Symbolic Reasoning & Concrete Execution

Andrii Vozniuk

Advanced Topics in Software Systems

EPFL

December 2, 2011

“Increasing vision is increasingly expensive”

From the “Andromeda Strain’’ by Michael Crichton

Outline

- Definitions
- Problem statement
- Solution: higher-order test generation
- Solution: symbolic execution with mixed concrete-symbolic solving
- Summary

Definitions

- ***Concretization***
- ***Sound*** path constraint *pc(w)*

```
if (x>=0) then return 0;
else return -1;
```

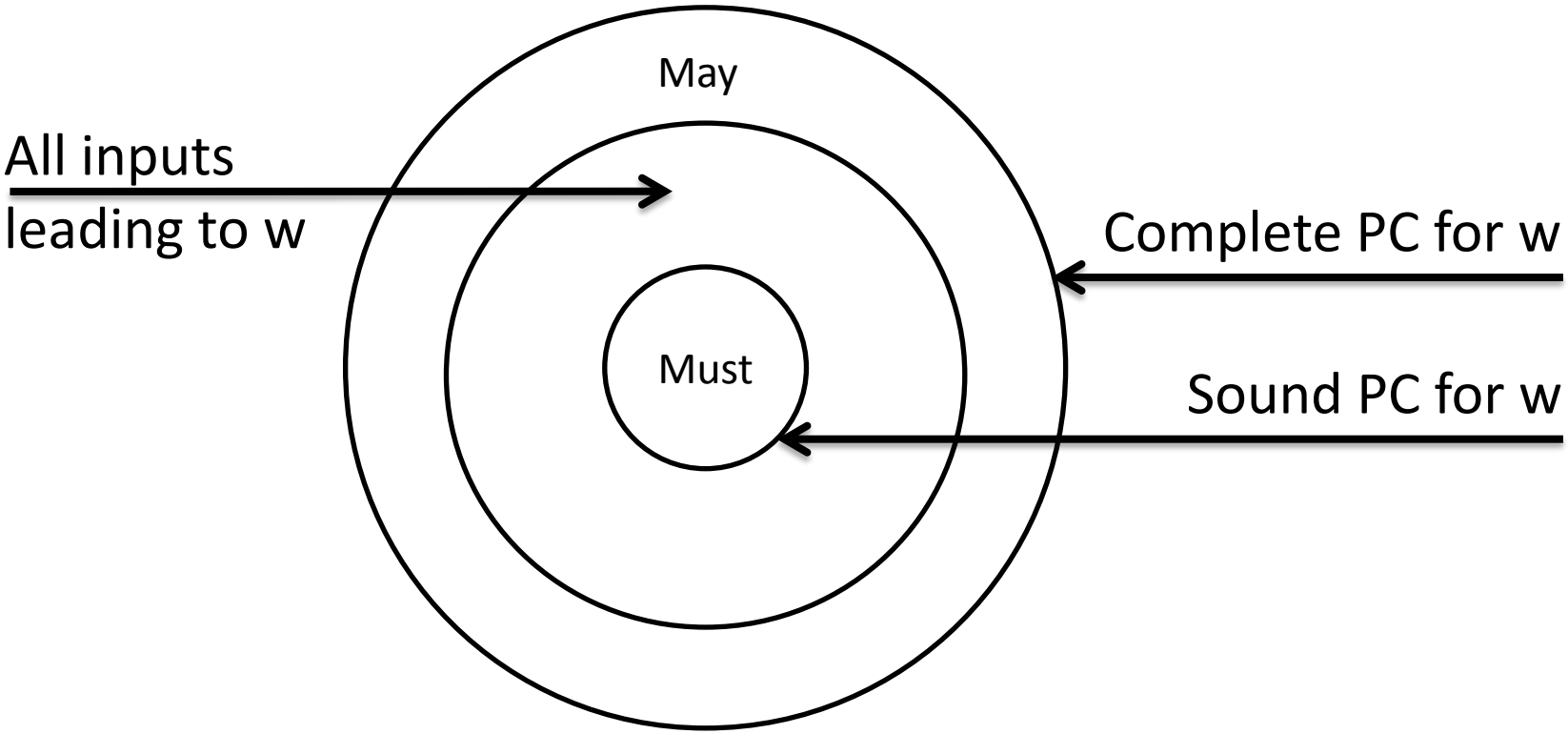
Sound	Unsound
x>=10	x>-3
x<0	x<=5

- ***Complete*** path constraint *pc(w)*

```
if (x>=0) then return 0;
else return -1;
```

Complete	Incomplete
x>=-3	x>10
x<10	x<=-5

Definitions



Problem Statement

```
int foo(int x, int y) {
  if (x == hash(y)) {…
    if (y == 10) return -1; //error
} … }
```

- Might happen:
  - `hash` is too complex or
  - code of `hash` is not available
- Constraint solver cannot reason symbolically
- Static Test Generation cannot generate the desired inputs

Problem Statement

```
int foo(int x, int y) {
  if (x == hash(y)) {…
    if (y == 10) return -1; //error
} … }
```

