Structural Test Input Generation for 3-Address Code Coverage Using Path-Merged Symbolic Execution

Soha Hussein, Stephen McCamant, Elena Sherman, Vaibhav Sharma, Mike Whalen

University of Minnesota
soha@umn.edu

University of Minnesota
mccamant@cs.umn.edu

Boise State University
elenasherman@boisestate.edu

University of Minnesota
vaibhav@umn.edu

University of Minnesota
mwwhalen@umn.edu

AST 2023

Motivation: Symbolic Execution vs. Path-Merging

- **Symbolic Execution (SE)** is a precise path-sensitive technique that creates a conjunctive constraint for each feasible path.

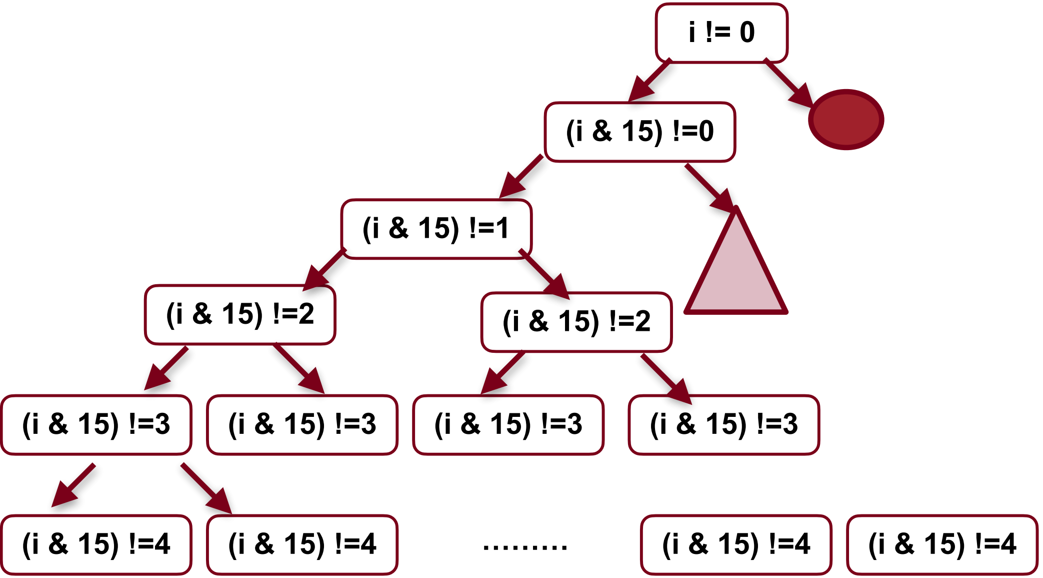
- **Path-merging (PM-SE)** reduces dynamically explored paths, by representing regions of code as a disjunctive constraint.

Motivation: example

- Counts the number of bit that are set in an integer.
- It checks every rightmost 4 bits.
- It increments the counter, if the first, second, third, or fourth bits are set.

```
public int getSetBits(int i) {
   int numOfSetBits = 0;
   while (i != 0) {
     if ((i & 15) != 0)
       numOfSetBits += count4Bits(i);
     i = (i >>> 4);    }
   return numOfSetBits;}

 public int count4Bits(int i) {
   int count = 0;
   if ((i & 1) == 1)
      count++;
   if ((i & 2) == 2)
     count++;
   if ((i & 4) == 4)
     count++;
   if ((i & 8) == 8)
     count++;
    return count;  }
```

Motivation: example



```
public int getSetBits(int i) {
   int numOfSetBits = 0;
   while (i != 0) {
     if ((i & 15) != 0)
       numOfSetBits += count4Bits(i);
     i = (i >>> 4);     }
   return numOfSetBits;}

 public int count4Bits(int i) {
   int count = 0;
   if ((i & 1) == 1)
      count++;
   if ((i & 2) == 2)
     count++;
   if ((i & 4) == 4)
     count++;
   if ((i & 8) == 8)
     count++;
    return count;   }
```

Explored paths with SE are exponential in the number of bits.

