Compositional Symbolic Execution through Program Specialization

José Miguel Rojas[1] and Corina Păsăreanu[2]

[1] Technical University of Madrid, Spain
[2] CMU-SV/NASA Ames, Moffett Field, CA, USA

BYTECODE 2013
March 23, Rome, Italy

Software Testing and Test Data Generation

- Quality assurance
- Software testing
- Automated test data generation
- Wide variety of approaches to test data generation
- Symbolic execution (SPF, Symbolic PathFinder)
  - High cost of symbolic execution on large programs
    - Large (possibly infinite) number of execution paths
    - Size of their associated constraint sets
  - Additional complexity to handle arbitrary data structures
  - `babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc`

Our approach

- Scalability towards handling realistic programs
- Compositional reasoning in SPF (on top of JPF, Java PathFinder)
- Generation and re-utilization of method summaries to scale up
- Leveraging program specialization

Symbolic Execution

- King [Comm. ACM 1976], Clarke [IEEE TSE 1976]
- Analysis of programs with unspecified inputs
- Symbolic states represent sets of concrete states
  - symbolic values/expressions for variables
  - Path condition
  - Program counter
- For each path, build path condition
  - condition on inputs, for the execution to follow that path
  - check path condition satisfiability, explore only feasible paths

Symbolic Execution

- Renewed interest in recent years
- Applications: test-case generation, error detection,...
- Tools
  - CUTE and jCUTE (UIUC)
  - EXE and KLEE (Stanford)
  - CREST and BitBlaze (UC Berkeley)
  - Pex, SAGE, YOGI and PREfix (Microsoft)
  - **Symbolic Pathfinder (NASA)**
  - ...

Program Specialization

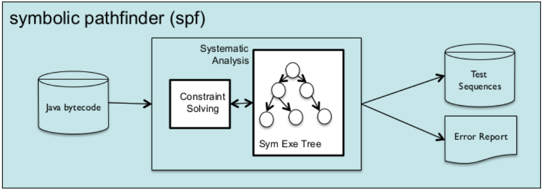
- Partial Evaluation and Automatic Program Generation [Jones, 1993]
- Partial evaluation creates a specialized version of a general program

```
int f(n,x) {
  if (n == 0)
    return 1;
  else
    if (even(n))
      return pow(f(n/2,x),2);
    else
      return x * f(n-1,x);
}
```

```
f3(x) {
  return x * pow(x * 1,2);
}
```

- Main benefit
  - speed of execution
  - specialized program faster than general program
- Some applications: compiler optimization, program transformation

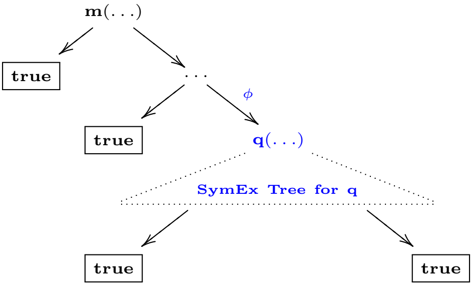
Symbolic PathFinder (SPF)



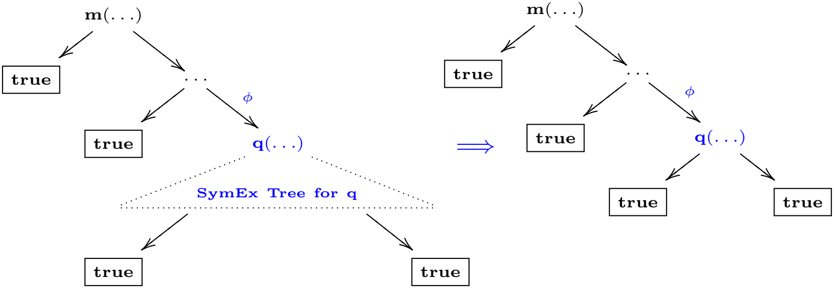
- Built on top of JPF (`http://babelfish.arc.nasa.gov/trac/jpf/`)
- SPF combines symbolic execution, model checking and constraint solving for test case generation
- Handles dynamic data structures, loops, recursion, multi-threading, arrays, strings,... [TACAS 2003, ISSTA 2008, ASE 2010]

