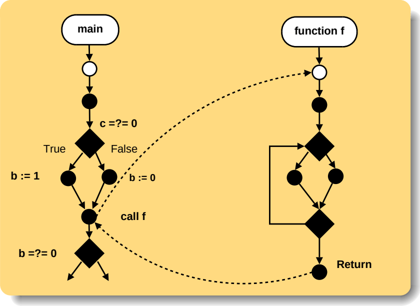
- `c` := `ReachSet(node)` ; `r` := `c` ∪ `rset`
- if (stack.empty or `ret` ∉ `c`) then return `r` ;
- else return `ReachSet-context(stack.top,stack.tail, r)`

Remark : the computation cache is still a map from *node* to *set*, rather than a map from (*node*, *stack*) to *set*

Nested function calls are often the major source of path explosion

BP explores all the paths in callees

But in unit testing, need to cover only paths of the top-level function



Example : only two TD to cover the `main` function, but $\approx 2^{k+1}$ paths (claim) top-level paths rarely depend only on specific behaviours in deep function calls

MCD heuristic : prevent backtracking in deep nested function calls

Implementation : a user-defined `mcd` parameter, a counter `depth` updated by `call` and `ret`, performs branching only if `depth` $\leq$ `mcd`

Theoretically : take care, the MCD heuristic is not sound

Empirically : experimental results show a very large pruning and no loss in coverage (see after)

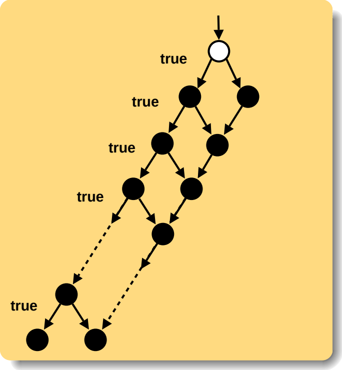
Heuristic 3 : Solve-First (SF)

DFS has two main drawbacks in our context

- if # TD is limited, DFS focuses only on a deep narrow portion of the program (slow coverage speed)
- longer (and more complex ?) prefixes are solved first

Example : assume #node = 2n+1, all paths are feasible,
goal = instruction coverage

- only two TD are necessary
- BP+LA : n+1 TD



Idea

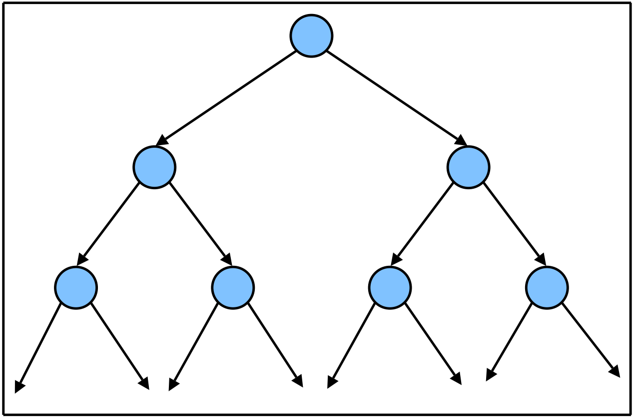
Slight modification of the concolic DFS procedure

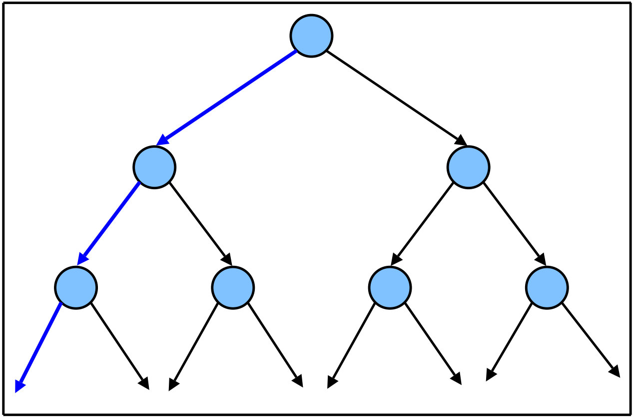
- on a choice point, choose which branch B1 will be followed (symbolically) first
- immediately solve the other branch B2 (TD2), execute TD2 and update coverage info, store TD2
- execute symbolically the procedure through branch B1 (as usual)
- when backtracking through B2, TD2 can be retrieved if needed

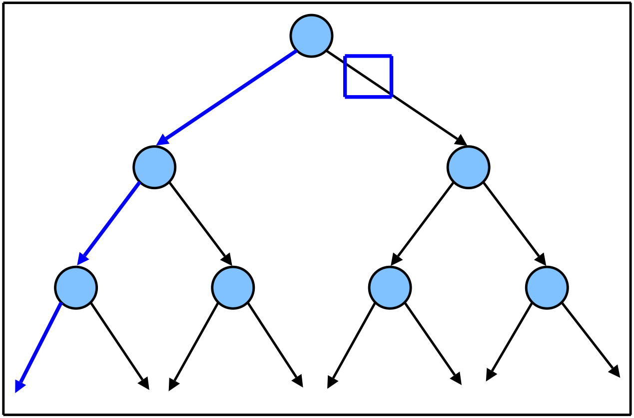
Mostly the DFS symbolic execution, except than along a given prefix, every alternative branch has been concretely expanded once

