Symbolic Program Decomposition & Compositional Symbolic Execution

Jonas Wagner

Advanced Topics in Software Systems
École polytechnique fédérale de Lausanne

November 25, 2011

Introduction

The whole is greater than the sum of its parts.
– Aristotle, Metaphysics

Example

- A quadratic equation is given as $ax^2 + bx + c = 0$
- There is a solution iff $\Delta = b^2 - 4ac \geq 0$

```
def main(a, b, c):
  if b*b - a*c*4 >= 0:
    print("OK")
  else:
    print("No solution!")

main(1, 2, 0)  # OK
main(1, 2, 3)  # No solution!
```

- Symbex can easily find inputs for both cases

Example

- A quadratic equation is given as $ax^2 + bx + c = 0$
- There is a solution iff $\Delta = b^2 - 4ac \geq 0$

```
def main(a, b, c):
    if b*b - a*c*4 >= 0:
        print("OK")
    else:
        print("No solution!")

main(1, 2, 0)   # OK
main(1, 2, 3)   # No solution!
```

- Symbex can easily find inputs for both cases

Example

- A quadratic equation is given as $ax^2 + bx + c = 0$
- There is a solution iff $\Delta = b^2 - 4ac \geq 0$

```
def main(a, b, c):
    if b*b - a*c*4 >= 0:
        print("OK")
    else:
        print("No solution!")

main(1, 2, 0)   # OK
main(1, 2, 3)   # No solution!
```

- Symbex can easily find inputs for both cases

Example

- With interval arithmetic

```
class Ival:
  def __mul__(self, other):
    if isinstance(other, int):
      other = Ival(other, other)
    vals = [self.lo*other.lo, self.lo*other.hi,
            self.hi*other.lo, self.hi*other.hi]
    return Ival(min(vals), max(vals))
```

```
def main(a, b, c):
  if b*b - a*c*4 >= 0:
    print("OK")
  else:
    print("No solution?")

main(Ival(1,2), Ival(2,3), Ival(-1,0))   #OK
```

- Now we have thousands of paths!

Example

- With interval arithmetic

```
class Ival:
  def __mul__(self, other):
    if isinstance(other, int):
      other = Ival(other, other)
    vals = [self.lo*other.lo, self.lo*other.hi,
            self.hi*other.lo, self.hi*other.hi]
    return Ival(min(vals), max(vals))
```

```
def main(a, b, c):
  if b*b - a*c*4 >= 0:
    print("OK")
  else:
    print("No solution?")
```

```
main(Ival(1,2), Ival(2,3), Ival(-1,0))   #OK
```

- Now we have thousands of paths!

Example

- With interval arithmetic

```
class Ival:
  def __mul__(self, other):
    if isinstance(other, int):
      other = Ival(other, other)
    vals = [self.lo*other.lo, self.lo*other.hi,
            self.hi*other.lo, self.hi*other.hi]
    return Ival(min(vals), max(vals))

def main(a, b, c):
  if b*b - a*c*4 >= 0:
    print("OK")
  else:
    print("No solution?")

main(Ival(1,2), Ival(2,3), Ival(-1,0))   #OK
```

- Now we have thousands of paths!

Example

```
1  def __mul__(self, other):
2      if isinstance(other, int):
3          other = Interval(other, other)
4      vals = [self.lo*other.lo, self.lo*other.hi,
5              self.hi*other.lo, self.hi*other.hi]
6      return Interval(min(vals), max(vals))
7
8  def main(a, b, c):
9      if b*b - a*c*4 >= 0:
10         print("OK")
11     else:
12         print("No solution?")
```

1. Path family condition
2. Find defs
3. Split path family
4. Replace values
5. Repeat

Example

```
1  def __mul__(self, other):
2      if isinstance(other, int):
3          other = Interval(other, other)
4      vals = [self.lo*other.lo, self.lo*other.hi,
5              self.hi*other.lo, self.hi*other.hi]
6      return Interval(min(vals), max(vals))
7
8  def main(a, b, c):
9      if b*b - a*c*4 >= 0:
10         print("OK")
11     else:
12         print("No solution?")
```

