Specification of Concretization and Symbolization Policies in Symbolic Execution

Sébastien Bardin
joint work with
Robin David, Josselin Feist, Laurent Mounier, Marie-Laure Potet, Thanh Dihn Ta, Jean-Yves Marion

CEA LIST (Paris-Saclay, France)

ISSTA 2016
Dynamic Symbolic Execution (DSE) : powerful approach to verif. and testing

- three key ingredients: path predicate computation & solving, path search, concretization & symbolization policy (C/S)

C/S is an essential part, yet mostly not studied

- many policies (one per tool), no systematic study of C/S
- undocumented, unclear
- tools: often a single hardcoded policy, no reuse across tools

Our goal: establish C/S as a proper field of study [focus first on specification]

- CSML, a specification language for C/S ✓
  - clear, non-ambiguous [documentation]
  - tool independent [reuse, sharing, tuning]
  - executable [input for tools]

- implemented in BINSEC ✓

- an experimental comparison of C/S policies ✓
Preamble

About formal verification

- Between Software Engineering and Theoretical Computer Science
- Goal = proves correctness in a mathematical way

Key concepts: \( M \models \varphi \)

- \( M \): semantic of the program
- \( \varphi \): property to be checked
- \( \models \): algorithmic check

Kind of properties

- absence of runtime error
- pre/post-conditions
- temporal properties
Industrial reality in some key areas, especially safety-critical domains

- hardware, aeronautics [airbus], railroad [metro 14], smartcards, drivers [Windows], certified compilers [CompCert] and OS [Sel4], etc.

---

Ex: Airbus

Verification of

- runtime errors [Astrée]
- functional correctness [Frama-C]
- numerical precision [Fluctuat]
- source-binary conformance [CompCert]
- ressource usage [Absint]
Preamble

Next big challenge

- Apply formal methods to less-critical software
- Very different context: no formal spec, less developer involvement, etc.

Difficulties

- Robustness [w.r.t. software constructs]
- No place for false alarms
- Scale
- Sometimes, not even source code
Apply formal methods to less-critical software

Very different context: no formal spec, less developer involvement, etc.

Difficulties
- robustness [w.r.t. software constructs]
- no place for false alarms
- scale
- sometimes, not even source code

DSE as a first step
- very robust
- (mostly) no false alarm
- scale in some ways
- ok for binary code
Dynamic Symbolic Execution [since 2004-2005: dart, cute, pathcrawler]

- a very powerful formal approach to verification and testing
- many tools and successful case-studies since mid 2000’s
  - SAGE, Klee, Mayhem, etc.
  - coverage-oriented testing, bug finding, exploit generation, reverse
- arguably one of the most wide-spread use of formal methods

Very good properties

- mostly no false alarm, robust, scale, ok for binary code
DSE in a nutshell

Introducing DSE

Dynamic Symbolic Execution [since 2004-2005: dart, cute, pathcrawler]

- A very powerful formal approach to verification and testing
- Many tools and successful case-studies since mid 2000's
  - SAGE, Klee, Mayhem, etc.
  - Coverage-oriented testing, bug finding, exploit generation, reverse
- Arguably one of the most widespread use of formal methods

Very good properties

- Mostly no false alarm, robust, scale, ok for binary code

Key idea: path predicate [King 70's]

- Consider a program $P$ on input $v$, and a given path $\sigma$
- A path predicate $\varphi_\sigma$ for $\sigma$ is a formula s.t.
  $v \models \varphi_\sigma \Rightarrow P(v)$ follows $\sigma$
- Intuitively the conjunction of all branching conditions
- Old idea, recent renewal interest [powerful solvers, dynamic+symbolic]
int main () {
    int x = input();
    int y = input();
    int z = 2 * y;
    if (z == x) {
        if (x > y + 10)
            failure;
    }
    success;
}

- given a path of the program
- automatically find input that follows the path
- then, iterate over all paths
int main () {
    int x = input();
    int y = input();
    int z = 2 * y;
    if (z == x) {
        if (x > y + 10)
            failure;
        else
            success;
    }
}