Motivation: example

i != 0

$$
\begin{aligned}
& t1 := i \,\&\, 15 \wedge \\
& t2 := i \,\&\, 1 \wedge \\
& count1 := ite(t2 == 1, 1, 0) \wedge \\
& t3 := i \,\&\, 2 \wedge \\
& count2 := count1 + 1 \wedge \\
& count3 := ite(t3 == 2, count2, count1) \wedge \\
& t4 := i \,\&\, 4 \wedge \\
& count4 := count3 + 1 \wedge \\
& count5 := ite(t4 == 4, count4, count3) \wedge \\
& t5 := i \,\&\, 8 \wedge \\
& count6 := count5 + 1 \quad \wedge \\
& count7 := ite(t5 == 8, count6, count5) \wedge \\
& numOfSetBits1 := 0 + count7 \wedge \\
& numOfSetBits2 := ite(t1 == 0, numOfSetBits1, 0)
\end{aligned}
$$

Explored paths with PM-SE is: 9 paths

```
public int getSetBits(int i) {
   int numOfSetBits = 0;
   while (i != 0) {
     if ((i & 15) != 0)
       numOfSetBits += count4Bits(i);
     i = (i >>> 4);     }
   return numOfSetBits;}

 public int count4Bits(int i) {
   int count = 0;
   if ((i & 1) == 1)
      count++;
   if ((i & 2) == 2)
     count++;
   if ((i & 4) == 4)
     count++;
   if ((i & 8) == 8)
     count++;
    return count;   }
```

But, what about generating test input suite that is branch adequate?

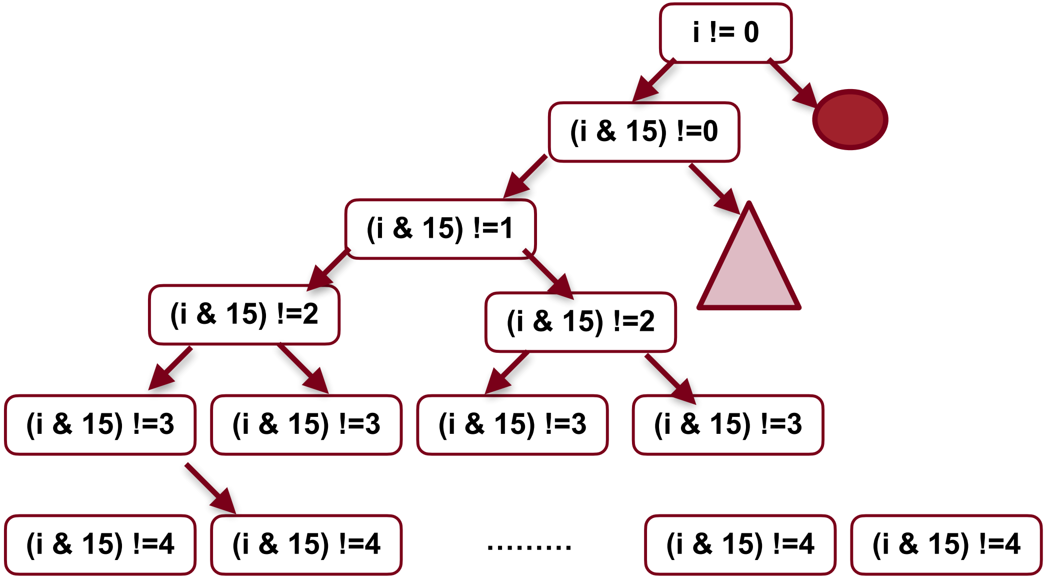
Branch Coverage

- *Branch coverage* is a type of structural coverage criteria.
- In branch adequate, there are two coverage targets: *Taken (TK)* and Not *Taken (NT)* sides, i.e., *true* and *false* sides.
- An *obligation*, is any target we want to cover, i.e., execute.

```
public int getSetBits(int i) {
   int numOfSetBits = 0;
   while (i != 0) {
     if ((i & 15) != 0)
       numOfSetBits += count4Bits(i);
     i = (i >>> 4);    }
   return numOfSetBits;}

 public int count4Bits(int i) {
   int count = 0;
   if ((i & 1) == 1)
      count++;
   if ((i & 2) == 2)
     count++;
   if ((i & 4) == 4)
     count++;
   if ((i & 8) == 8)
     count++;
    return count;  }
```

SE Branch Adequate Test-Input Generation

- Test inputs are one of the main applications of SE.
- Path adequate test-input suite is collected at the end of every execution path.
- Branch adequate can be collected when an a side of a branch is executed for the first time.



