Binary-Level Testing of Embedded Programs

Sébastien Bardin

joint work with
P. Baufreton, N. Cornuet, P. Herrmann and S. Labbé

CEA LIST, Software Safety Lab (Paris area, France)

QSIC 2013

Overview

Focus on binary-level testing of safety-critical programs

We have been developing the OSMOSE tool since 2006
[ICST-08, ICST-09, TACAS-10, STVR-11]

- rely on Dynamic Symbolic Execution (DSE)
- first DSE tool over executable code, with SAGE [Godefroid-08]

Collaborations with industrial partners from energy and aeronautics

Overview

Focus on binary-level testing of safety-critical programs

We have been developing the OSMOSE tool since 2006
[ICST-08, ICST-09, TACAS-10, STVR-11]

- rely on Dynamic Symbolic Execution (DSE)
- first DSE tool over executable code, with SAGE [Godefroid-08]

Collaborations with industrial partners from energy and aeronautics

---

Contribution : between research and experience report

- original and practically-relevant features for DSE over safety-critical programs
- experience report on several case-studies
- (up-to-date description of OSMOSE)

Dynamic Symbolic Execution

- choose a path π of P
- compute a *path predicate* φ_π :

  $$\mathtt{v} \models \varphi_\pi \Rightarrow \mathtt{P}(\mathtt{v}) \text{ follows } \pi$$ [wpre, spost]

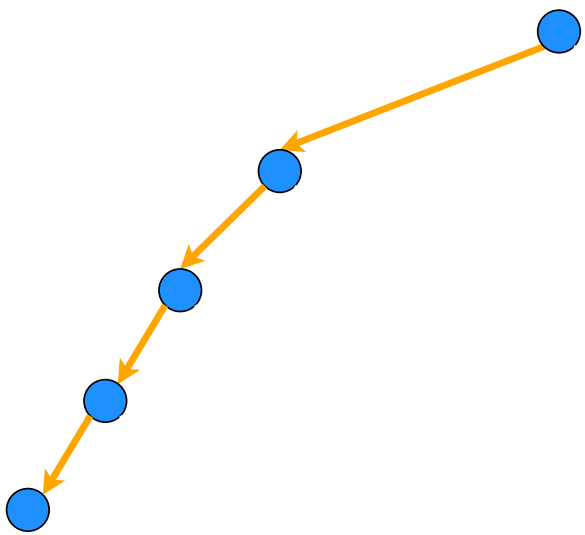
- solve φ_π for satisfiability
- SAT(s) ? get a new pair $<$ s, π $>$, update coverage
- loop until nothing more to cover

Dynamic Symbolic Execution

- choose a path π of P
- compute a *path predicate* φ_π :

  $$\mathtt{v} \models \varphi_\pi \Rightarrow \mathtt{P}(\mathtt{v}) \text{ follows } \pi$$ [wpre, spost]

- solve φ_π for satisfiability
- SAT(s) ? get a new pair $<$ s, π $>$, update coverage
- loop until nothing more to cover

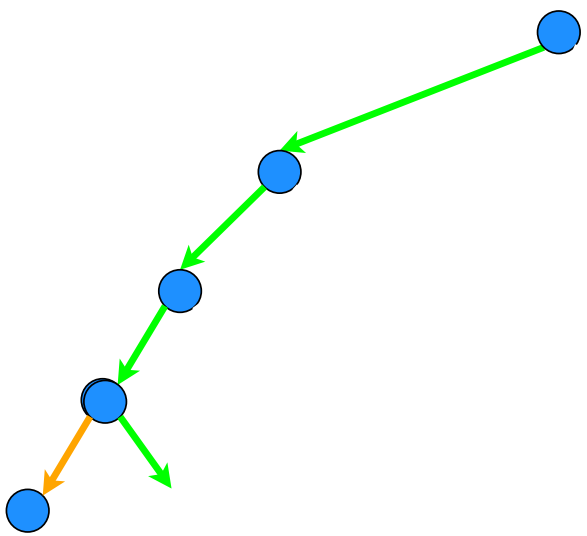


Dynamic Symbolic Execution

- choose a path π of P
- compute a *path predicate* φ_π :

  $$\mathrm{v} \models \varphi_\pi \Rightarrow \mathrm{P}(\mathrm{v}) \text{ follows } \pi$$ [wpre, spost]

- solve φ_π for satisfiability
- SAT(s) ? get a new pair $<$ s, π $>$, update coverage
- loop until nothing more to cover

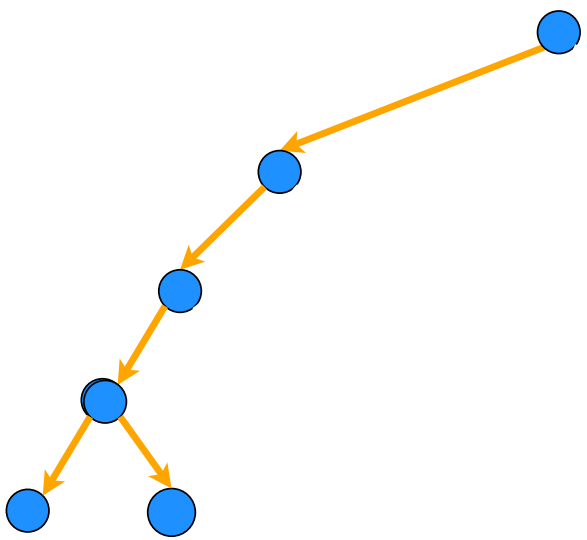


Dynamic Symbolic Execution

- choose a path π of P
- compute a *path predicate* φ_π :

  $$\mathtt{v} \models \varphi_\pi \Rightarrow \mathtt{P}(\mathtt{v}) \text{ follows } \pi$$ [wpre, spost]

- solve φ_π for satisfiability
- SAT(s) ? get a new pair $<$ s, π $>$, update coverage
- loop until nothing more to cover

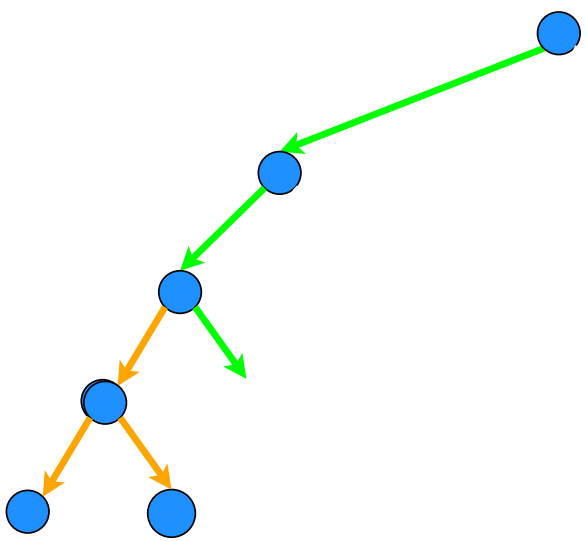


Dynamic Symbolic Execution

- choose a path π of P
- compute a *path predicate* φ_π :

  $$\mathtt{v} \models \varphi_\pi \Rightarrow \mathtt{P}(\mathtt{v}) \text{ follows } \pi$$ [wpre, spost]

- solve φ_π for satisfiability
- SAT(s) ? get a new pair $<$ s, π $>$, update coverage
- loop until nothing more to cover

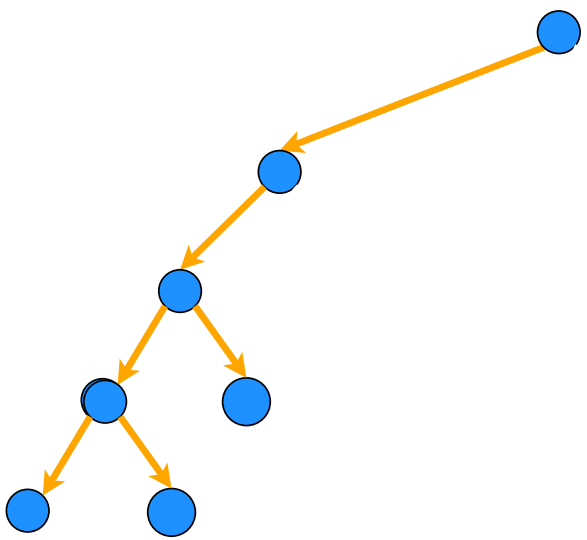


Dynamic Symbolic Execution

- choose a path π of P
- compute a *path predicate* φ_π :

  $$\mathtt{v} \models \varphi_\pi \Rightarrow \mathtt{P}(\mathtt{v}) \text{ follows } \pi$$ [wpre, spost]

- solve φ_π for satisfiability
- SAT(s) ? get a new pair $<$ s, π $>$, update coverage
- loop until nothing more to cover