Three key ingredients

- path predicate computation & solving
- path search
- C/S policy

- given a path of the program
- automatically find input that follows the path
- then, iterate over all paths
DSE in a nutshell

Path predicate computation

**Usually easy to compute**  [forward, introduce new logical variables at each step]

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>input(y,z)</td>
</tr>
<tr>
<td>1</td>
<td>w := y+1</td>
</tr>
<tr>
<td>2</td>
<td>x := w + 3</td>
</tr>
<tr>
<td>3</td>
<td>if (x &lt; 2 * z)</td>
</tr>
<tr>
<td>4</td>
<td>if (x &lt; z)</td>
</tr>
</tbody>
</table>

Path predicate (input $Y_0$ et $Z_0$)
**DSE in a nutshell**

**Path predicate computation**

**Usually easy to compute**  
[forward, introduce new logical variables at each step]

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>input(y,z)</td>
</tr>
<tr>
<td>1</td>
<td>( w := y + 1 )</td>
</tr>
<tr>
<td>2</td>
<td>( x := w + 3 )</td>
</tr>
<tr>
<td>3</td>
<td>if ( x &lt; 2 \times z ) [True branch]</td>
</tr>
<tr>
<td>4</td>
<td>if ( x &lt; z ) [False branch]</td>
</tr>
</tbody>
</table>

**Path predicate (input \( Y_0 \) et \( Z_0 \))**

\[
\text{let } W_1 \triangleq Y_0 + 1 \text{ in }
\]
DSE in a nutshell
Path predicate computation

**Usually easy to compute** [forward, introduce new logical variables at each step]

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>input(y,z)</td>
</tr>
<tr>
<td>1</td>
<td>w := y+1</td>
</tr>
<tr>
<td>2</td>
<td>x := w + 3</td>
</tr>
<tr>
<td>3</td>
<td>if (x &lt; 2 * z) [True branch]</td>
</tr>
<tr>
<td>4</td>
<td>if (x &lt; z) [False branch]</td>
</tr>
</tbody>
</table>

Path predicate (input $Y_0$ et $Z_0$)
let $W_1 \triangleq Y_0 + 1$ in
let $X_2 \triangleq W_1 + 3$ in
Usually easy to compute
[forward, introduce new logical variables at each step]

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>input(y,z)</td>
</tr>
<tr>
<td>1</td>
<td>w := y+1</td>
</tr>
<tr>
<td>2</td>
<td>x := w + 3</td>
</tr>
<tr>
<td>3</td>
<td>if (x &lt; 2 * z) [True branch]</td>
</tr>
<tr>
<td>4</td>
<td>if (x &lt; z) [False branch]</td>
</tr>
</tbody>
</table>

Path predicate (input $Y_0$ et $Z_0$)
let $W_1 \triangleq Y_0 + 1$ in
let $X_2 \triangleq W_1 + 3$ in
$X_2 < 2 \times Z_0$
Path predicate computation

**Usually easy to compute**

[forward, introduce new logical variables at each step]

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>input(y,z)</td>
</tr>
<tr>
<td>1</td>
<td>w := y+1</td>
</tr>
<tr>
<td>2</td>
<td>x := w + 3</td>
</tr>
<tr>
<td>3</td>
<td>if (x &lt; 2 * z) [True branch]</td>
</tr>
<tr>
<td>4</td>
<td>if (x &lt; z) [False branch]</td>
</tr>
</tbody>
</table>

Path predicate (input $Y_0$ et $Z_0$)

let $W_1 \triangleq Y_0 + 1$ in
let $X_2 \triangleq W_1 + 3$ in
$X_2 < 2 \times Z_0 \land X_2 \geq Z_0$
input: a program $P$
output: a test suite $TS$ covering all feasible paths of $Paths^{\leq k}(P)$

- pick a path $\sigma \in Paths^{\leq k}(P)$
- compute a path predicate $\varphi_\sigma$ of $\sigma$
- solve $\varphi_\sigma$ for satisfiability
- SAT(s)? get a new pair $< s, \sigma >$
- loop until no more path to cover
**input**: a program $P$