- Non-standard interpreter of byte-codes
  - Symbolic execution replaces concrete execution semantics
  - Enables JPF to perform systematic symbolic analysis
- Lazy Initialization for arbitrary input data structures
  - Non-determinism handles aliasing
  - Different heap configurations explored explicitly
- Attributes store symbolic information
- Choice generators
  - Non-deterministic choices in branching conditions
- Listeners
  - Influence the search, collect and print results
- Bounded exploration to handle loops

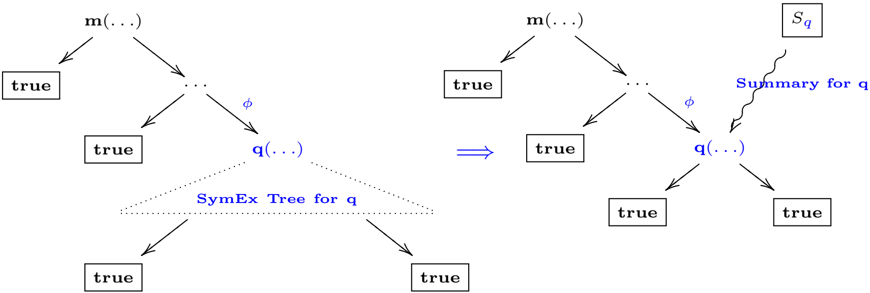
Compositional Symbolic Execution



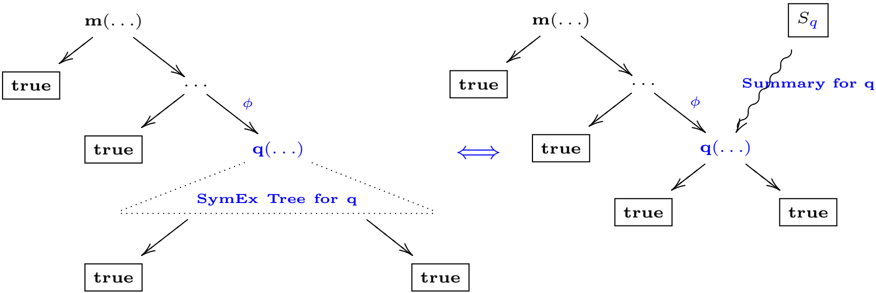
Compositional Symbolic Execution



Compositional Symbolic Execution



Compositional Symbolic Execution



Composition

- Compatibility check between summary cases of `q` and current state of `m`
- Only compatible summary cases are composed
- Summary cases's path constraints are conjoined with current state
- Summary for method `m` is created

Compositional Symbolic Execution

- Challenge
  - Composition in the presence of heap operations
- Previous approaches
  - Explicit representation of input and output heap [Albert et al., LOPSTR'10]
    - Potentially expensive, not natural in SPF
  - Summarize program as logical disjunctions [Godefroid, POPL'07]
    - No treatment of the heap
- Our approach
  - Leverage partial evaluation to build method summaries
  - Summaries are specialized versions of method code
  - Used to reconstruct the heap

Method Summaries

A method summary is a set of tuples of the form:

$$\langle \textbf{PC}, \textbf{HeapPC}, \textbf{SpC}, \textbf{CmpSch} \rangle$$

where:

- **PC**: Path Condition
  - Conjunction of constraints over symbolic inputs
  - Generated from conditional statements (`ifle`, `if_icmpeq`, etc.)
- **HeapPC**: Heap Path Condition
  - Conjunction of constraints over the heap
  - Generated via lazy initialization (`aload`, `getfield`, `getstatic`)
- **SpC**: Specialized Code
  - Sequence of byte-codes executed along a specific path
  - Does not contain conditional statements
- **CmpSch**: Composition schedule
  - For each `invoke` instruction, determines which case from the invoked method's summary to compose
  - Incremental, deterministic composition of method summaries

Method Summaries

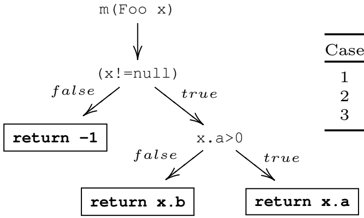
Java source code

```
int m(Foo x) {
  if (x != null)
    if (x.a > 0)
      return x.a;
    else
      return x.b;
  else return -1;
}
```