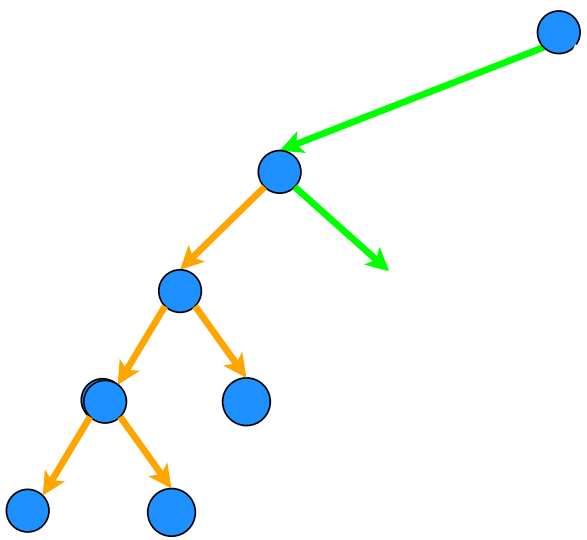


Dynamic Symbolic Execution

- choose a path π of P
- compute a *path predicate* φ_π :

  $$\mathtt{v} \models \varphi_\pi \Rightarrow \mathtt{P}(\mathtt{v}) \text{ follows } \pi$$ [wpre, spost]

- solve φ_π for satisfiability
- SAT(s) ? get a new pair $<$ s, π $>$, update coverage
- loop until nothing more to cover

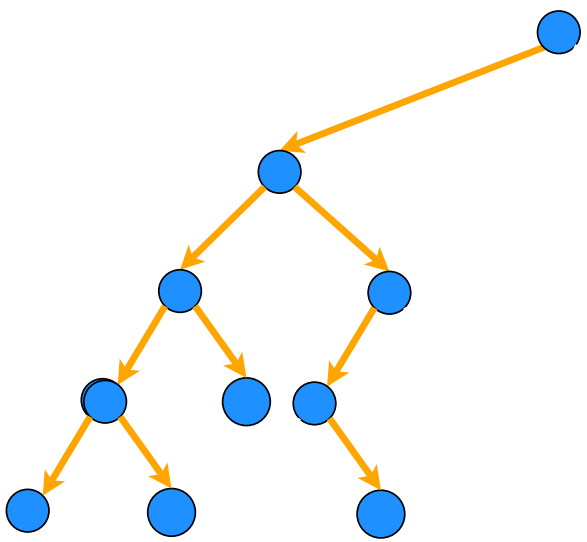


Dynamic Symbolic Execution

- choose a path π of P
- compute a *path predicate* φ_π :

  $$\mathtt{v} \models \varphi_\pi \Rightarrow \mathtt{P}(\mathtt{v}) \text{ follows } \pi$$ [wpre, spost]

- solve φ_π for satisfiability
- SAT(s) ? get a new pair $<$ s, π $>$, update coverage
- loop until nothing more to cover

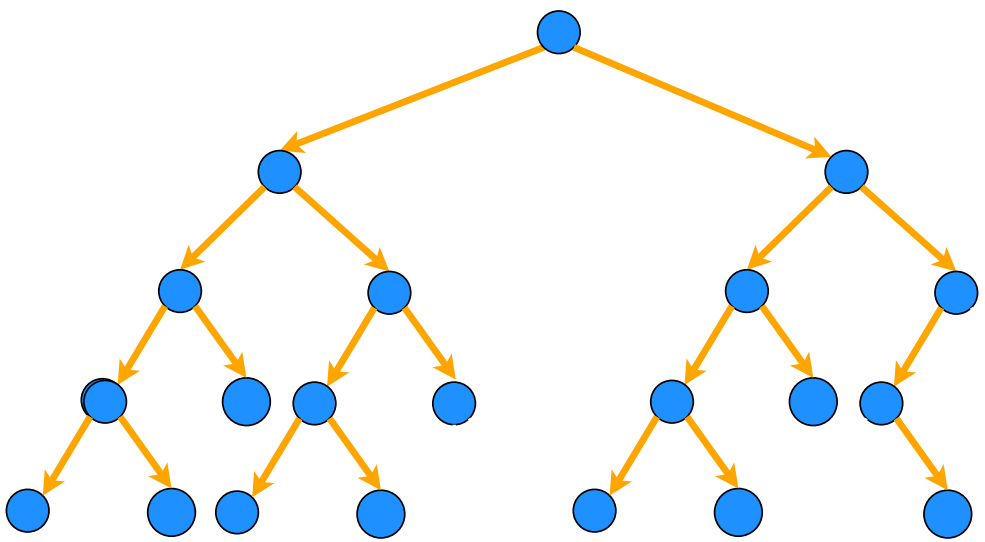


Dynamic Symbolic Execution

- choose a path π of P
- compute a *path predicate* φ_π :

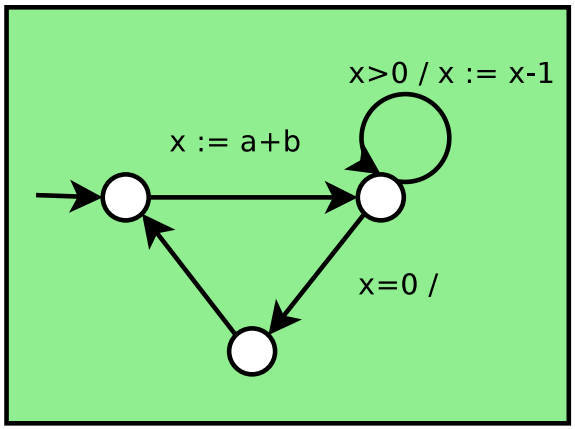
  $$\mathtt{v} \models \varphi_\pi \Rightarrow \mathtt{P}(\mathtt{v}) \text{ follows } \pi$$ [wpre, spost]

- solve φ_π for satisfiability
- SAT(s) ? get a new pair $<$ s, π $>$, update coverage
- loop until nothing more to cover



Binary-level software analysis

Model



Source code

```
int foo(int x, int y) {
 int k= x;
 int c=y;
 while (c>0) do {
   k++;
   c--;}
 return k;
}
```

Assembly

```
_start:
  load  A 100
  add B A
  cmp B 0
  jle label

label:
  move @100 B
```

Executable

```
ABFFF780BD70696CA101001BDE45
145634789234ABFFE678ABDCF456
5A2B4C6D009F5F5D1E0835715697
145FEDBCADACBDAD459700346901
3456KAHA305G67H345BFFADECAD3
00113456735FFD451E13AB080DAD
344252FFAADBDA457345FD780001
FFF22546ADDAE989776600000000
```

Safety-Critical Programs

- Highly critical
- Reactive, embedded
- Very demanding certification processes

Safety-Critical Programs (2)

A nice class of programs

- no dynamic memory allocation, no dynamic thread creation
- smaller size, self-contained code (no huge libraries)

Typical program structure

- a (big) non-terminating main loop
  - read input, perform internal computations, update output
  - all other loops are statically bounded
- a few programing idioms, for example self-tests
  - `A:=0; assert(A == 0);`

Very strong validation requirements

- unit testing aims at very high coverage
- all uncovered objectives must be justified
- automated tools must come with some guarantees

Binary-Level Testing of Safety-Critical Programs

Motivation 1 : validation w/o any access to source code

- commercial off-the-shelf components
- legacy code

Motivation 2 : "compiler-aware" validation

- ex : aeronautics and optimizing compilers

Binary-Level Testing of Safety-Critical Programs

Motivation 1 : validation w/o any access to source code

- commercial off-the-shelf components
- legacy code

Motivation 2 : "compiler-aware" validation

- ex : aeronautics and optimizing compilers

Appealing, but more challenging than source code analysis

Challenges of binary code analysis

D1 : Low-level semantics of data

- machine arithmetic, bit-level operations, untyped memory
- ▶ difficult for any state-of-the-art formal technique

D2 : Low-level semantics of control

- no distinction data / instructions, dynamic jumps (`goto A`)
- no (easy) syntactic recovery of Control-Flow Graph (CFG)
- ▶ violate an implicit prerequisite for most formal techniques

D3 : Diversity of architectures and instruction sets

- support for many instructions, modelling issues
- ▶ tedious, time consuming and error prone

Challenges of binary code analysis

D1 : Low-level semantics of data

- machine arithmetic, bit-level operations, untyped memory
- ▶ difficult for any state-of-the-art formal technique

D2 : Low-level semantics of control

- no distinction data / instructions, dynamic jumps (`goto A`)
- no (easy) syntactic recovery of Control-Flow Graph (CFG)
- ▶ violate an implicit prerequisite for most formal techniques