**output**: a test suite $TS$ covering all feasible paths of $Paths^{\leq k}(P)$

- **pick a path** $\sigma \in Paths^{\leq k}(P)$
- **compute a path predicate** $\varphi_\sigma$ of $\sigma$
- **solve** $\varphi_\sigma$ for satisfiability
- **SAT(s)?** get a new pair $< s, \sigma >$
- loop until no more path to cover
**input**: a program $P$

**output**: a test suite $TS$ covering all feasible paths of $Paths^{\leq k}(P)$

- **pick a path** $\sigma \in Paths^{\leq k}(P)$
- **compute a path predicate** $\varphi_\sigma$ of $\sigma$
- **solve** $\varphi_\sigma$ for satisfiability
- **SAT(s)**? get a new pair $<s, \sigma>$
- **loop until no more path to cover**
input: a program $P$
output: a test suite $TS$ covering all feasible paths of $Paths^{\leq k}(P)$

- pick a path $\sigma \in Paths^{\leq k}(P)$
- compute a path predicate $\varphi_\sigma$ of $\sigma$
- solve $\varphi_\sigma$ for satisfiability
- SAT(s)? get a new pair $< s, \sigma >$
- loop until no more path to cover
**input**: a program $P$

**output**: a test suite $TS$ covering all feasible paths of $Paths^{\leq k}(P)$

- pick a path $\sigma \in Paths^{\leq k}(P)$
- compute a *path predicate* $\varphi_\sigma$ of $\sigma$
- solve $\varphi_\sigma$ for satisfiability
- SAT(s)? get a new pair $< s, \sigma >$
- loop until no more path to cover
input: a program $P$
output: a test suite $TS$ covering all feasible paths of $Paths^{\leq k}(P)$

- pick a path $\sigma \in Paths^{\leq k}(P)$
- compute a path predicate $\varphi_\sigma$ of $\sigma$
- solve $\varphi_\sigma$ for satisfiability
- $\text{SAT}(s)\ ?$ get a new pair $< s, \sigma >$
- loop until no more path to cover
DSE in a nutshell

Path Exploration

**input**: a program $P$

**output**: a test suite $TS$ covering all feasible paths of $Paths_{\leq k}(P)$

- pick a path $\sigma \in Paths_{\leq k}(P)$
- compute a *path predicate* $\varphi_\sigma$ of $\sigma$
- solve $\varphi_\sigma$ for satisfiability
- SAT(s)? get a new pair $< s, \sigma >$
- loop until no more path to cover
**DSE in a nutshell**

**Path Exploration**

**input** : a program $P$

**output** : a test suite $TS$ covering all feasible paths of $Paths^{\leq k}(P)$

- **pick a path** $\sigma \in Paths^{\leq k}(P)$
- **compute a path predicate** $\varphi_\sigma$ of $\sigma$
- **solve** $\varphi_\sigma$ for satisfiability
- **SAT(s)**? get a new pair $<s, \sigma>$
- **loop until no more path to cover**

**Beware**

- $\times$ #paths!
- $\times$ incomplete
Robustness: what if the instruction cannot be reasoned about?

- missing code, self-modification
- hash functions, dynamic memory accesses, NLA operators

Solutions

- Concretization: replace by runtime value [lose completeness]
- Symbolization: replace by fresh variable [lose correctness]
Robustness: what if the instruction cannot be reasoned about?

- missing code, self-modification
- hash functions, dynamic memory accesses, NLA operators

C/S essential to DSE

- robustness to real-life code
- trade-off correction / completeness / efficiency

Solutions

- **Concretization**: replace by runtime value [lose completeness]
- **Symbolization**: replace by fresh variable [lose correctness]
The problem

Outline

- about DSE
- the problem with C/S
- goal and results
- experiments
- conclusion
The problem with C/S policies

State of DSE

- Path predicate computation + solving ✓
- Path search: under active research
- C/S: ??? kind of black magic

- hardcoded
- often a single C/S
- no easy tuning
- no reuse across tools

- undocumented, unclear
- many policies (one per tool)
- no comparison of C/S
- no systematic study of C/S
Consider the following situation