- Might happen:
  - `hash` is too complex or
  - code of `hash` is not available
- Constraint solver cannot reason symbolically
- Static Test Generation cannot generate the desired inputs

Problem Statement

```
int foo(int x, int y) {
  if (x == hash(y)) {…
    if (y == 10) return -1; //error
} … }
```

- Might happen:
  - `hash` is too complex or
  - code of `hash` is not available
- Constraint solver cannot reason symbolically
- Static Test Generation cannot generate the desired inputs

Problem Statement

- Idea: use concrete values and randomization
- Dynamic Test Generation, as in the DART:
  - starts with random run
  - uses run-time information for concretization
  - should systematically execute all branches

Problem Statement

```
int foo(int x, int y) {
  if (x == hash(y)) {…
    if (y == 10) return -1; //error
} … }
Assume: x=567, y=42, hash(42) == 567
```

- First run of DART with: `x=567,y=42`
- Concretizes `hash`: `x=567∧y≠10`
- Not sound: `x=567∧y=15,hash(15)=600`
- May lead to divergence: `x=567∧y=10`

Problem Statement

```
int foo(int x, int y) {
  if (x == hash(y)) {…
    if (y == 10) return -1; //error
} … }
Assume: x=567, y=42, hash(42) == 567
```

- First run of DART with: `x=567,y=42`
- Concretizes `hash`: `x=567∧y≠10`
- Not sound: `x=567∧y=15,hash(15)=600`
- May lead to divergence: `x=567∧y=10`

Problem Statement

```
int foo(int x, int y) {
  if (x == hash(y)) {…
    if (y == 10) return -1; //error
} … }
Assume: x=567, y=42, hash(42) == 567
```

- First run of DART with: $x=567, y=42$
- Concretizes `hash`: $x=567 \wedge y \neq 10$
- Not sound: $x=567 \wedge y=15, hash(15)=600$
- May lead to divergence: $x=567 \wedge y=10$

Problem Statement

```
int foo(int x, int y) {
  if (x == hash(y)) {…
    if (y == 10) return -1; //error
} … }
Assume: x=567, y=42, hash(42) == 567
```

- First run of DART with: `x=567,y=42`
- Concretizes `hash`: `x=567∧y≠10`
- Not sound: `x=567∧y=15,hash(15)=600`
- May lead to divergence: `x=567∧y=10`

Problem Statement

How to deal with imprecision caused by complex functions, in order to improve test generation?

Problem Statement

How to deal with imprecision caused by complex functions, in order to improve test generation?

-Use concrete values cleverly

Higher-Order Test Generation*

- Tries to model concretization symbolically
- Uses a higher-order logic for path constraints
- Has three steps:
  1. hash(y)→h(y)
  2. $\exists x, y\colon\ x = h(y)$
  3. <567,hash(42)> ,<312,hash(58)>,…

* Higher Order Test Generation, P. Godefroid, *PLDI 2011*

SymbEx with Uninterpreted Functions

```
int foo(int x, int y) {
  if (x == hash(y)) {…
    if (y == 10) return -1; //error
} … }
Assume: x=567, y=42, hash(42) == 567
```

- An uninterpreted function (UF) symbol **h** is introduced
- The path constraint becomes $x=\mathbf{h}(y)$
- Example: $x=h(y) \wedge y \neq 10 \wedge y=42$

Generating Test from Validity Proofs

- Sound path constraints with UFs symbols
- Validity proofs instead of satisfiability proofs
- Pre-processing of path constraints:

  $a(h,x) \rightarrow \exists x : a(h,x) \rightarrow \exists x, y\colon\ [\forall h]\ x = h(y)$
- Validity check: find a strategy to make the formula always true
- Example: “fix *y*, then set *x* to the value *h(y)*”

Uninterpreted Functions Sampling

- "fix *y*, then set *x* to the value *h(y)*"
- To compute concrete input vector (x,y) concrete values of h(y) are needed
- Record concrete values in runtime each function application
- Example:`<567,hash(42)> → input y=42, x=567`
- Can be used to generate additional constraints