D3 : Diversity of architectures and instruction sets

- support for many instructions, modelling issues
- ▶ tedious, time consuming and error prone

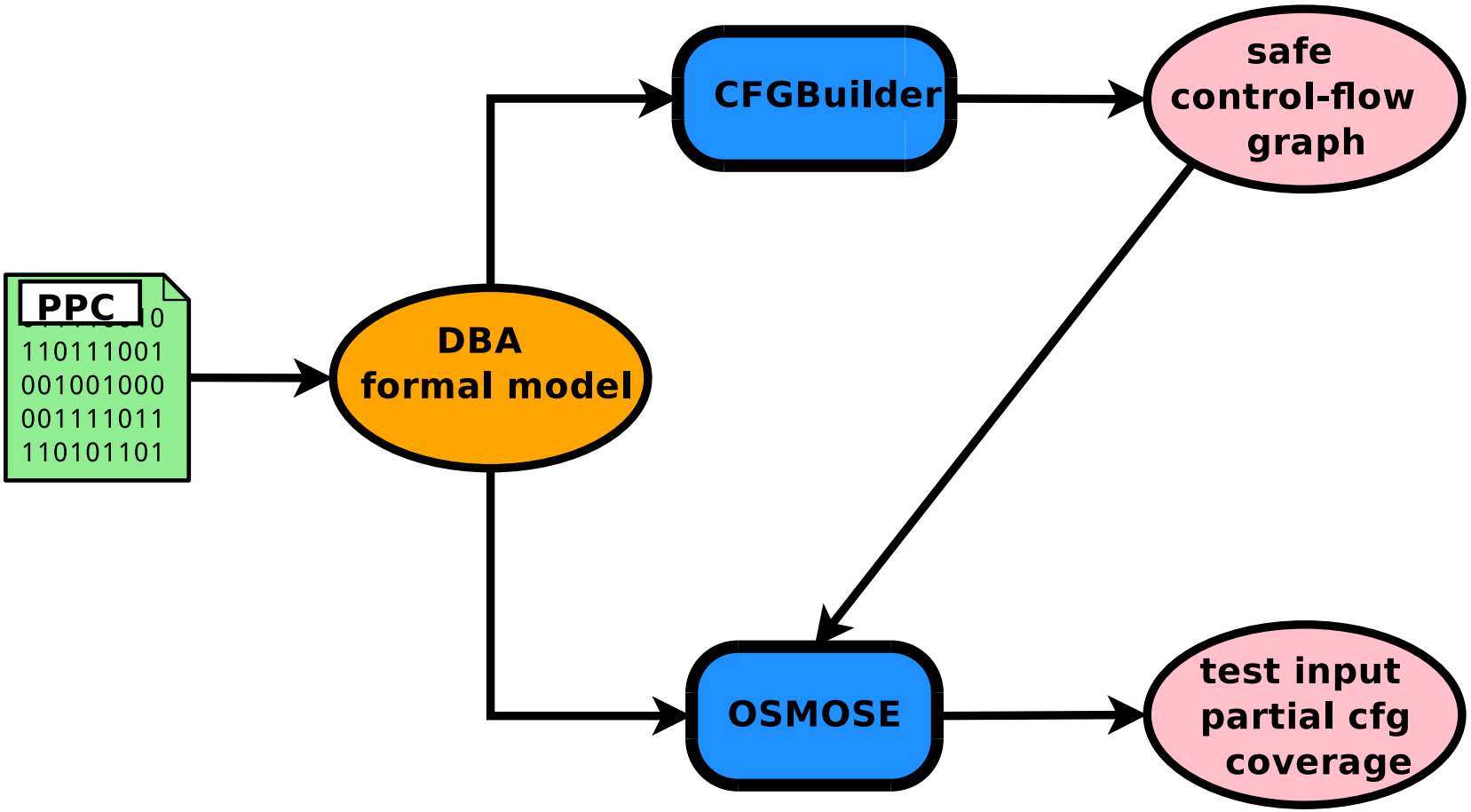
Outline

- Introduction
- The OSMOSE tool
- New features for DSE over safety-critical programs
- Case-studies
- Conclusion

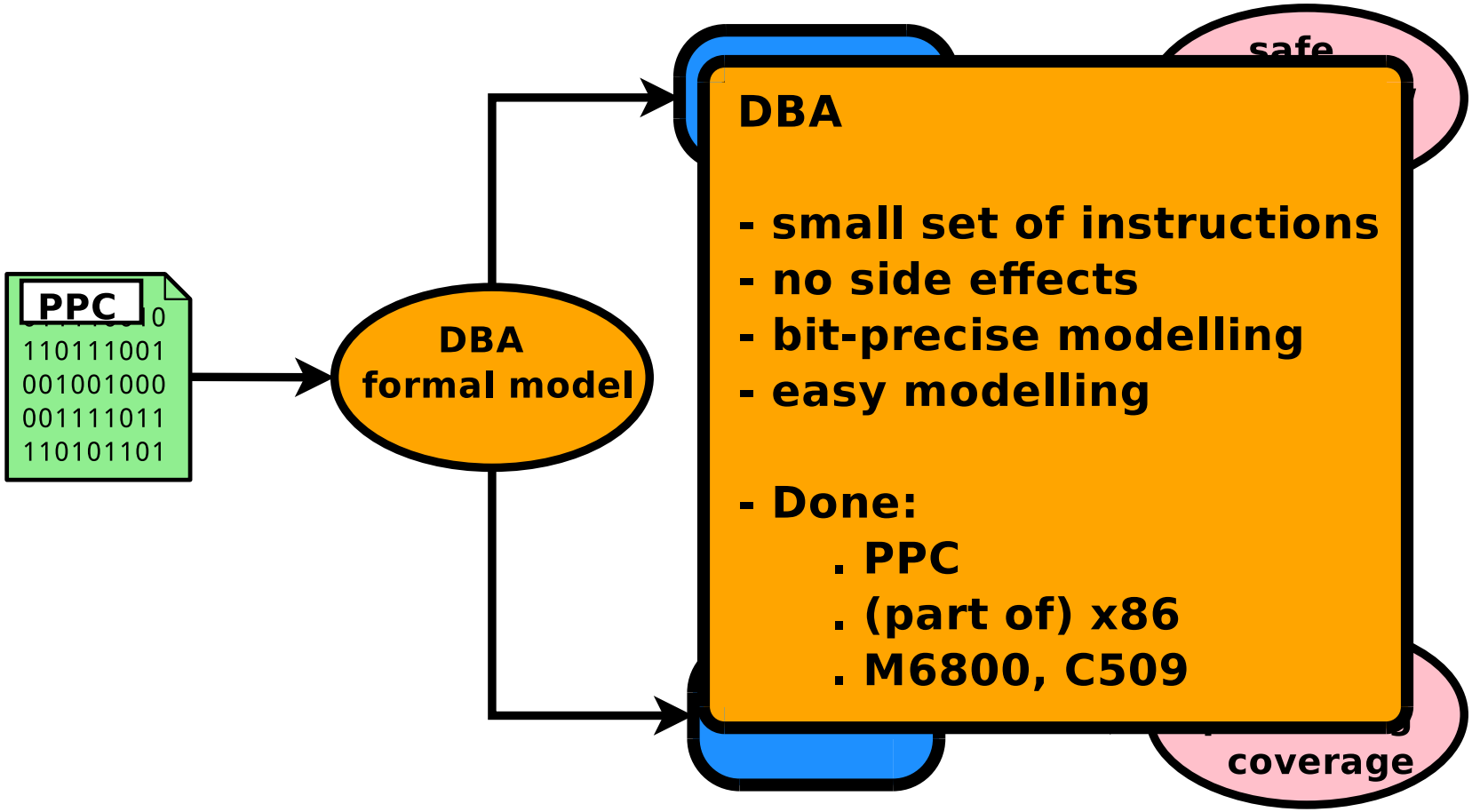
Outline

- Introduction
- The OSMOSE tool
- New features for DSE over safety-critical programs
- Case-studies
- Conclusion