- instruction \( x := @ (a \times b) \)
- your tool documentation says: "memory accesses are concretized"
- suppose that at runtime: \( a = 7, b = 3 \)

What is the intended meaning? [perfect reasoning: \( x \equiv \text{select}(M, a \times b) \)]

- **CS1**: \( x \equiv \text{select}(M, 21) \) [incorrect]
- **CS2**: \( x \equiv \text{select}(M, 21) \land a \times b \equiv 21 \) [minimal]
- **CS3**: \( x \equiv \text{select}(M, 21) \land a \equiv 7 \land b \equiv 3 \) [atomic]

No best choice, depends on the context

- acceptable loss of correctness / completeness?
- is it mandatory to get rid off \( \times \)?
The problem
Too many C/S policies

Just for C/S on memory accesses

- 4 basic policies: concretize or keep symbolic reads / writes
- Exotic variations: multi-level dereferencement [exe], domain restriction [osmose], taint-based [s. heelan], dataflow-based [mayhem], etc.
- Flavors of concretization: minimal, atomic, incorrect
- All can be combined together
Our goal

Establish C/S as a proper field of study

- what is a generic C/S?
- how DSE can handle generic C/S?
- identify tradeoffs, sweetspots, etc.

First step: a specification mechanism for C/S

- clear, non-ambiguous
- tool independent
- executable
Establish C/S as a proper field of study

- what is a generic C/S?
- how DSE can handle generic C/S?
- identify tradeoffs, sweetspots, etc.

First step: a specification mechanism for C/S

- clear, non-ambiguous
- tool independent
- executable
- input for tools

Results

- formal definition of a generic C/S ✓
- a variant of DSE supporting generic C/S ✓
- CSML, a specification language for C/S ✓
- implementation in BINSEC ✓
- an experimental comparison of C/S policies ✓
Our goal

Overview

DSE

Generic C/S

C/S policy

CSML input

path

formula

csml spec

csml spec

csml spec
Our goal

Overview

Tool users

- clear, well-doc. C/S
- change, reuse, share
- best C/S available

Tool builders

- flexibility
- do not reimplement existing C/S
- futur-proof wrt C/S

DSE

Generic C/S

C/S policy

CSML input

path

formula

csml spec

csml spec

csml spec
Technical keys

What is a C/S policy?

A decision function queried

- within path predicate computation
- before logical evaluation of an expression
- in the scope of a given location, instruction and memory state

\[ \text{cs} : \text{loc} \times \text{instr} \times \text{state} \times \text{expr} \mapsto \left\{ C \quad \text{concretization} \\
S \quad \text{symbolization} \\
P \quad \text{propagation} \right\} \]
Technical keys

DSE with parametric C/S

Example:

- `loc: x := a + b`
- Concrete memory state: `{a ↦ 3; b ↦ 5}`
- Symbolic memory state: `{a ↦ a; b ↦ b^9}`

Standard evaluation, no C/S: $\llbracket a + b \rrbracket \mapsto a^2 + b^9$

Evaluation with propagation: $\llbracket a + b \rrbracket_{cs=P} \mapsto (a^2 + b^9, \top)$

Evaluation with symbolization: $\llbracket a + b \rrbracket_{cs=S} \mapsto (\text{fresh}, \top)$

Evaluation with concretization: $\llbracket a + b \rrbracket_{cs=C} \mapsto (8, a^2 + b^9 = 8)$
Rule-based language \( guard \Rightarrow \{C, S, P\} \)

Guard of the form \( \pi_{loc} :: \pi_{ins} :: \pi_{expr} :: \pi_{\Sigma} \)

- predicates on the location, instruction, expression, concrete memory state
- \( \pi_{ins} \) and \( \pi_{expr} \) mostly based on pattern matching and subterm checking
- predicates checked sequentially
- limited communication: meta-variables \((?x, ?\star)\) and placeholders \((!x, !\Box)\)

Set of rules

- checked sequentially, the first fireable rule returns
- presence of a default rule
Technical keys

Example of specifications (1)

\[ \pi_{loc} :: \pi_{ins} :: \pi_{expr} :: \pi_{\Sigma} \Rightarrow \{C, S, P\} \]

Meaning

- concretize result of a read value
- or: "if we are evaluating an expression e built with @, then e is concretized, otherwise it is propagated."