PM-SE Branch Adequate Test-Input?

i != 0

$$
\begin{aligned}
&t1 := i \,\&\, 15 \wedge \\
&t2 := i \,\&\, 1 \wedge \\
&count1 := ite(t2 == 1,1,0) \wedge \\
&t3 := i \,\&\, 2 \wedge \\
&count2 := count1 + 1 \wedge \\
&count3 := ite(t3 == 2, count2, count1) \wedge \\
&t4 := i \,\&\, 4 \wedge \\
&count4 := count3 + 1 \wedge \\
&count5 := ite(t4 == 4, count4, count3) \wedge \\
&t5 := i \,\&\, 8 \wedge \\
&count6 := count5 + 1 \quad \wedge \\
&count7 := ite(t5 == 8, count6, count5) \wedge \\
&numOfSetBits1 := 0 + count7 \wedge \\
&numOfSetBits2 := ite(t1 == 0, numOfSetBits1, 0)
\end{aligned}
$$

Problem: summarization has no structure, or links to the original branch targets

```java
public int getSetBits(int i) {
   int numOfSetBits = 0;
   while (i != 0) {
     if ((i & 15) != 0)
       numOfSetBits += count4Bits(i);
     i = (i >>> 4);     }
   return numOfSetBits;}

 public int count4Bits(int i) {
   int count = 0;
   if ((i & 1) == 1)
      count++;
   if ((i & 2) == 2)
     count++;
   if ((i & 4) == 4)
     count++;
   if ((i & 8) == 8)
     count++;
    return count;  }
```

The goal of this paper

- Generate branch adequate test-inputs suite for PM-SE without compromising its benefits.

- PM-SE’s benefits could be compromised if:
  - we make too many solver calls.
  - we make too complex summary that is difficult to solve.

Technique

1. We intercept code regions that PM is about to merge.
2. Annotate summarization in 3 main steps:
    a. Expanding conditions.
    b. Marking obligations using *obligation variables*.
    c. Creating conditional assignments to obligation variables.
3. At the end of an execution path, we collect satisfiable new coverage using obligation variables.

Marking Obligations and Creating GSA

- We augment the PM-SE IR with assignments to obligation variables each obligation.

```
t1 := (i & 15 )
if (! ( t1 == 0 )) {
   oblg_10_TK1 := 1
   t2 := (i & 1 )
   if ( t2 == 1 ) {
      oblg_6_TK1 := 1
      count2 :=  0 + 1
   } else {
      oblg_6_NT1 := 1
   }


...}

```

Marking Obligations and Creating GSA

- These conditions can only be true, if their corresponding branch targets are satisfied.

- We propagate obligation variables conditions using Gated Single Assignment (GSA).

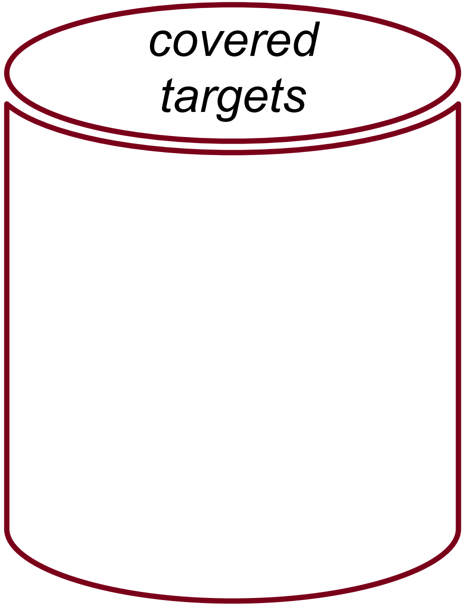
```
t1 := (i & 15 )
if (! ( t1 == 0 )) {
  oblg_10_TK1 := 1
  t2 := (i & 1 )
  if ( t2 == 1 ) {
    oblg_6_TK1 := 1
    count2 :=  0 + 1
  } else {
    oblg_6_NT1 := 1
  }
  oblg_6_TK2 := ite(t2==1, oblg_6_TK1, 0)
  oblg_6_NT2 := ite(t2==1, 0, oblg_6_NT1)
...}