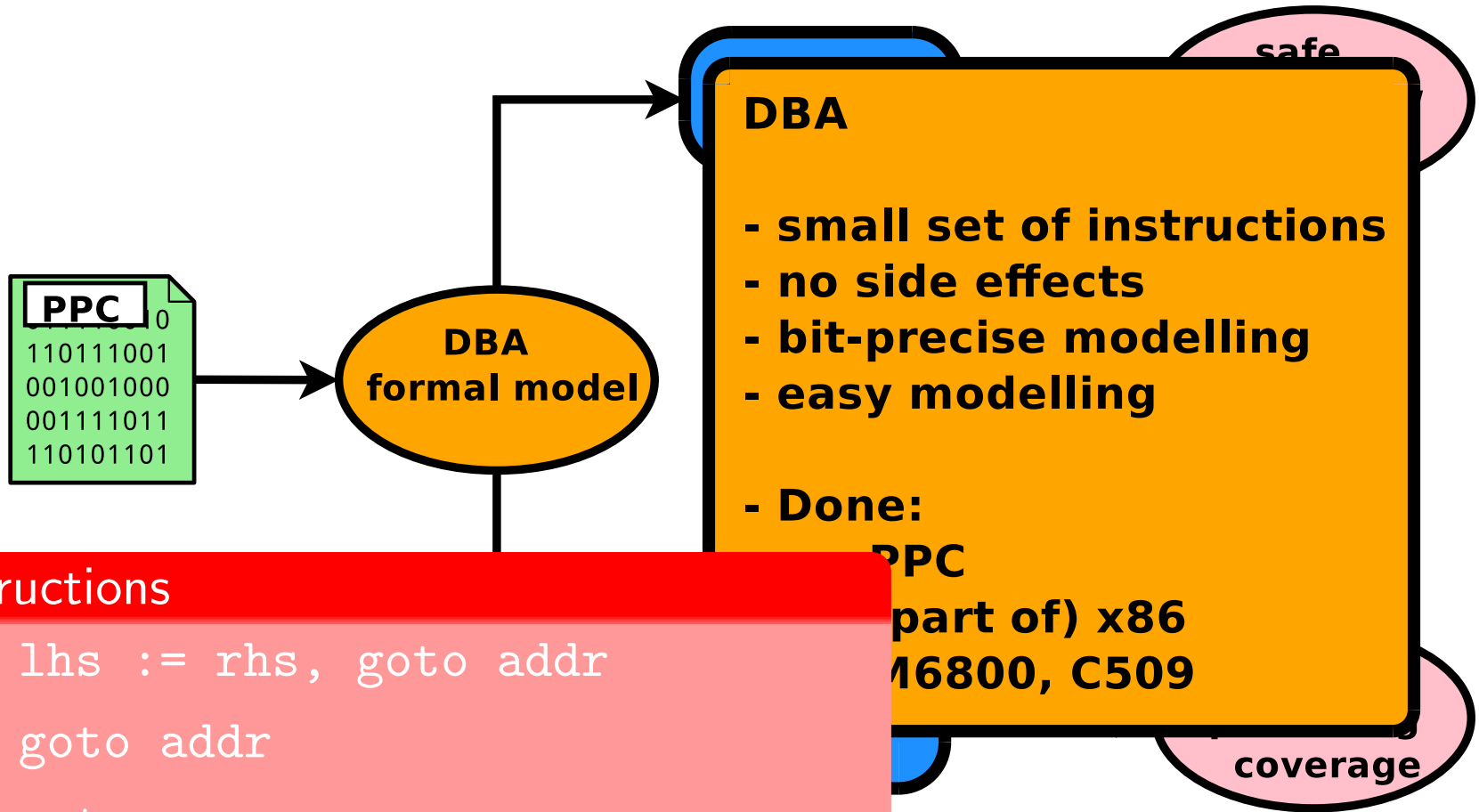
The OSMOSE tool



The OSMOSE tool



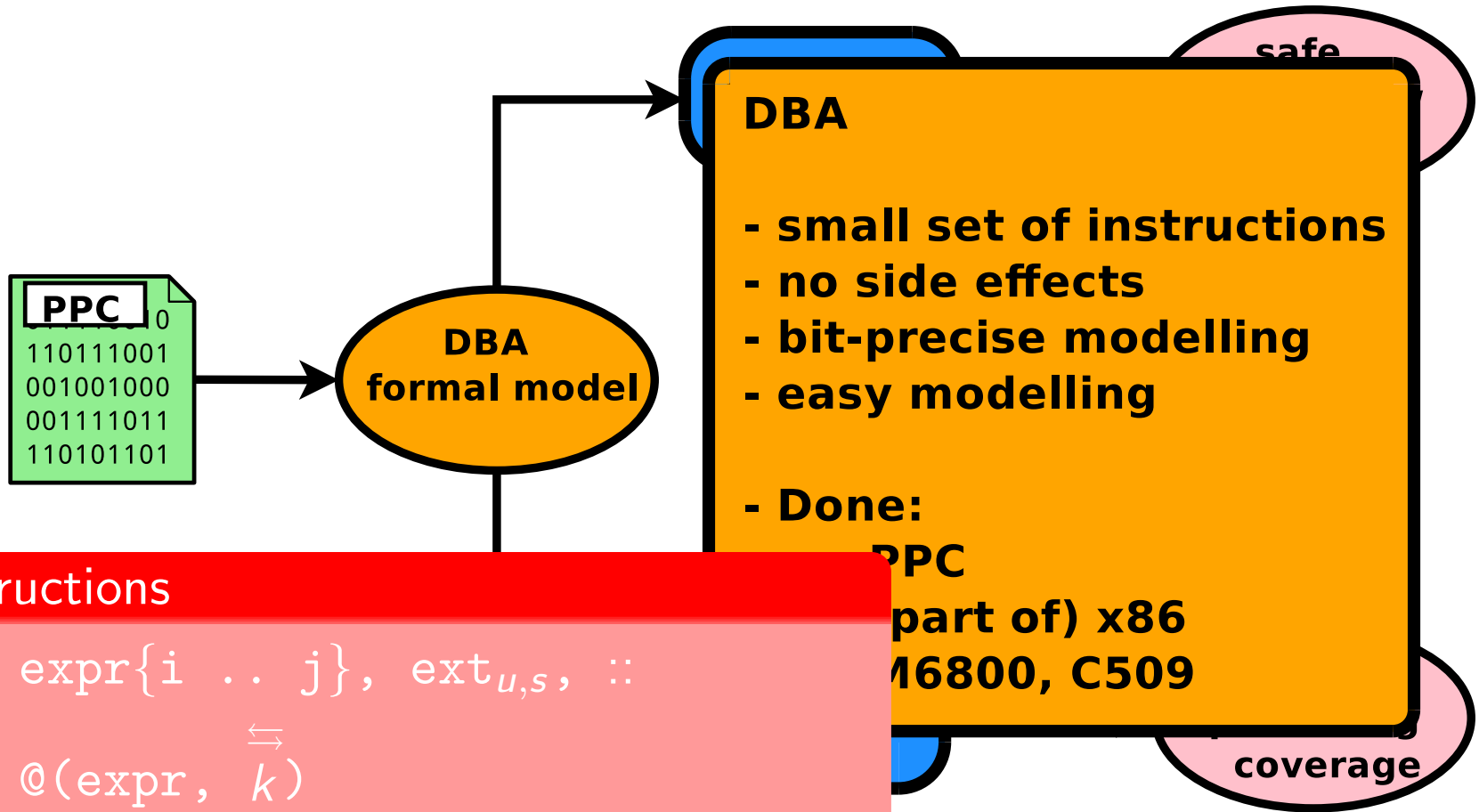
The OSMOSE tool



Instructions

- `lhs := rhs, goto addr`
- `goto addr`
- `goto expr`
- `ite(cond)?goto addr:goto addr'`