Examples

- \( x := a + @b \): @b is concretized
Technical keys

Example of specifications (2)

\[ \pi_{loc} :: \pi_{ins} :: \pi_{expr} :: \pi_{\Sigma} \Rightarrow \{C, S, P\} \]

\[
\begin{array}{c c c c c c c}
* & : : & \langle @?e := ?\star \rangle & : : & \langle !e \rangle & : : & * & \Rightarrow C; \\
\text{default} & & & & & & & \Rightarrow P;
\end{array}
\]

Meaning

- concretize write addresses
- or: “if we are evaluating an expression \( e \) in the context of an assignment where \( e \) is used as the write address, then \( e \) is concretized, otherwise it is propagated.”

Examples

- \( x := a + @b \): nothing is concretized
- \( @x := a + @b \): \( x \) is concretized
Consider instruction \( x := @a \times b \), suppose at runtime: \( a = 7 \), \( b = 3 \)

- **Minimal concretization of r/w expressions [CS2]**
  \[
  * :: \langle ?i \rangle :: (@ !\Box) \prec !i :: * \Rightarrow C
  \]

- **Recursive concretization of r/w expressions**: [concretize \( a \times b \), \( a \), \( b \)]
  \[
  * :: \langle ?i \rangle :: !\Box \prec (@ ?\star) \prec !i :: * \Rightarrow C
  \]

- **Atomic concretization of r/w expressions [CS3]** [concretize \( a \), \( b \)]
  \[
  * :: \langle ?i \rangle :: \text{var}(!\Box) \land !\Box \prec (@ ?\star) \prec !i :: * \Rightarrow C
  \]

- **Incorrect concretization of r/w expressions [CS1]** [replace \( a \times b \) by 21]
  \[
  * :: \langle ?i \rangle :: (@ !\Box) \prec !i :: * \Rightarrow S_{eval\Sigma(!\Box)}
  \]
Technical keys

CSML good properties

Well-defined

- any CSML spec defines a C/S policy
- only $C$ and $P$ : keeps correctness
- only $S$ and $P$ : keeps completeness

Expressive enough

- sufficient for all examples from literature [systematic review]
- yet, still limited [say something about current C/S ?]

Implementable : see after
Technical keys

CSML good properties

Well-defined

- any CSML spec defines a C/S policy
- only $C$ and $P$: keeps correctness
- only $S$ and $P$: keeps completeness

Expressive enough

- sufficient for all examples from literature [systematic review]
- yet, still limited [say something about current C/S ?]

Implementable: see after

About the language itself

- we describe the inner engine, not the user view
- syntax can be improved
- complexity can be hidden (predefined options, patterns)
Experiments

Implementation and experiments

CSML implemented in BINSEC/SE [binary-level dse tool]
- first DSE tool with generic C/S support

Experiment 1: evaluate CSML overhead
- vs: no C/S, C/S encoded via callbacks
- result: CSML does yield a cost, yet negligible wrt. solving time

Experiment 2: experimental comparison of C/S policies
- five C/S policies for memory accesses: CC, CP, PC, PP*, PP
- result: PP* better on average, yet no clear winner: need different C/S!
- first time such a C/S comparison is performed!
Experiments

CSML Overhead

Bench

- 167 programs (100 coreutils, 17 malware, 50 nist samate/verisec)
- ≈ 45,000 queries