1. Path family condition
2. Find defs
3. Split path family
4. Replace values
5. Repeat

$$b_9 * b_9 - 4 * a_9 * c_9 \geq 0$$

Example

```
1  def __mul__(self, other):
2      if isinstance(other, int):
3          other = Interval(other, other)
4      vals = [self.lo*other.lo, self.lo*other.hi,
5              self.hi*other.lo, self.hi*other.hi]
6      return Interval(min(vals), max(vals))
7
8  def main(a, b, c):
9      if b*b - a*c*4 >= 0:
10         print("OK")
11     else:
12         print("No solution?")
```

1. Path family condition
2. Find defs
3. Split path family
4. Replace values
5. Repeat

$$b_9 * b_9$$

Example

```
1  def __mul__(self, other):
2      if isinstance(other, int):
3          other = Interval(other, other)
4      vals = [self.lo*other.lo, self.lo*other.hi,
5              self.hi*other.lo, self.hi*other.hi]
6      return Interval(min(vals), max(vals))
7
8  def main(a, b, c):
9      if b*b - a*c*4 >= 0:
10         print("OK")
11     else:
12         print("No solution?")
```

1. Path family condition
2. Find defs
3. Split path family
4. Replace values
5. Repeat

$$\min(vals)_6, \max(vals)_6$$

Example

```
1  def __mul__(self, other):
2      if isinstance(other, int):
3          other = Interval(other, other)
4      vals = [self.lo*other.lo, self.lo*other.hi,
5              self.hi*other.lo, self.hi*other.hi]
6      return Interval(min(vals), max(vals))
7
8  def main(a, b, c):
9      if b*b - a*c*4 >= 0:
10         print("OK")
11     else:
12         print("No solution?")
```

1. Path family condition
2. Find defs
3. Split path family
4. Replace values
5. Repeat

$$s_1.lo * o_1.lo < \cdots < s_1.hi * o_1.hi : s_1.lo * o_1.lo, s_1.hi * o_1.hi$$

$$s_1.lo * o_1.hi < \cdots < s_1.hi * o_1.hi : s_1.lo * o_1.hi, s_1.hi * o_1.hi$$

$$s_1.lo * o_1.hi < \cdots < s_1.lo * o_1.lo : s_1.lo * o_1.hi, s_1.lo * o_1.lo$$

$$s_1.hi * o_1.hi < \cdots < s_1.lo * o_1.lo : s_1.hi * o_1.hi, s_1.lo * o_1.lo$$

Example

```
1  def __mul__(self, other):
2      if isinstance(other, int):
3          other = Interval(other, other)
4      vals = [self.lo*other.lo, self.lo*other.hi,
5              self.hi*other.lo, self.hi*other.hi]
6      return Interval(min(vals), max(vals))
7
8  def main(a, b, c):
9      if b*b - a*c*4 >= 0:
10         print("OK")
11     else:
12         print("No solution?")
```

1. Path family condition
2. Find defs
3. Split path family
4. Replace values
5. Repeat

$$b_8.lo * b_8.lo < \cdots < b_8.hi * b_8.hi : b_8.lo * b_8.lo, b_8.hi * b_8.hi$$

$$b_8.lo * b_8.hi < \cdots < b_8.hi * b_8.hi : b_8.lo * b_8.hi, b_8.hi * b_8.hi$$

$$b_8.lo * b_8.hi < \cdots < b_8.lo * b_8.lo : b_8.lo * b_8.hi, b_8.lo * b_8.lo$$

$$b_8.hi * b_8.hi < \cdots < b_8.lo * b_8.lo : b_8.hi * b_8.hi, b_8.lo * b_8.lo$$

Example

```
1  def __mul__(self, other):
2      if isinstance(other, int):
3          other = Interval(other, other)
4      vals = [self.lo*other.lo, self.lo*other.hi,
5              self.hi*other.lo, self.hi*other.hi]
6      return Interval(min(vals), max(vals))
7
8  def main(a, b, c):
9      if b*b - a*c*4 >= 0:
10         print("OK")
11     else:
12         print("No solution?")
```

1. Path family condition
2. Find defs
3. Split path family
4. Replace values
5. Repeat

same procedure for

$$a_9 * c_9 * 4$$

Symbolic Program Decomposition

- Using SPD, we can analyze subexpressions individually.
- The number of paths explored is $\#P_1 + \#P_2$ instead of $\#P_1 \cdot \#P_2$
- Pros of SPD:
  - exponential speed-up
  - demand-driven
- Cons of SPD:
  - relies on static dependency analysis (read: difficulties with pointers)