Java bytecode

```
0: aload x
1: ifnull 11
2: aload x
3: getfield a
4: ifle 8
5: aload x
6: getfield a
7: ireturn
8: aload x
9: getfield b
10: ireturn
11: iconst -1
12: ireturn
```

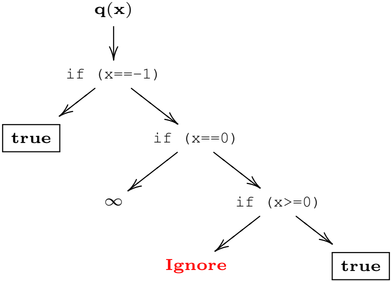
Symbolic Execution



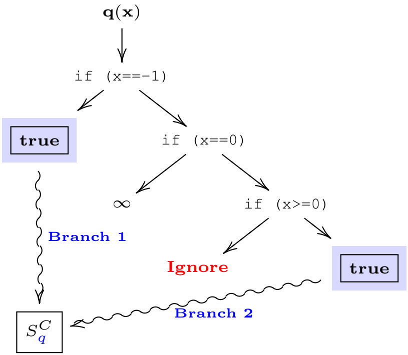
Method summary

Case	PC	HeapPC	Code
1	$\emptyset$	$\{x = null\}$	[iconst -1, ireturn]
2	$\{x.a > 0\}$	$\{x \neq null\}$	[aload x, getfield a, ireturn]
3	$\{x.a \leq 0\}$	$\{x \neq null\}$	[aload x, getfield b, ireturn]

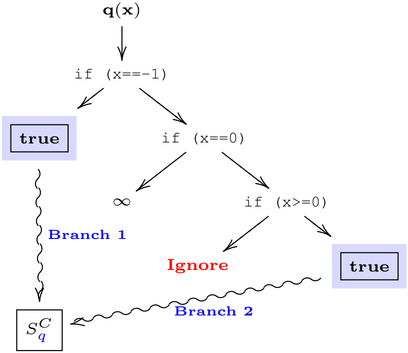
Generating Summaries



Generating Summaries



Generating Summaries



- The execution tree to be traversed is in general infinite. A termination criterion is needed
- A summary is a finite representation of the symbolic execution tree
- Complete for the given termination criterion, but Partial, in general
- Each element in a summary is said to be a (test) case of method `q`

```
Input:   insn:Instruction, currentState ≡ ⟨ pc, hpc, code, sched ⟩
  procedure SPECIALIZATION
    switch TYPE(insn) do
      case ConditionalInstruction
        code ← SLICECODE(code,insn)
      case InvokeInstruction
        COMPOSESUMMARY(getInvokedMethod(insn),duringSP)
        code ← APPEND(code,insn)
      case ReturnInstruction
        code ← APPEND(code,insn)
        STORESUMMARYCASE(pc,hpc,code,sched)
      case GotoInstruction
        IGNORE
      default
        code ← APPEND(code,insn)
  end procedure
```

Generating Summaries

Example of Program Specialization

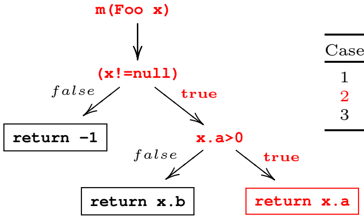
Java source code

```
int m(Foo x) {
  if (x != null)
    if (x.a > 0)
      return x.a;
    else
      return x.b;
  else return -1;
}
```

Java bytecode

```
0: aload x
1: ifnull 11
2: aload x
3: getfield a
4: ifle 8
5: aload x
6: getfield a
7: ireturn
8: aload x
9: getfield b
10: ireturn
11: iconst -1
12: ireturn
```