Symbolic Execution with Mixed Concrete-Symbolic Solving*

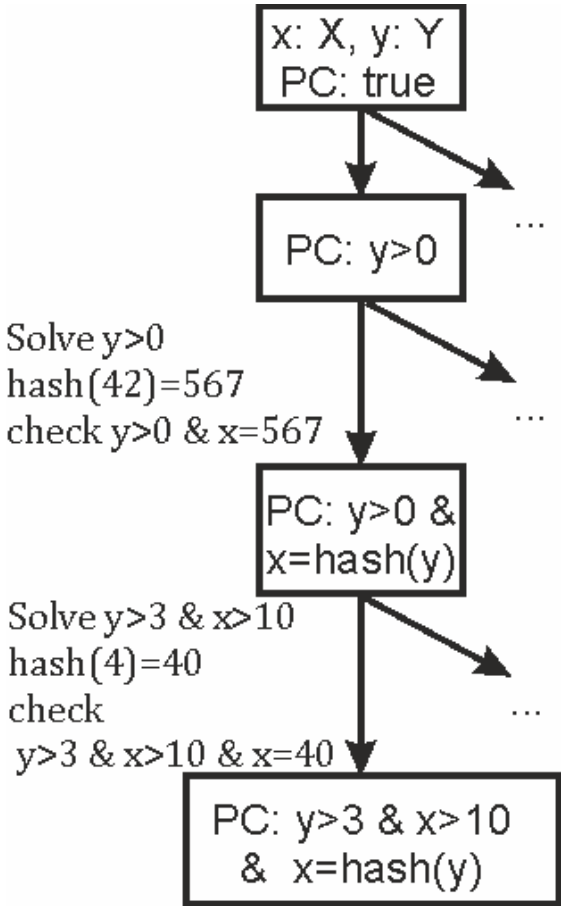
- The path conditions are split into 2 sets:
  1. simplePC - can be solved by a decision procedure
  2. complexPC - complex constraints with UFs
- Idea: solve the simple part of PC, use obtained values to concretize complexPC

* Symbolic Execution with Mixed Concrete-Symbolic Solving, C. Păsăreanu, N. Rungta, W. Visser, *ISSTA 2011*

Example

```
@Concrete("true")
@Partition({"y>3.0","y<=3.0"})
double hash(double y)

int foo(int x, int y) {
if (y > 0) {
  if (x == hash(y)) {
    if (x>10 && y>3) return -1;
  }
}
```



@Concrete("true") marks the uninterpreted methods

Heuristics

- Goal: force the solver to generate “more interesting” solutions
- Three heuristics:

1. Incremental solving
2. Partitioning: `@Partition({”y>3.0”,”y<=3.0”})`
3. Random solving

Summary

High Order TG	vs	**SE with Mixed Solving**
Theoretical, well grounded Not implemented yet		Implemented in Symbolic PathFinder Demonstrated results
Improves DART		Improves ‘‘classical’’ SymbEx
Concrete values obtained from the run-time		Uses concrete solutions of *simplePC* to simplify *complexPC*
Concretization constraints are added to the PC for the rest of the path		May pick different concretizations on the same path
Relies on validity checking, which poses implementation challenge		Relies on standard constraint solving
Sound and incomplete		Sound and incomplete

Summary

High Order TG	vs	**SE with Mixed Solving**
Theoretical, well grounded Not implemented yet		Implemented in Symbolic PathFinder Demonstrated results
Improves DART		Improves ‘’classical’’ SymbEx
Concrete values obtained from the run-time		Uses concrete solutions of *simplePC* to simplify *complexPC*
Concretization constraints are added to the PC for the rest of the path		May pick different concretizations on the same path
Relies on validity checking, which poses implementation challenge		Relies on standard constraint solving
Sound and incomplete		Sound and incomplete

Summary

High Order TG	vs	SE with Mixed Solving
Theoretical, well grounded Not implemented yet		Implemented in Symbolic PathFinder Demonstrated results
Improves DART		Improves ‘’classical’’ SymbEx
Concrete values obtained from the run-time		Uses concrete solutions of *simplePC* to simplify *complexPC*
Concretization constraints are added to the PC for the rest of the path		May pick different concretizations on the same path
Relies on validity checking, which poses implementation challenge		Relies on standard constraint solving
Sound and incomplete		Sound and incomplete

Summary

High Order TG	vs	SE with Mixed Solving
Theoretical, well grounded Not implemented yet		Implemented in Symbolic PathFinder Demonstrated results
Improves DART		Improves ‘’classical’’ SymbEx
Concrete values obtained from the run-time		Uses concrete solutions of *simplePC* to simplify *complexPC*
Concretization constraints are added to the PC for the rest of the path		May pick different concretizations on the same path
Relies on validity checking, which poses implementation challenge		Relies on standard constraint solving
Sound and incomplete		Sound and incomplete

Summary

High Order TG	vs	**SE with Mixed Solving**
Theoretical, well grounded Not implemented yet		Implemented in Symbolic PathFinder Demonstrated results
Improves DART		Improves ‘’classical’’ SymbEx
Concrete values obtained from the run-time		Uses concrete solutions of *simplePC* to simplify *complexPC*
Concretization constraints are added to the PC for the rest of the path		May pick different concretizations on the same path
Relies on validity checking, which poses implementation challenge		Relies on standard constraint solving
Sound and incomplete		Sound and incomplete

Thank your for your attention!

Andrii.Vozniuk@epfl.ch

Discussion

1. How far could we go increasing order of the logic? 4
2. Integrate computer algebra systems with constraints solvers? 6
3. For what types of functions partitioning is useful? 4
4. Input-output pairs of uninterpreted functions gathering strategies. 3
5. How to come with strategies for validity proofs? 5
6. Comparison between KLEE, DART, PathFinder. 3
7. Dedicated hardware for constraints solving. 7
8. Concretization on the path: one vs many? 5

The main obstacle

- Functionality vs Correctness 2
- Scalability / Performance 4
- Usability 3
- Bugs classification 0
- The Oracle issue 2