Symbolic Computation via Program Transformation

Henrich Lauko, Petr Ročkai and Jiří Barnat

ParaDiSe
Parallel & Distributed
Systems Laboratory

Masaryk University

Brno, Czech Republic

18th October 2018

Agenda

1. Context of symbolic computation
2. Transformation-based approach
3. Integration to tool with explicit computation

Symbolic Computation

Explicit computation

- variables represent concrete values
- compiled or interpreted programs

```
x ← input() // x = 7
if (x > 0)
  ...
else
  ...
```

Symbolic computation

- variables represent sets of possible values
- mostly interpreted

```
x ← input() // x = {...}
if (x > 0)
  ... // x = {v|v > 0}
else
  ... // x = {v|v <= 0}
```

- verification, test generation, concolic testing

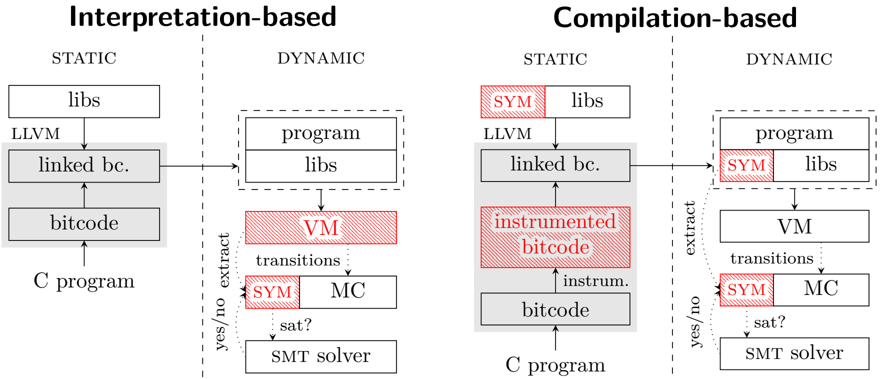
Symbolic Execution

- interpretation-based approach

```
a ← input()    // pc: true
b ← input()    // pc: true
if (a > 0)     // pc: a > 0
  if (b < 0)   // pc: a > 0 ∧ b < 0
    b ← a + 1  // pc: a > 0 ∧ b < 0, data: b = a + 1
  else
    b ← a - 1  // pc: a > 0 ∧ b ≥ 0, data: b = a − 1
```

- interpreter builds **formula** in some theory of SMT logic:
    1. data representation: $b = a + 1$
    2. path condition: $a > 0$
- program does not know anything about symbolic values
- TODO show apparoach img. here

- let the program build *data representation* and *path condition*



- minimizes complexity of the verification algorithm

Goals

1. mixing of explicit and symbolic computation
2. expose a small interface to the rest of the system
3. impose minimal run-time overhead

Transformation of Bitcode

```
x:int  ← input()              x: sym_int  ← lift(*)
y:int  ← factorial(7)         y: int      ← factorial(7)
z:int  ← x + y                z: sym_int  ← sym_add(x, lift(y))
b:bool ← y < z                b: sym_bool ← sym_lt(x, z)
```

- transform instructions, types, functions
- preserve concrete computation
- lift concrete values
- provide library with implementation of symbolic operations:
  - `lift`, `lower`, `sym_add`, ...

Generalization of the Transformation

1. syntactically abstract the input program:
   - values: `int` → `abstract_int`
   - instructions: `add` → `abstract_add`
2. concretely realize abstraction:
   - values: `abstract_int` → `sym_int`
   - instructions: `abstract_add` → `sym_add`

- realization inserts an arbitrary domain that is provided

Closer look on the Transformation 1.

1. Branching

```
cond: bool ← x < 0
if (cond)
  ...
else
  ...
```

```
cond: sym_bool ← sym_lt(x, 0)
if (*)
  x': sym_int ← assume(cond)
  ...
else
  x': sym_int ← assume(!cond)
  ...
```

- assume constrains values of `x`
- extend *path condition*

Closer look on the Transformation 2.

2. Aggregate types

```
arr: int[]  ← [1, 2, 3]
arr[1]: int ← input()
```

- we want to minimize the number of symbolic values

Solution: use discriminated union type

- realize abstract value as union of *concrete* and *symbolic* value

```
arr: union[]  ← [1, 2, 3] // either int or sym_int
arr[1]: int ← lift(*)
```

- similarly deal with recursive structures

Closer look on the Transformation 3.

3. Function Calls

- how to transform functions with symbolic arguments?

```
int foo(a: int, b: int, c: int)
```

- may produce exponentially many duplicates:

```
int foo(a: sym_int, b: int, c: int)
int foo(a: int, b: sym_int, c: int)
int foo(a: int, b: int, c: sym_int)
int foo(a: sym_int, b: sym_int, c: int)
...
```

- resolve return type

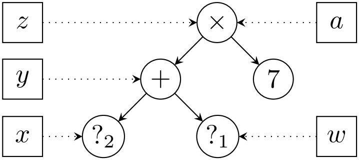
Solution: static analysis + use discriminated union

```
union foo(a: union, b: union, c: int)
```

Data Representation

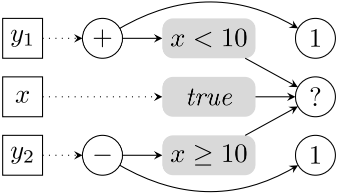
Symbolic execution:

```
a: pointer ← malloc()
w: sym_int ← lift(*)
x: sym_int ← lift(*)
y: sym_int ← sym_add(w, x)
z: sym_int ← sym_mul(y, 7)
store z → a
```

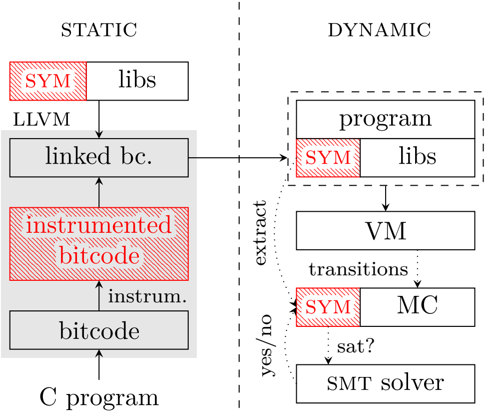


Branching example:

```
x : sym_int ← lift(*)
if (*) // nondeterministic
  x': sym_int ← assume(x < 10)
  y : sym_int ← sym_add(x, 1)
else
  x': sym_int ← assume(x < 10)
  y : sym_int ← sym_sub(x, 1)
```



Symbolic Verification Algorithm



Required support in a tool:

- nondeterminism
- feasibility check
- equality check
- values metadata

Simpler domains do not even need *SMT* support (sign domain).

Results

Component sizes: (lines of code)

	DIVINE	KLEE	SymDIVINE	CBMC
symbolic support	5.4	24.2	7	39.8
shared code	136.5	125	423	27.5

- reduced complexity of verification tool

SV-COMP Benchmarks:
TODO

Conclusion

Goals

1. mixing of explicit and symbolic computation ✓
2. expose a small interface to the rest of the system ✓
3. impose minimal run-time overhead ✓

Summary

- introduced compilation-based symbolic verification
- generalized approach to the abstraction of programs