Symbolic Execution

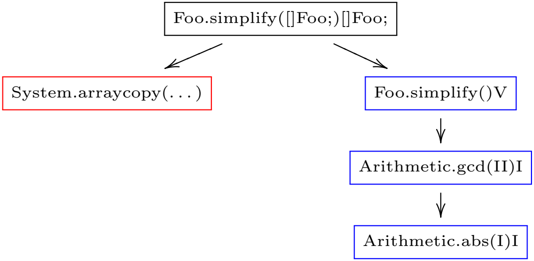


Method summary

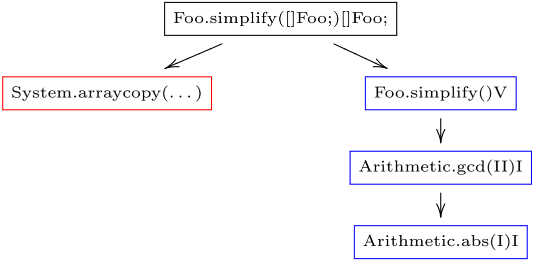
Case	PC	HeapPC	Code
1	$\emptyset$	$\{x = null\}$	[iconst -1, ireturn]
2	$\{x.a > 0\}$	$\{x \neq null\}$	[**aload x, getfield a, ireturn**]
3	$\{x.a \leq 0\}$	$\{x \neq null\}$	[aload x, getfield b, ireturn]

Composition Strategy

Composition Strategy



Composition Strategy

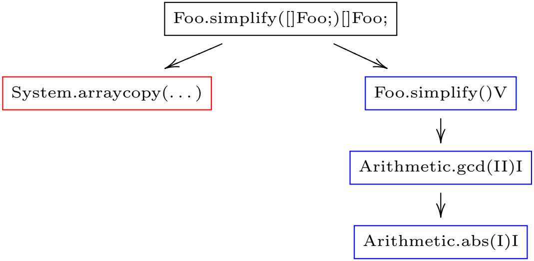


Context-sensitive

Pros. Only required information is computed

Cons. Reusability of summaries is not always possible

Composition Strategy



Context-sensitive

Pros. Only required information is computed

Cons. Reusability of summaries is not always possible

Context-insensitive

Pros. Composition can always be performed

Cons. Summaries can contain more cases than necessary (more expensive)

Composition Algorithm

```
procedure composeSummary(m,mode)
    if mode = duringSP then
        S ← getSummary(m)
        for all case ∈ S do
            setCompositionSchedule(case.getCompSched())
            composeCase(case)
        end for
    else
        S ← getSummary(m)
        caseIndex ← compositionSchedule.getNext()
        case ← getSummaryCase(S,caseIndex)
        composeCase(case)
    end if
end procedure
```

Composition Algorithm

```
procedure composeCase(case)
    heapPC ← case.getHeapPC()
    projectActualParameters(heapPC)
    if checkAndSet(currentHeapPC,heapPC) then
        pc ← case.getPC()
        projectActualParameters(pc)
        currentPC ← currentPC ∪ pc
        if satisfy(currentPC) then
            ReplaceCode(invokedMethod,case.getCode())
            ContinueSymbolicExecution          ▷ mode ≠ duringSP
        else
            Backtrack
        end if
    else
        Backtrack
    end if
end procedure
```

```
int abs(int a){
  if (a >= 0) return a;
  else return -a;
}
int q(Foo x){
  if (x != null && x.next != null
      && x.next.next != null
      && x.next.next.f != 0)
    return abs(x.next.f);
  else
   ...
}
```

```
void m(Foo x,Foo y, Foo z){
  Foo[] arr = new Foo[]{x,y,z};
  for (int i=0;i<arr.length;i++){
    if (arr[i] != null)
      arr[i].f = q(arr[i]);
    else
      ...
  }
}
```

Case	PC	HeapPC	Code	Sched
		Method `abs`		
0	$\{a \geq 0\}$	$\emptyset$	[`iload a,ireturn`]	[]
1	$\{a < 0\}$	$\emptyset$	[`iload a,ineg,ireturn`]	[]
		Method `q`		
...				
6	$\{x.f \geq 0, x.f \neq 0\}$	$\{x \neq null, x.next = x\}$	[`aload x,getfield next,getfield next, getfield f,invoke abs,ireturn`]	[0]
...				
		Method `m`		
...				
22	$\{z.f \geq 0, z.f \neq 0\}$	$\{x = null, y = null$ $z \neq null, z.next = z\}$	[`...,invoke q,...`]	[6, 0]
...				