- minimal overhead
- along a path, shorter prefixes are solved first
- some distant portion of the program (in a DFS ordering) are exercised very early

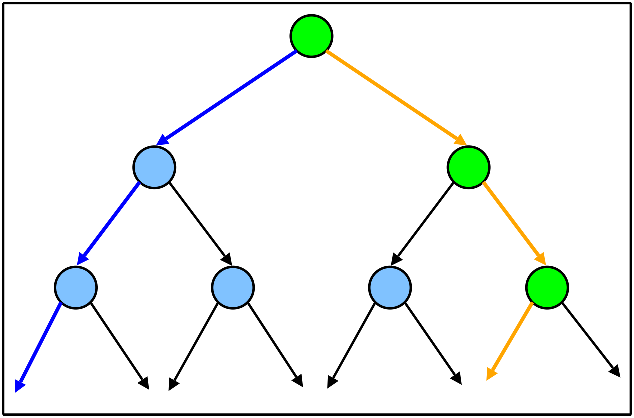
Idea (2)



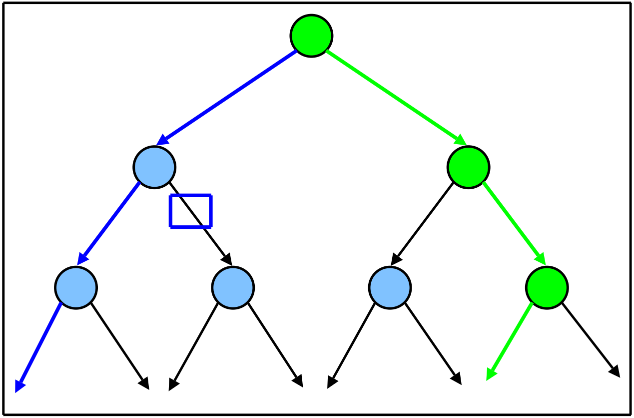


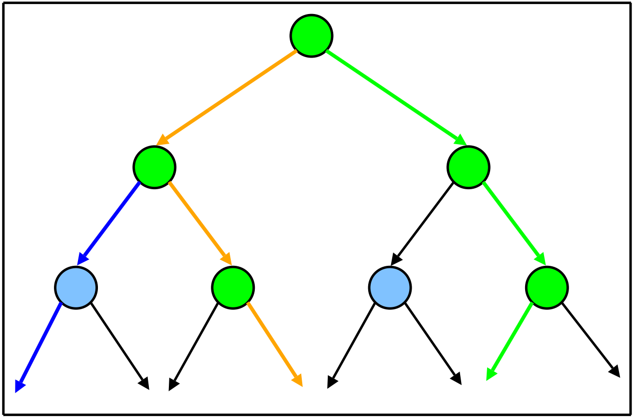


Idea (2)

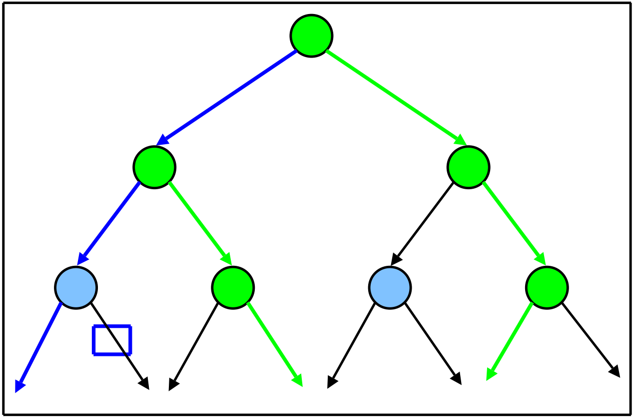


Idea (2)

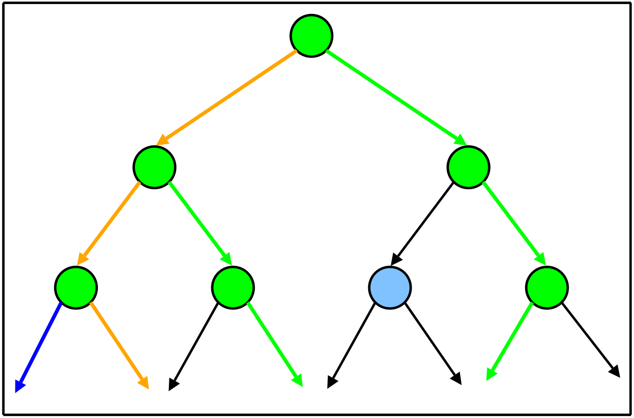




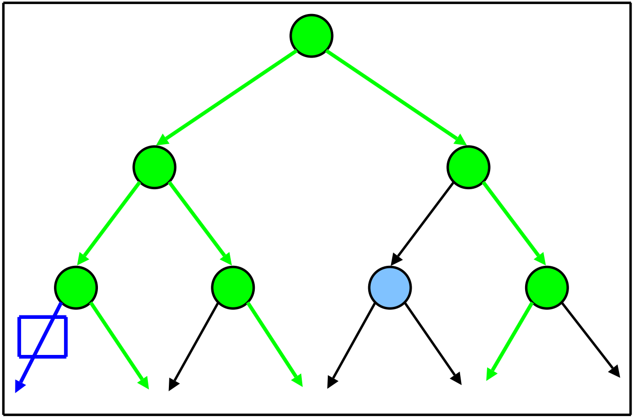
Idea (2)