<table>
<thead>
<tr>
<th></th>
<th>min</th>
<th>max</th>
<th>average</th>
</tr>
</thead>
<tbody>
<tr>
<td>base (PP)</td>
<td>0.04%</td>
<td>3%</td>
<td>0.3%</td>
</tr>
<tr>
<td>rule-based</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C/S policy</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CC</td>
<td>0.1%</td>
<td>17%</td>
<td>1.2%</td>
</tr>
<tr>
<td>CP</td>
<td>0.1%</td>
<td>23.5%</td>
<td>1.45%</td>
</tr>
<tr>
<td>PC</td>
<td>0.08%</td>
<td>12.8%</td>
<td>0.85%</td>
</tr>
<tr>
<td>PP*</td>
<td>0.08%</td>
<td>12.3%</td>
<td>0.95%</td>
</tr>
<tr>
<td>PP</td>
<td>0.05%</td>
<td>4%</td>
<td>0.48%</td>
</tr>
<tr>
<td>hard-coded</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C/S policy</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CC</td>
<td>0.05%</td>
<td>8.5%</td>
<td>0.5%</td>
</tr>
<tr>
<td>CP</td>
<td>0.05%</td>
<td>8.2%</td>
<td>0.5%</td>
</tr>
<tr>
<td>PC</td>
<td>0.05%</td>
<td>8%</td>
<td>0.45%</td>
</tr>
<tr>
<td>PP*</td>
<td>0.05%</td>
<td>6%</td>
<td>0.45%</td>
</tr>
<tr>
<td>PP</td>
<td>0.04%</td>
<td>3%</td>
<td>0.3%</td>
</tr>
</tbody>
</table>

Reported figures

- ratio between cost of formula creation and creation + solving
- note: solving time does not depend on the way C/S is implemented

Bardin et al.

ISSTA 2016
Experiments

Quantitative comparison

Five policies for memory accesses

- CC, PC, CP, PP*, PP
- first letter $\rightarrow$ read operation, second letter $\rightarrow$ write operation

<table>
<thead>
<tr>
<th></th>
<th>samate opt</th>
<th>samate best</th>
<th>core opt</th>
<th>core best</th>
<th>malware opt</th>
<th>malware best</th>
<th>total opt</th>
<th>total best</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC</td>
<td>20</td>
<td>0</td>
<td>44</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>69</td>
<td>1</td>
</tr>
<tr>
<td>PC</td>
<td>20</td>
<td>2</td>
<td>49</td>
<td>4</td>
<td>6</td>
<td>1</td>
<td>75</td>
<td>7</td>
</tr>
<tr>
<td>CP</td>
<td>23</td>
<td>1</td>
<td>61</td>
<td>11</td>
<td>4</td>
<td>0</td>
<td>88</td>
<td>12</td>
</tr>
<tr>
<td>PP*</td>
<td>36</td>
<td>12</td>
<td>71</td>
<td>24</td>
<td>10</td>
<td>5</td>
<td>117</td>
<td>41</td>
</tr>
<tr>
<td>PP</td>
<td>33</td>
<td>9</td>
<td>36</td>
<td>7</td>
<td>7</td>
<td>2</td>
<td>76</td>
<td>18</td>
</tr>
</tbody>
</table>

best (resp. opt) : number of programs for which the considered policy returns the strictly highest (resp. highest) number of SAT answers
Dynamic Symbolic Execution (DSE) : powerful approach to verif. and testing

- three key ingredients: path predicate computation & solving, path search, concretization & symbolization policy (C/S)

C/S is an essential part, yet mostly not studied

- many policies (one per tool), no systematic study of C/S
- undocumented, unclear
- tools: often a single hardcoded policy, no reuse across tools

Our goal: establish C/S as a proper field of study [focus first on specification]

- CSML, a specification language for C/S ✓
  - clear, non-ambiguous
  - tool independent
  - executable

- implemented in BINSEC ✓

- an experimental comparison of C/S policies ✓
Dynamic Symbolic Execution [Korel+, Williams+, Godefroid+]

- Interleave dynamic and symbolic executions
- Drive the search towards feasible paths for free
- Give hints for relevant under-approximations [robustness]

Concretization: force a symbolic variable to take its runtime value

- Application 1: follow only feasible path for free
- Application 2: correct approximation of “difficult” constructs [out of scope or too expensive to handle]
About robustness

Goal = find input leading to ERROR
(assume we have only a solver for linear integer arith.)