Implementation

- Specialization Listener
  - Slice code for conditional instructions (`ifle`,`if_icmpeq`,`ifnull`,...)
  - Invoke instructions: Update specialized code, compose summary
  - Return instructions: Update specialized code and store summary case
  - Ignore `goto` instructions
  - For the remaining instructions, append instruction to specialized code
- Compositional Listener
  - Execute composition algorithm
- Other new classes: `MethodSummary`, `MethodSummaryCase`, `SpecializedCode`, `BindingMap`, `CompositionSchedule`, `NewSummaryChoiceGenerator`, `CompositionChoiceGenerator`
- Optimized conditional bytecode instructions

Experience

Example featuring linear integer constraints

Java source code

```
public static int abs(int x){
  if (x >= 0) return x;
  else return -x;
}
public static int gcd(int x, int y) {
  if (x == 0) return abs(y);
  while ((y != 0)  && (i<2)) {
    if (x > y) x = x-y;
    else y = y-x;
    if (i==2) return -1;
    i++;
  }
  return abs(x);
}
public class R{
  private int num, den;
  public void simplify(int a, int b){
    int gcd = gcd(a,b);
    if (gcd != 0) {
      num = num/gcd; den = den/gcd;
    }
  }
  public static R[] simp(R[] rs){
    R[] oldRs = new R[rs.length];
    arraycopy(rs,oldRs,length);
    for (int i = 0;i < length;i++)
      rs[i].simplify(rs[i].num,rs[i].den);
    return oldRs;
  }
}
```

Preliminary Experimental Results

Number of summary cases

Method `abs`: 2
Method `gcd`: 13
Method `simplify`: 14
Method `simp`: 2744

SPF vs. Compositional SPF

	SPF	**CompSPF**
Time	00:02:50	00:01:02
States	24899	13928
Choice Generators	12449	5689
Instructions	145908	139992
Max. Memory	106MB	170MB

Experience

Example featuring input data structures to stress lazy initialization

```
public int q(Foo x, Foo y){
  if (x != null) {
    if ((x.next != null) &&
        (x.next.next != null) &&
        (x.next.next.next != null) &&
        (x.next.next.next.f == 0))
      return -1;
    else
      return 0;
  } else if ((y != null) &&
             (y.next != null) &&
             (y.next.f == 0))
           return 1;
         else
           return 2;
}
public void m(Foo x,Foo y, Foo z){
  Foo[] arr = new Foo[]{x,y,z};
  for (int i=0; i < arr.length; i++) {
    if (arr[i] != null)
      arr[i].f = q(arr[i],y);
    else
      arr[i] = new Foo(0,0);
  }
}
```

Preliminary Experimental Results

Number of summary cases

Method `q`: 22
Method `m`: 9938

SPF vs. Compositional SPF

	SPF	CompSPF
Time	00:00:51	00:00:13
States	86175	27762
Choice Generators	29550	1215
Instructions	1215959	223786
Max. Memory	242MB	364MB

Related Work

Compositional symbolic execution

- Compositional dynamic test generation [Godefroid, POPL'07]
- Demand-driven compositional symbolic execution [Anand et al., TACAS'08]
- Compositional test case generation in CLP [Albert et al., LOPSTR'10]
- Theoretical aspects of compositional symbolic execution [Vanoverberghe et al., FASE'11]

Symbolic execution and program specialization

- Software specialization via symbolic execution [Coen-Porisini et al., IEEE TSE'91]
- Interleaving symbolic execution and partial evaluation [Bubel et al., FMCO'10]

Conclusions and Future Work

- Compositional reasoning based on partial evaluation
  - alleviate scalability problems in Symbolic Execution for Software Testing
- Implementation in SPF
- Practical issues:
  - Validate and optimize implementation
  - Full integration in SPF
  - Experimental evaluation
- Optimization
  - Constraints simplification
  - Save sequence instruction indexes in the specialized code
- Proofs of correctness
- Multi-threaded Java programs
- Focus on error detection

Thank you!