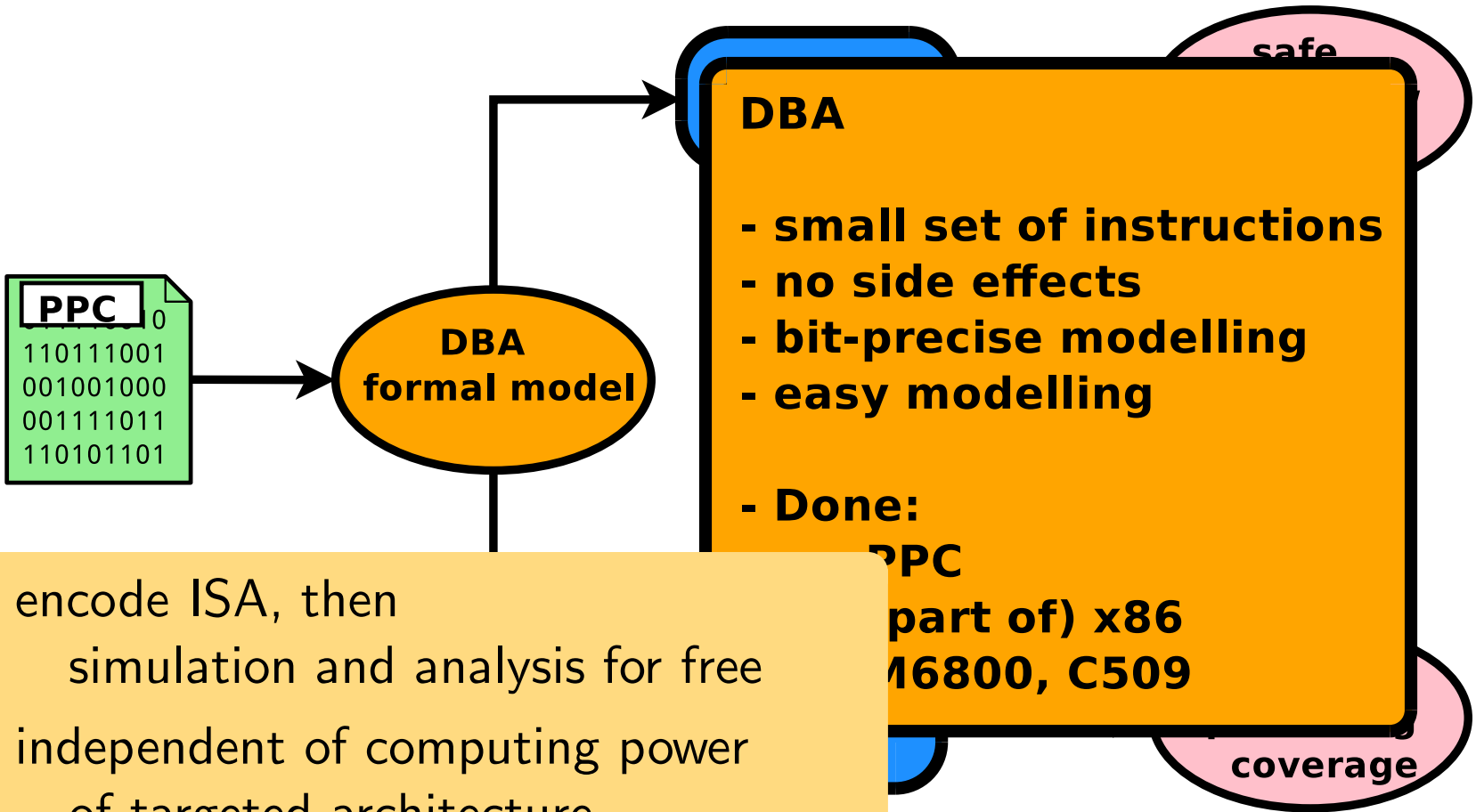
The OSMOSE tool



Instructions

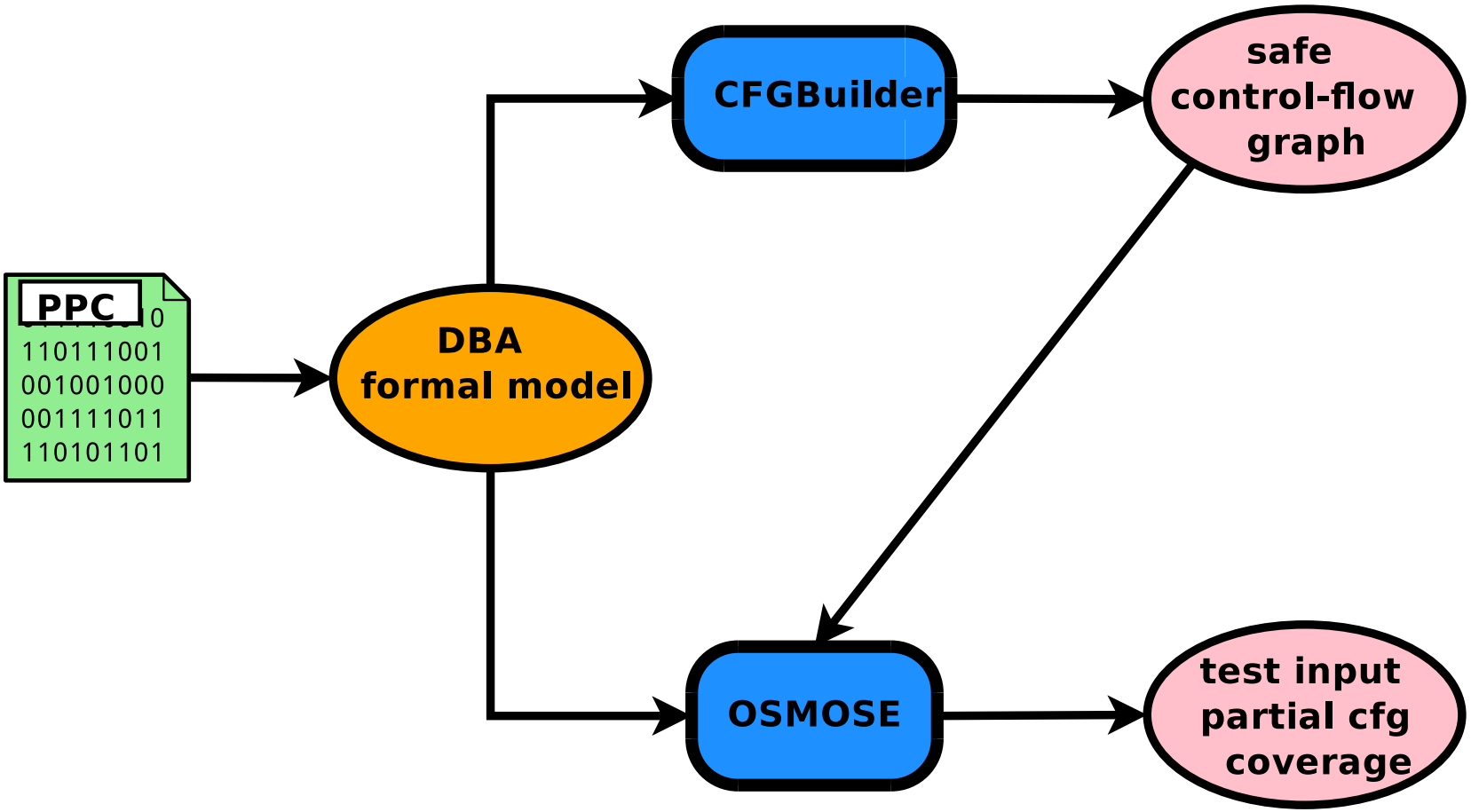
- $\texttt{expr\{i .. j\}}$, $\texttt{ext}_{u,s}$, $::$
- $\texttt{@(expr,}\ \overset{\leftrightarrows}{k}\texttt{)}$
- $+, -, \times, /_{u,s}, \%_{u,s}, =, \leq_{u,s}, \ldots$
- $!, \wedge, \vee, \oplus, <<, >>_{u,s}$