f(int x, int y) {z=x*x; if (y == z) ERROR; else OK }

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>input(x,y)</td>
</tr>
<tr>
<td>1</td>
<td>z := x * x</td>
</tr>
<tr>
<td>2</td>
<td>if (z == y) [True branch]</td>
</tr>
</tbody>
</table>

Path predicate (input \( X_0 \) et \( Y_0 \)) — Unrealistic perfect symbolic reasoning
Goal = find input leading to ERROR
(assume we have only a solver for linear integer arith.)

\[ f(\text{int } x, \text{int } y) \{ z= x^2; \text{ if } (y == z) \text{ ERROR; else OK } \} \]

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>input(x,y)</td>
</tr>
<tr>
<td>1</td>
<td>( z := x \times x )</td>
</tr>
<tr>
<td>2</td>
<td>if (z == y) [True branch]</td>
</tr>
</tbody>
</table>

Path predicate (input \( X_0 \) et \( Y_0 \)) — Unrealistic perfect symbolic reasoning
\[ \top \land Z_1 = X_0 \times X_0 \]
Goal = find input leading to ERROR
(assume we have only a solver for linear integer arith.)

\[ f(\text{int } x, \text{int } y) \{ z=x*x; \text{ if } (y == z) \text{ ERROR; else OK} \} \]

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>input(x,y)</td>
</tr>
<tr>
<td>1</td>
<td>( z := x \times x )</td>
</tr>
<tr>
<td>2</td>
<td>if ( (z == y) ) [True branch]</td>
</tr>
</tbody>
</table>

Path predicate (input \( X_0 \) et \( Y_0 \)) — Unrealistic perfect symbolic reasoning

\[ \top \land Z_1 = X_0 \times X_0 \land Z_1 = Y_0 \]
Goal = find input leading to ERROR
(assume we have only a solver for linear integer arith.)

\[ f(int \ x, int \ y) \{ z=x^2; \text{if } (y == z) \text{ ERROR}; \text{else OK} \} \]

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>input(x,y)</td>
</tr>
<tr>
<td>1</td>
<td>z := x * x</td>
</tr>
<tr>
<td>2</td>
<td>if (z == y) [True branch]</td>
</tr>
</tbody>
</table>

Path predicate (input \( X_0 \) et \( Y_0 \)) — Unrealistic perfect symbolic reasoning
OK, but how to solve? \( X \)
Bonus

About robustness

Goal = find input leading to ERROR
(assume we have only a solver for linear integer arith.)

\[ f(int \ x, int \ y) \{ z=x\times x; \ if \ (y == z) \ ERROR; \ else \ OK \} \]

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>input(x,y)</td>
</tr>
<tr>
<td>1</td>
<td>z := x * x</td>
</tr>
<tr>
<td>2</td>
<td>if (z == y) [True branch]</td>
</tr>
</tbody>
</table>

Path predicate (input \ X_0 \ et \ Y_0) — Limited symbolic reasoning
Goal = find input leading to ERROR
(assume we have only a solver for linear integer arith.)

f(int x, int y) {z=x*x; if (y == z) ERROR; else OK }
About robustness

Goal = find input leading to ERROR
(assume we have only a solver for linear integer arith.)

\[ f(\text{int } x, \text{int } y) \{ z = x \times x; \text{ if (} y == z \text{) ERROR; else OK} \} \]

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>input(x, y)</td>
</tr>
<tr>
<td>1</td>
<td>( z := x \times x )</td>
</tr>
<tr>
<td>2</td>
<td>if (z == y) [True branch]</td>
</tr>
</tbody>
</table>

Path predicate \( (\text{input } X_0 \text{ et } Y_0) \) — Limited symbolic reasoning
\( T \land T \)
Goal = find input leading to ERROR
(assume we have only a solver for linear integer arith.)

\[ f(\text{int } x, \text{ int } y) \{ z = x \times x; \text{ if } (y == z) \text{ ERROR}; \text{ else } \text{ OK } \} \]

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>input(x,y)</td>
</tr>
<tr>
<td>1</td>
<td>z := x \times x</td>
</tr>
<tr>
<td>2</td>
<td>if \ (z == y) \ [True branch]</td>
</tr>
</tbody>
</table>