Symbolic Program Decomposition

- Using SPD, we can analyze subexpressions individually.
- The number of paths explored is $\#P_1 + \#P_2$ instead of $\#P_1 \cdot \#P_2$
- Pros of SPD:
  - exponential speed-up
  - demand-driven
- Cons of SPD:
  - relies on static dependency analysis (read: difficulties with pointers)

Symbolic Program Decomposition

- Using SPD, we can analyze subexpressions individually.
- The number of paths explored is $\#P_1 + \#P_2$ instead of $\#P_1 \cdot \#P_2$
- Pros of SPD:
  - exponential speed-up
  - demand-driven
- Cons of SPD:
  - relies on static dependency analysis (read: difficulties with pointers)

Compositional Symbolic Execution

- Idea: explore each function individually
  - on first execution, generate a *summary*
  - on subsequent calls, apply this summary
- Similar speed-up as SPD: $\#P_1 \cdot \#P_2 \rightarrow \#P_1 + \#P_2 + c$

Example

```
1  def __mul__(self, other):
2      if isinstance(other, int):
3          other = Interval(other, other)
4      vals = [self.lo*other.lo, self.lo*other.hi,
5              self.hi*other.lo, self.hi*other.hi]
6      return Interval(min(vals), max(vals))
7
8  def main(a, b, c):
9      if b*b - a*c*4 >= 0:
10         print("OK")
11     else:
12         print("No solution?")
```

1. Execute to function call
2. Explore all paths in function
3. Later, apply summary

Example

```
1  def __mul__(self, other):
2      if isinstance(other, int):
3          other = Interval(other, other)
4      vals = [self.lo*other.lo, self.lo*other.hi,
5              self.hi*other.lo, self.hi*other.hi]
6      return Interval(min(vals), max(vals))
7
8  def main(a, b, c):
9      if b*b - a*c*4 >= 0:
10         print("OK")
11     else:
12         print("No solution?")
```

1. Execute to function call
2. Explore all paths in function
3. Later, apply summary

Start with random input: $b = (1, 2)$

Yields $s_1.lo * o_1.lo < \ldots \quad \wedge \quad res.lo = s_1.lo * o_1.lo$

Example

```
1  def __mul__(self, other):
2      if isinstance(other, int):
3          other = Interval(other, other)
4      vals = [self.lo*other.lo, self.lo*other.hi,
5              self.hi*other.lo, self.hi*other.hi]
6      return Interval(min(vals), max(vals))
7
8  def main(a, b, c):
9      if b*b - a*c*4 >= 0:
10         print("OK")
11     else:
12         print("No solution?")
```

1. Execute to function call
2. Explore all paths in function
3. Later, apply summary

Flip constraints s. t. `min` takes 2nd element (e. g. $b = (-1, 2)$):
Yields $s_1.lo * o_1.hi < \ldots \quad \wedge \quad res.lo = s_1.lo * o_1.hi$

Example

```
1  def __mul__(self, other):
2      if isinstance(other, int):
3          other = Interval(other, other)
4      vals = [self.lo*other.lo, self.lo*other.hi,
5              self.hi*other.lo, self.hi*other.hi]
6      return Interval(min(vals), max(vals))
7
8  def main(a, b, c):
9      if b*b - a*c*4 >= 0:
10         print("OK")
11     else:
12         print("No solution?")
```

1. Execute to function call
2. Explore all paths in function
3. Later, apply summary

Flip constraints s. t. `min` takes 3rd element:
Unsatisfiable (because `self == other`)

Example

```
1  def __mul__(self, other):
2      if isinstance(other, int):
3          other = Interval(other, other)
4      vals = [self.lo*other.lo, self.lo*other.hi,
5              self.hi*other.lo, self.hi*other.hi]
6      return Interval(min(vals), max(vals))
7
8  def main(a, b, c):
9      if b*b - a*c*4 >= 0:
10         print("OK")
11     else:
12         print("No solution?")
```

1. Execute to function call
2. Explore all paths in function
3. Later, apply summary

Flip constraints s. t. `min` takes 4^{th} element (e. g. $b = (2, 1)$):
Yields $s_1.hi * o_1.hi < \ldots \quad \wedge \quad res.lo = s_1.hi * o_1.hi$

Example

```
1  def __mul__(self, other):
2      if isinstance(other, int):
3          other = Interval(other, other)
4      vals = [self.lo*other.lo, self.lo*other.hi,
5              self.hi*other.lo, self.hi*other.hi]
6      return Interval(min(vals), max(vals))
7
8  def main(a, b, c):
9      if b*b - a*c*4 >= 0:
10         print("OK")
11     else:
12         print("No solution?")
```

1. Execute to function call
2. Explore all paths in function
3. Later, apply summary

Continue until all combinations of `min` and `max` results are found.