The OSMOSE tool

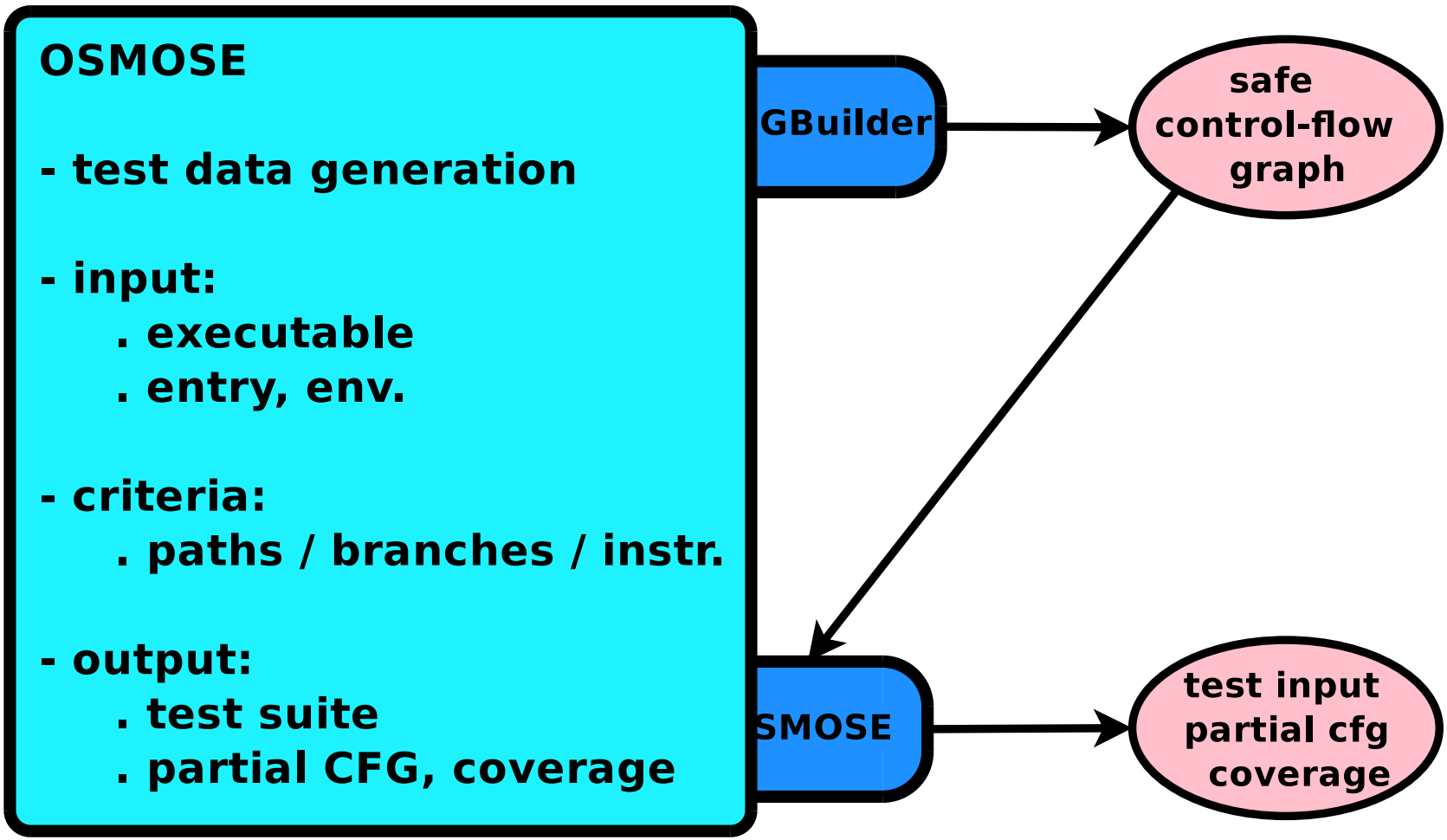


- encode ISA, then
  simulation and analysis for free
- independent of computing power
  of targeted architecture

The OSMOSE tool



The OSMOSE tool



The OSMOSE tool

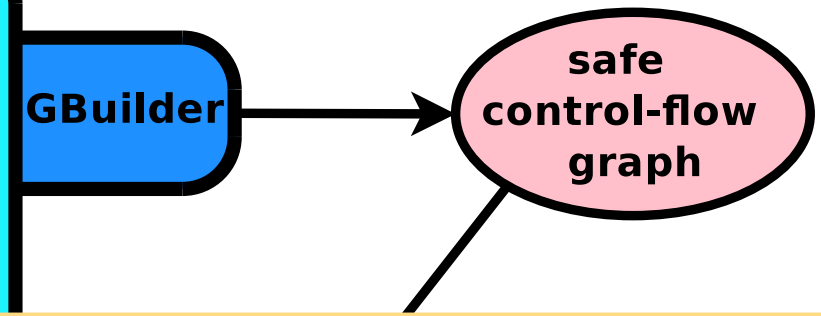
OSMOSE

- test data generation

- input:
. executable
. entry, env.

- criteria:
. paths / branches / inst

- output:
. test suite
. partial CFG, coverage



- dynamic symbolic execution [ICST-08,STVR-11]
- bit-precise constraint solving [TACAS-10]
- symbolic reasoning to discover new dynamic targets [STVR-11]
- path pruning optimisations [ICST-09]
- solver-independent optimizations (preprocessing, solution reuse, etc.)

Limits of our approach

Constraints

- memory model or strings : nothing fancy, but sufficient for critical programs
- floats : only programs without tricky reasoning on floats [real issue] [orthogonal challenge]

Low-level synchronization mechanisms

- interrupts, multi-threading, time-based synchronization
- left to the validation expert (methodology)
- match current methodologies at SAGEM and EDF

Outline

- Introduction
- The OSMOSE tool
- New features for DSE over safety-critical programs
- Case-studies
- Conclusion

New features

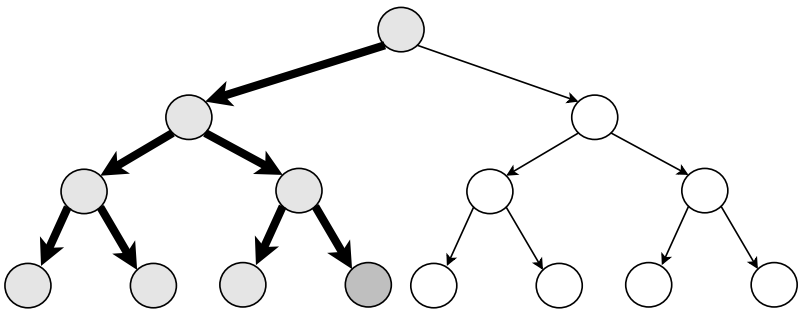
- generic search engine
- search directives
- test suite replay and completion
- output of concrete and symbolic states
- specification of dynamic targets
- goal-oriented testing

Remember our goals

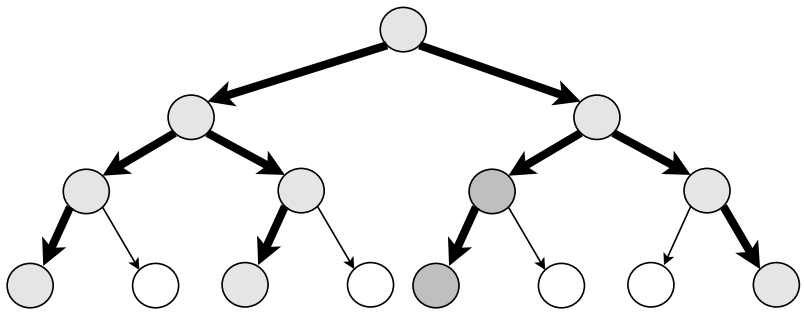
- very high coverage
- reliable results
- flexibility, allows guidance from user

Generic search engine

DFS has a low "coverage speed"



(a) DFS



(b) BFS

Many heuristics have been defined in the literature, but no best one

Generic search engine (2)

Idea = Generic search engine for easy integration of new searches

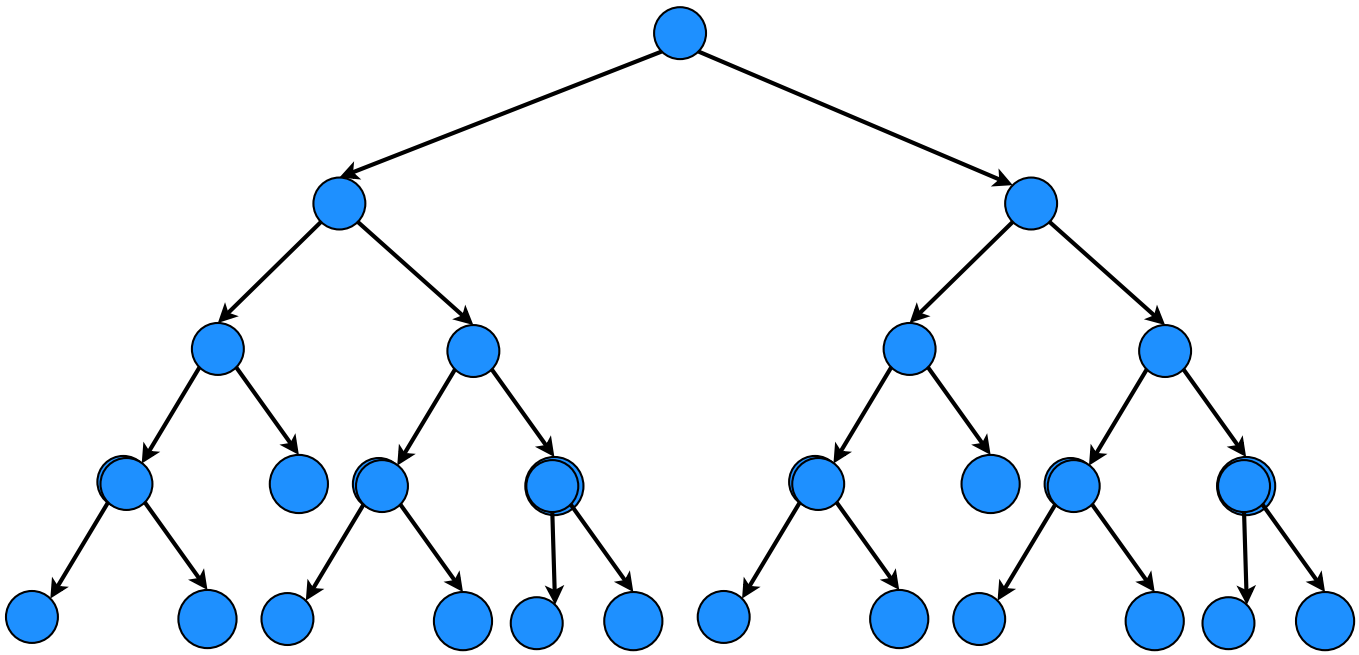
Our search engine requires

- an abstract data type `SCORE`
- function $\texttt{score} : \texttt{path} \mapsto \texttt{SCORE}$
- function $\texttt{cmp} : \texttt{SCORE} \times \texttt{SCORE} \mapsto \{<, =, >\}$