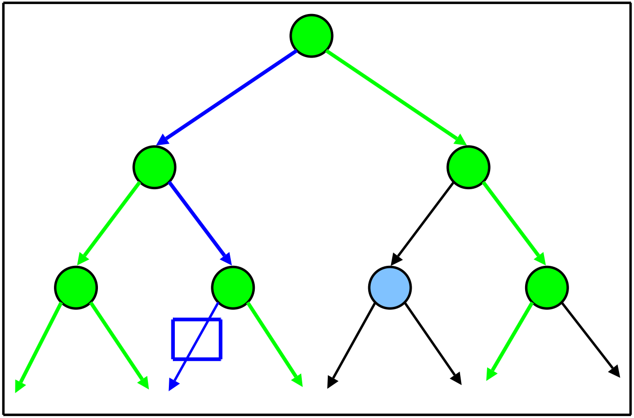
Idea (2)



Idea (2)



Idea (2)



Summary

	relative completeness	# path reduction	implementation in BP
Look-Ahead	yes	always	efficient reach. test
Max-CallDepth	no	not sure	easy
Solve-First	yes	not sure	easy (concolic setting)

Outline

- Context
- Symbolic Execution
- Heuristics
- Experiments
- Conclusion

About experiments

Heuristics implemented in the OSMOSE tool (SE for executable files)
Small C programs cross-compiled to C509 and PPC architectures
Configuration : Intel Pentium M 2Ghz, RAM 1.2 GBytes, Linux

program	#I	#Br	#F	CD	# T
check-pressure	59	10	3	1	4
square 3x3	272	46	1	0	43
square 4x4	274	46	1	0	123
hysteresis	91	16	2	1	35
merge	56	24	3	1	70
triangle	102	38	5	3	15
ppc-square 4x4	226	30	1	0	125
ppc-hysteresis	76	16	2	1	251
ppc-merge	188	16	3	2	2
ppc-triangle	40	18	3	2	19

#I : n. of instructions
#Br : n. of branches
#F : n. of functions
CD : maximal call depth
#T : n. of tests (full Br cover)

Results

Notations : BP (Basic Procedure), UT (Unit Testing)

Comparisons

- BP+LA vs BP
- BP+UT+MCD vs BP+UT
- BP+SF vs BP

	average benefit (time | #path)	win-loss W/D/L	max benefit	max loss
LA	-57% | -57%	7/2/1 | 8/2/0	-80% | -85%	+4% | _
MCD	-85% | -72%	5/1/0 | 5/1/0	-97% | -80%	_ | _
SF+LA	-61% | -80%	4/0/5 | 5/0/4	-86% | -98%	+120% | +50%

Summary (2)

	theoretical		empirical	
	relative completeness	# path reduction	relative completeness	# path reduction
LA	yes	always	yes	-57%
MCD	no	not sure	yes	-72%
SF+LA	yes	not sure	yes	-80%

Other experiments

LA overhead : reachability set is computed, but test inclusion always answers yes

overhead	mean	variability
RS computed on backtrack only	+0%	+0% - +1%
RS computed at each branch	+2.4%	+0% - +7%

Outline

- Context
- Symbolic Execution
- Heuristics
- Experiments
- Conclusion

Related work (1)

Path enumeration strategy for better coverage speed

- best-first search (Exe, Sage, Pex) : active prefixes are ranked, and the best one is expanded
- hybrid search (Cute) : DFS + random

Redundant paths

- discard a path prefix if similar to an already expanded path prefix
  rwset (Exe), state caching / state abstraction (Jpf)
- discard a path prefix when it cannot reach an interesting state
  yogi and the Synergy approach

Concurrent systems and interleaving

- dynamic partial orders (Cute)

Functon calls

Techniques similar to MCD

- when the maximal depth is reached, a `call` returns $\top$ (JPF)
- function concretisation (CUTE) can also be used for path pruning

Other techniques

- lazy handling of function calls via uninterpreted symbols (SAGE)
- incremental construction of a summary function (DART)
- user-defined function specification (PATHCRAWLER)

Conclusion

We propose three heuristics to perform path pruning in Symbolic Execution

- easy to implement, whatever the path enumeration strategy is
- all the three techniques are complementary

Very encouraging results for Look-Ahead and Max-CallDepth on limited benchmarks

Solve-First shows a positive global gain, but much more variability

Future work

- experiments on larger programs and with other path search methods
- application to search-based testing ?