Example

```
1  def __mul__(self, other):
2      if isinstance(other, int):
3          other = Interval(other, other)
4      vals = [self.lo*other.lo, self.lo*other.hi,
5              self.hi*other.lo, self.hi*other.hi]
6      return Interval(min(vals), max(vals))
7
8  def main(a, b, c):
9      if b*b - a*c*4 >= 0:
10         print("OK")
11     else:
12         print("No solution?")
```

1. Execute to function call
2. Explore all paths in function
3. Later, apply summary

Resulting summary:

$$
\begin{array}{llll}
s_1.lo * o_1.lo < \ldots & \wedge & res.lo = s_1.lo * o_1.lo & \vee \\
s_1.lo * o_1.li < \ldots & \wedge & res.lo = s_1.lo * o_1.hi & \vee \\
 & \ldots & &
\end{array}
$$

Example

```
1  def __mul__(self, other):
2      if isinstance(other, int):
3          other = Interval(other, other)
4      vals = [self.lo*other.lo, self.lo*other.hi,
5              self.hi*other.lo, self.hi*other.hi]
6      return Interval(min(vals), max(vals))
7
8  def main(a, b, c):
9      if b*b - a*c*4 >= 0:
10         print("OK")
11     else:
12         print("No solution?")
```

1. Execute to function call
2. Explore all paths in function
3. Later, apply summary

Caveat: Summary is not complete because now arguments can differ.

Looks Like Magic

- SMART finds the same bugs as DART...
- but uses exponentially fewer program runs
- How can this be?

Trade-Off for Compositional Methods

- Compositional methods create a trade-off

```
1  if a < b:
2      res = a
3  else:
4      res = b
```

$\rightsquigarrow$

$$a < b \wedge res = a \quad \vee$$
$$a \geq b \wedge res = b$$

- 2 paths
- simple expression for `res`
- path condition for 1 path (same for all variables)

- 1 path
- disjunction
- path condition for many paths simultaneously (slicing)

Trade-Off for Compositional Methods

When should we trade many simple paths vs few complicated path families?

- SMART uses *functions*
  - Paths are merged if branches are in the same function
  - Some flexibility
  - Might miss opportunities for merging paths
- SPD uses *output*
  - Paths are merged if they produce the same value for a variable
  - Might miss opportunities for merging paths

Trade-Off for Compositional Methods

When should we trade many simple paths vs few complicated path families?

- SMART uses *functions*
  - Paths are merged if branches are in the same function
  - Some flexibility
  - Might miss opportunities for merging paths
- SPD uses *output*
  - Paths are merged if they produce the same value for a variable
  - Might miss opportunities for merging paths

Trade-Off for Compositional Methods

When should we trade many simple paths vs few complicated path families?

- SMART uses *functions*
  - Paths are merged if branches are in the same function
  - Some flexibility
  - Might miss opportunities for merging paths
- SPD uses *output*
  - Paths are merged if they produce the same value for a variable
  - Might miss opportunities for merging paths

Summary

- Compositional methods can provide exponential speed-up
  - More work is done by the SMT solver, less by SymbEx
  - Benefit from slicing
- SMART and SPD decide differently when to merge paths
- Same guarantees as traditional SymbEx (sound)
- Both methods offer different opportunities for abstraction

Summary

- Compositional methods can provide exponential speed-up
  - More work is done by the SMT solver, less by SymbEx
  - Benefit from slicing
- SMART and SPD decide differently when to merge paths
- Same guarantees as traditional SymbEx (sound)
- Both methods offer different opportunities for abstraction

Summary

- Compositional methods can provide exponential speed-up
  - More work is done by the SMT solver, less by SymbEx
  - Benefit from slicing
- SMART and SPD decide differently when to merge paths
- Same guarantees as traditional SymbEx (sound)
- Both methods offer different opportunities for abstraction