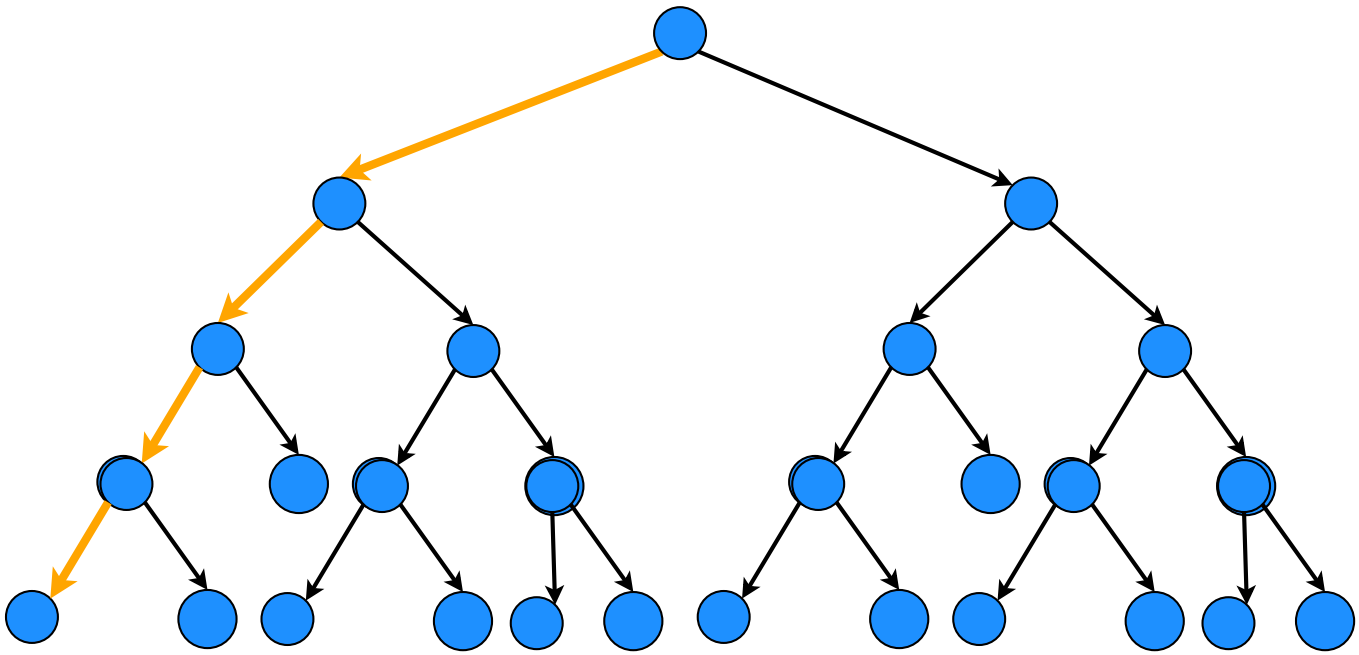
Algorithm

- rank all active paths (active $\approx$ uncovered)
- choose one among the best
- solve its path predicate, add the resulting new paths

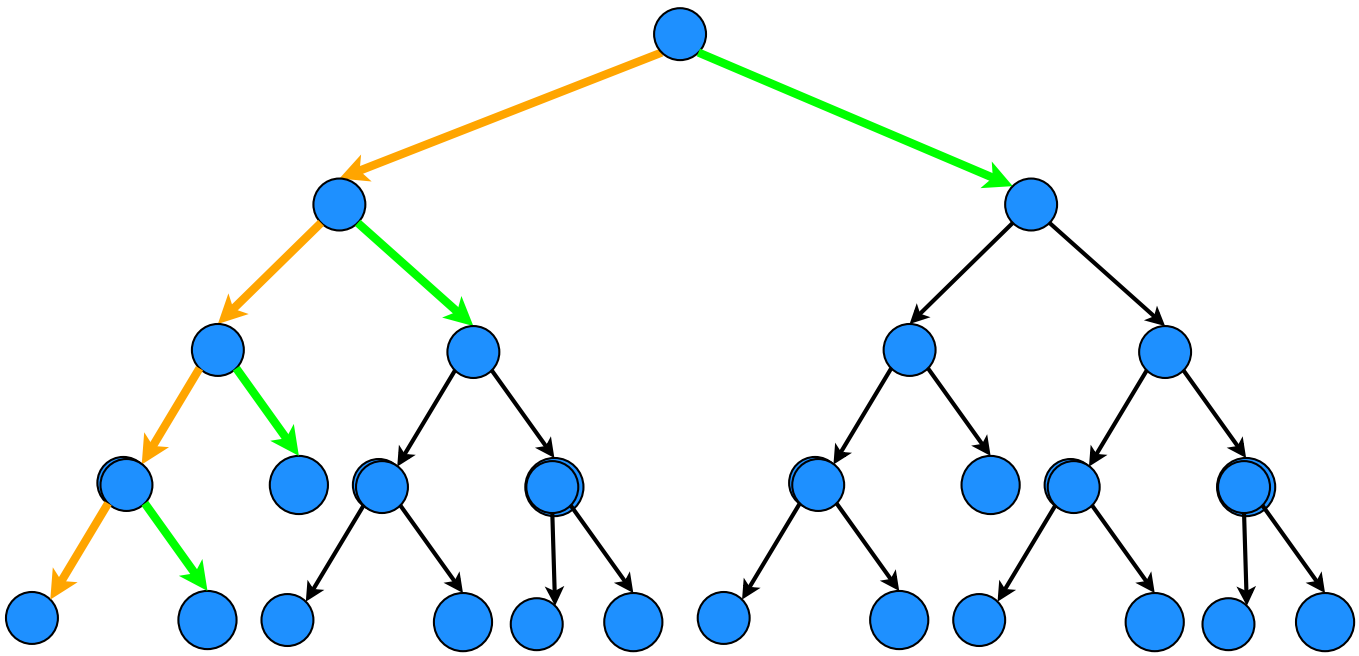
Generic search engine (3)



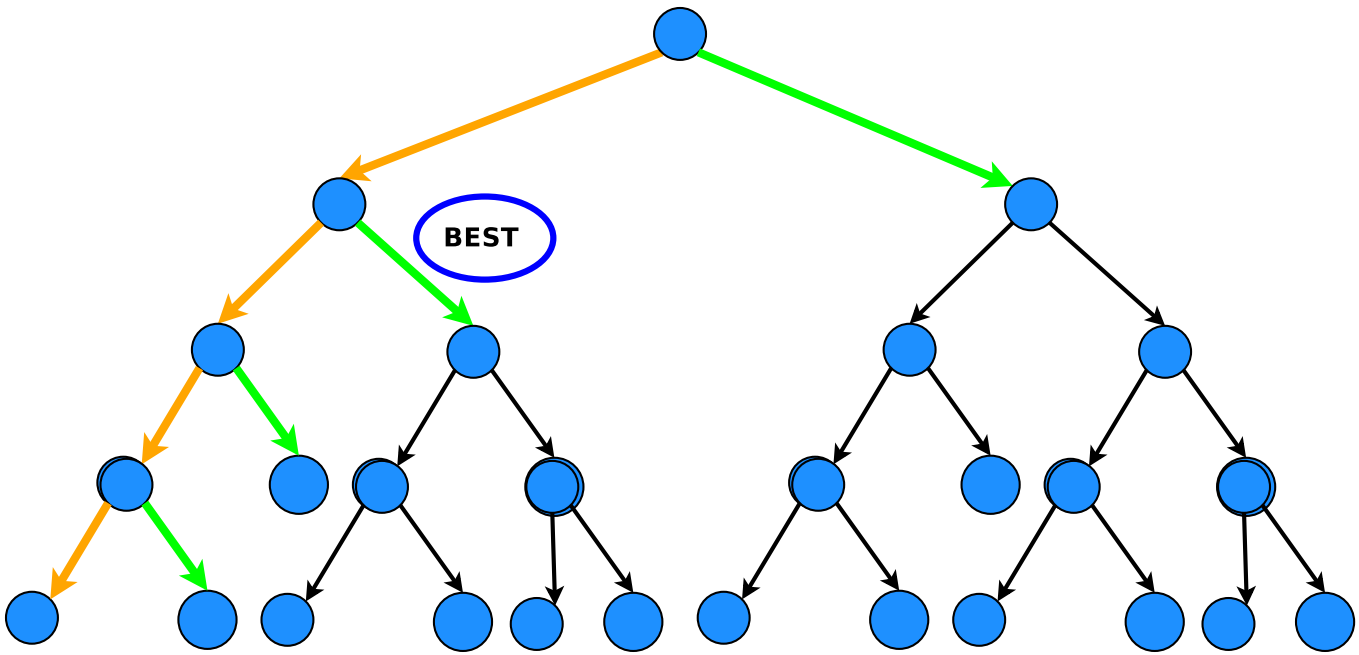
Generic search engine (3)