Path predicate (input \( X_0 \) et \( Y_0 \)) — Limited symbolic reasoning
\[ \top \land \top \land Z_1 = Y_0 \]
**Goal** = find input leading to **ERROR**
(assume we have only a solver for linear integer arith.)

```c
f(int x, int y) {z=x*x; if (y == z) ERROR; else OK }
```

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>input(x,y)</td>
</tr>
<tr>
<td>1</td>
<td>z := x * x</td>
</tr>
<tr>
<td>2</td>
<td>if (z == y) [True branch]</td>
</tr>
</tbody>
</table>

**Path predicate (input \( X_0 \) et \( Y_0 \))** — Limited symbolic reasoning
Incorrect, may find a bad solution (ex : \( X_0 = 10, \ Y_0 = 34 \) ×
Goal = find input leading to ERROR
(assume we have only a solver for linear integer arith.)

\[ f(int x, int y) \{ z=x*x; \text{if (y == z) ERROR; else OK} \} \]

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>input(x,y)</td>
</tr>
<tr>
<td>1</td>
<td>z := x * x</td>
</tr>
<tr>
<td>2</td>
<td>if (z == y) [True branch]</td>
</tr>
</tbody>
</table>

Path predicate (input \( X_0 \) et \( Y_0 \)) — Limited dynamic symbolic reasoning

\[ \top \]
Goal = find input leading to ERROR
(assume we have only a solver for linear integer arith.)

\[ f(\text{int } x, \text{ int } y) \{ z=x \times x; \text{ if } (y == z) \text{ ERROR; else OK } \} \]

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>input(x,y)</td>
</tr>
<tr>
<td>1</td>
<td>( z := x \times x )</td>
</tr>
<tr>
<td>2</td>
<td>if (z == y) [True branch]</td>
</tr>
</tbody>
</table>

Path predicate (input \( X_0 \) et \( Y_0 \)) — Limited dynamic symbolic reasoning
\[ \top \land Z_1 = X_0 \times X_0 \] [assume runtime values : \( x=3, z=9 \)]
Goal = find input leading to ERROR
(assume we have only a solver for linear integer arith.)

\[ f(int \ x, \ int \ y) \{ z=x*x; \ if \ (y == z) \ ERROR; \ else \ OK \} \]

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>input(x,y)</td>
</tr>
<tr>
<td>1</td>
<td>z := x * x</td>
</tr>
<tr>
<td>2</td>
<td>if (z == y) [True branch]</td>
</tr>
</tbody>
</table>

Path predicate (input \( X_0 \) et \( Y_0 \)) — Limited dynamic symbolic reasoning

\[ \top \land Z_1 = 9 \land X_0 = 3 \]
Goal = find input leading to ERROR  
(assume we have only a solver for linear integer arith.)

\( f(\text{int } x, \text{int } y) \{ z = x \times x; \text{if } (y == z) \text{ERROR}; \text{else } \text{OK} \} \)

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>input(x,y)</td>
</tr>
<tr>
<td>1</td>
<td>z := x * x</td>
</tr>
<tr>
<td>2</td>
<td>if (z == y) [True branch]</td>
</tr>
</tbody>
</table>

Path predicate (input \( X_0 \) et \( Y_0 \)) — Limited dynamic symbolic reasoning
\[ \top \land Z_1 = 9 \land X_0 = 3 \land Z_1 = Y_0 \]
About robustness

**Goal** = find input leading to ERROR
(assume we have only a solver for linear integer arith.)

```c
f(int x, int y) {z=x*x; if (y == z) ERROR; else OK }
```

<table>
<thead>
<tr>
<th>Loc</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>input(x,y)</td>
</tr>
<tr>
<td>1</td>
<td>z := x * x</td>
</tr>
<tr>
<td>2</td>
<td>if (z == y) [True branch]</td>
</tr>
</tbody>
</table>

Path predicate (input $X_0$ et $Y_0$) — Limited dynamic symbolic reasoning
Correct, find a real solution (ex : $X_0 = 3$, $Y_0 = 9$) ✓