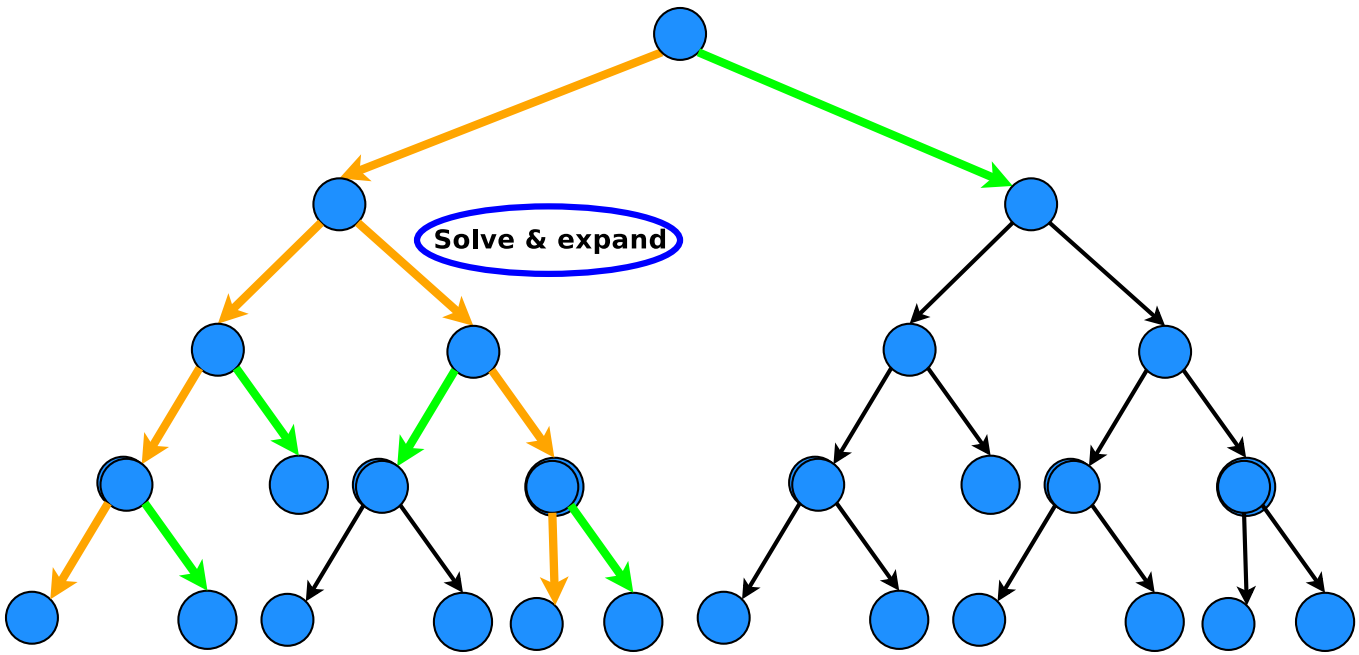
Generic search engine (3)



Generic search engine (3)



Generic search engine (3)



Generic search engine (4)

Easy to encode many existing heuristics

- DFS : `score` based on length, `cmp` on *max*
- BFS : `score` based on length, `cmp` on *min*
- random prefix : `score` is random, `cmp` is *min (arbitrary)*
- generational search [Godefroid *et al*, 2008] : `score` is $(generation, gain)$, `cmp` is $max_{generation} \circ max_{gain}$

In OSMOSE

- generic search engine :
  - DFS, BFS, random path
  - minCall-DFS, minCall-BFS
- a dedicated DFS-based DSE engine [more memory efficient]
- random data generation

Other features (1)

Directives restricting the search space

- `unsat_br (addr, bool)`
- `repeat addr1 at most N ( with reset on addr2)`
- `maxTryBranch (addr, bool) N`

Test replay and completion

- validation : replay test suite in external simulator
- incremental testing : complete existing test suites, smooth integration with existing test process
- combination of search heuristics

Other features (2)

Export (and reuse) of symbolic states

- useful for modular reasoning (typically : initialization)
- beware : may lost completeness or correctness (no silver-bullet)

Specification of dynamic targets

- by a human or a static analyser
- the coverage measure reported by OSMOSE is sound w.r.t. the specification
- OSMOSE checks the specification along the DSE process, but no completeness

Summary

features	High coverage	Trust	Flexibility
generic search engine (+ goal-oriented testing)	✓		
search directives	✓		✓
test suite replay and completion	✓	✓	✓
output of conc/symb. states	✓		✓
specification of dynamic targets		✓	

Outline

- Introduction
- The OSMOSE tool
- New features for DSE over safety-critical programs
- Case-studies
- Conclusion

Experiments

- automatic unit testing of medium-sized aircraft application
- full testing of a small (but tricky) aircraft application
- testing and comprehension of a third-party program
- experimental comparison of source vs binary coverage criteria

First case-study

Medium-size aircraft program (Sagem)

- 30,000 instructions, 250 functions
- max calldepth $= 10$

Goal : unit testing, no expert guidance

Results

- good coverage results for procedures with low height in the call graph (even with 2,000 instructions)
- tested on 40 functions with call-depth ≤ 4 :
  - full cover for 31 functions (in less than a few minutes)
  - bad cover ($< 50\%$) for only 5 functions
- robustness issue with higher-level procedures

Second case-study

Small program (17 procedures and 2,600 instructions), SAGEM

Goal = full testing from the program entry point

Program recognized hard to cover by testing teams

- random testing or DFS-DSE stuck to 50% coverage
- many infeasible paths
- huge search space :
  - one loop must be unrolled $\geq$ 380 times
  - artifical paths due to read-loops on volatile memory

Second case-study

Small program (17 procedures and 2,600 instructions), SAGEM

Goal = full testing from the program entry point

Program recognized hard to cover by testing teams

Approach

- search directives (main loop, read-loops)
- combination of MinCall-BFS and MinCall-DFS

Results

- 100% coverage of 15/17 procedures
- 50% coverage of 2 "library" procedures
- several uncovered branches have been shown to be uncoverable (in progress)

Third case-study

Toy control-command program written in assembly language (EdF)

- 3,000 instructions, 10 modules and 10 “library functions”
- Third-party software, sparse documentation

Complex to analyse : many unsat branches, long init

A modular approach

- analyse library functions in isolation to detect likely-unsat branches or other issues (ex : volatile memory)
- insert `unsat_br` directives
- modular analysis through export of the symbolic state obtained after initialization

Third case-study

Toy control-command program written in assembly language (EdF)

- 3,000 instructions, 10 modules and 10 “library functions”
- Third-party software, sparse documentation

Complex to analyse : many unsat branches, long init

Results

- achieve high coverage in 2 min only (otherwise : 35 min)
- help to understand the code (unfeasible branches, volatile memory, entries, etc.)
- help to pinpoint problems in doc (ack. by vendor)

Outline

- Introduction
- The OSMOSE tool
- New features for DSE over safety-critical programs
- Case-studies
- Conclusion

Conclusion

Binary-level testing of safety-critical programs

- important issue
- DSE is an interesting tool

Our contribution

- original and practically-relevant features for DSE over safety-critical programs
- experience report on several case-studies

Current challenges

- improve scaling w.r.t. call depth
- floats
- low-level synchronization (can handle through methodology)
- automatic sound CFG recovery

Experiments (2)

name	I	Br	Osmose cover	Osmose time	Osmose #tests	random cover	random time
aircraft0	237	36	100%	10	19	40%	20
aircraft1	290	140	98%	60	43	64%	100
aircraft2	201	72	100%	10	37	35%	20
aircraft3	977	190	50%	60	3	96%	60
aircraft4	2347	500	87%	600	15	68%	600
aircraft5	121 4103	2 509	100%	1	2	100%	10
aircraft6	250 425	18 34	94%	100	9	83%	120
aircraft7	506 15640	20 2790	80%	20	4	75%	500
aircraft8	957 30969	14 4952	14%	10	3	50%	500
aircraft9	627 31793	74 5034	77%	600	12	63%	600

Time in sec. Random tests : 1000 tests - unit testing