```

Collecting Test-Inputs

- In PM-SE multiple coverage targets can be satisfied at the end of an execution path.
- Thus we repeat until no new coverage targets are found.
- We conjoin the path condition with uncovered branch targets within the execution of the path.
- For those satisfied:
    1. We collect the Input values from the solver model as a test-input for the newly satisfied targets.
    2. We update our covered targets.

Step3: Collect Test-Inputs - Example

$$
\begin{gathered}
PathCondition \wedge \\
(oblg\_4\_TK \vee oblg\_4\_NT \vee \\
oblg\_10\_TK \vee oblg\_10\_NT \vee \\
oblg\_12\_TK \vee oblg\_12\_NT \vee \\
oblg\_14\_TK \vee oblg\_14\_NT \vee \\
oblg\_16\_TK \vee oblg\_16\_NT)
\end{gathered}
$$



Step3: Collect Test-Inputs - Example

$$PathCondition \wedge$$

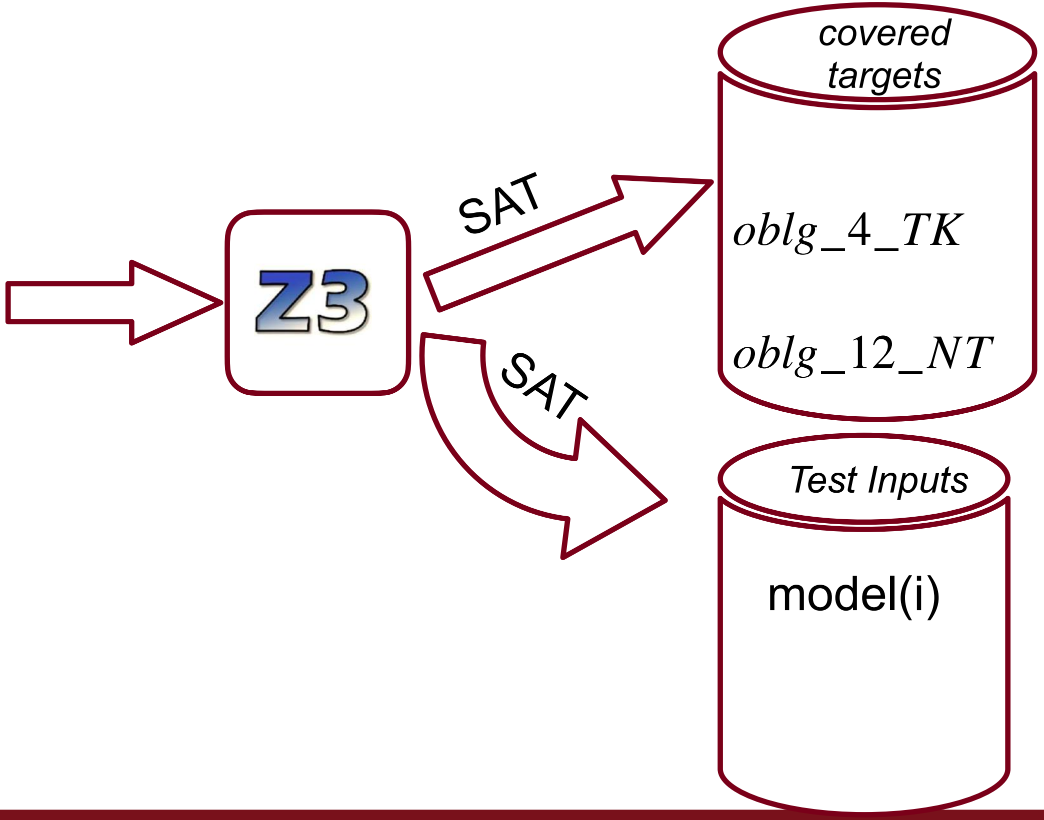
$$(oblg\_4\_TK \vee oblg\_4\_NT \vee$$

$$oblg\_10\_TK \vee oblg\_10\_NT \vee$$

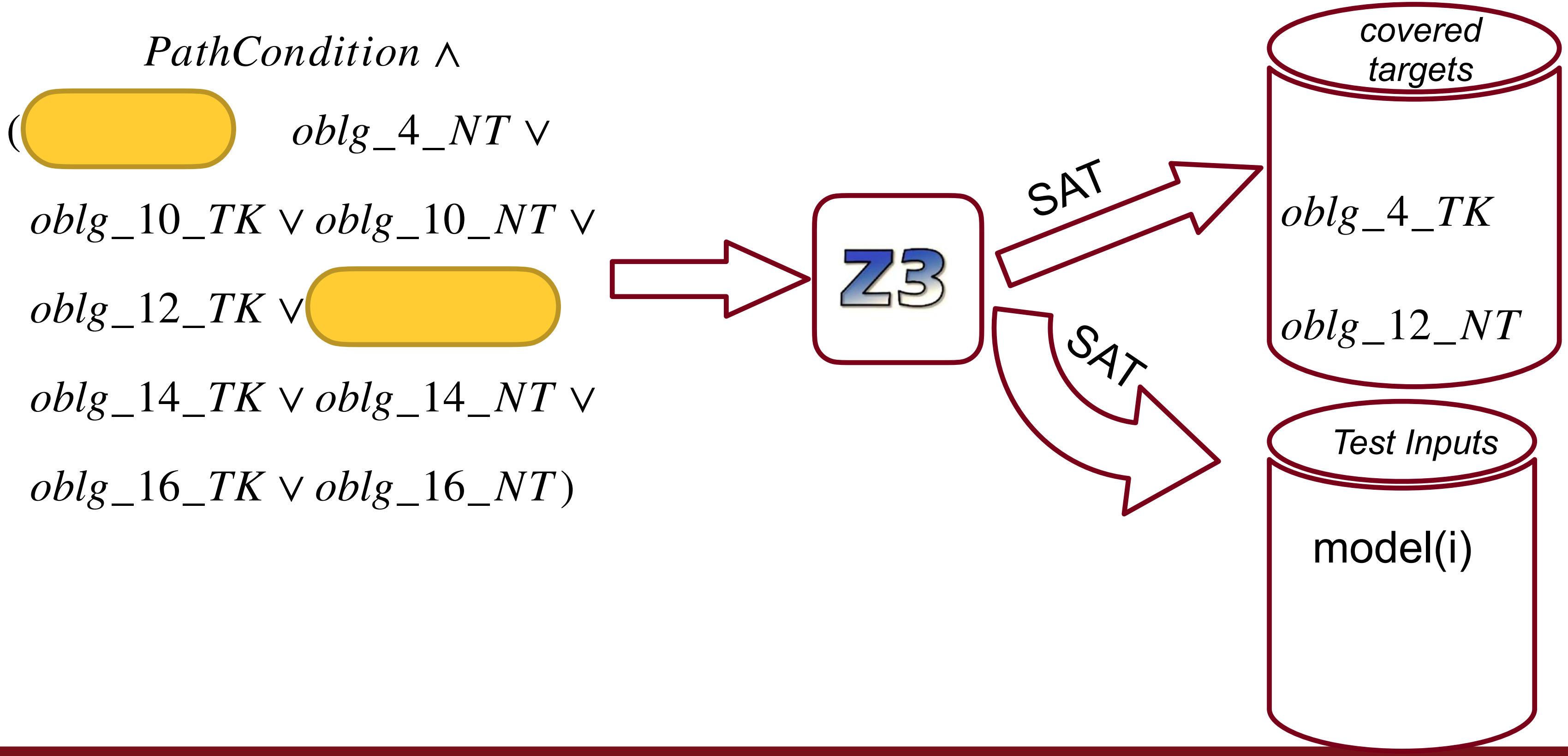
$$oblg\_12\_TK \vee oblg\_12\_NT \vee$$

$$oblg\_14\_TK \vee oblg\_14\_NT \vee$$

$$oblg\_16\_TK \vee oblg\_16\_NT)$$



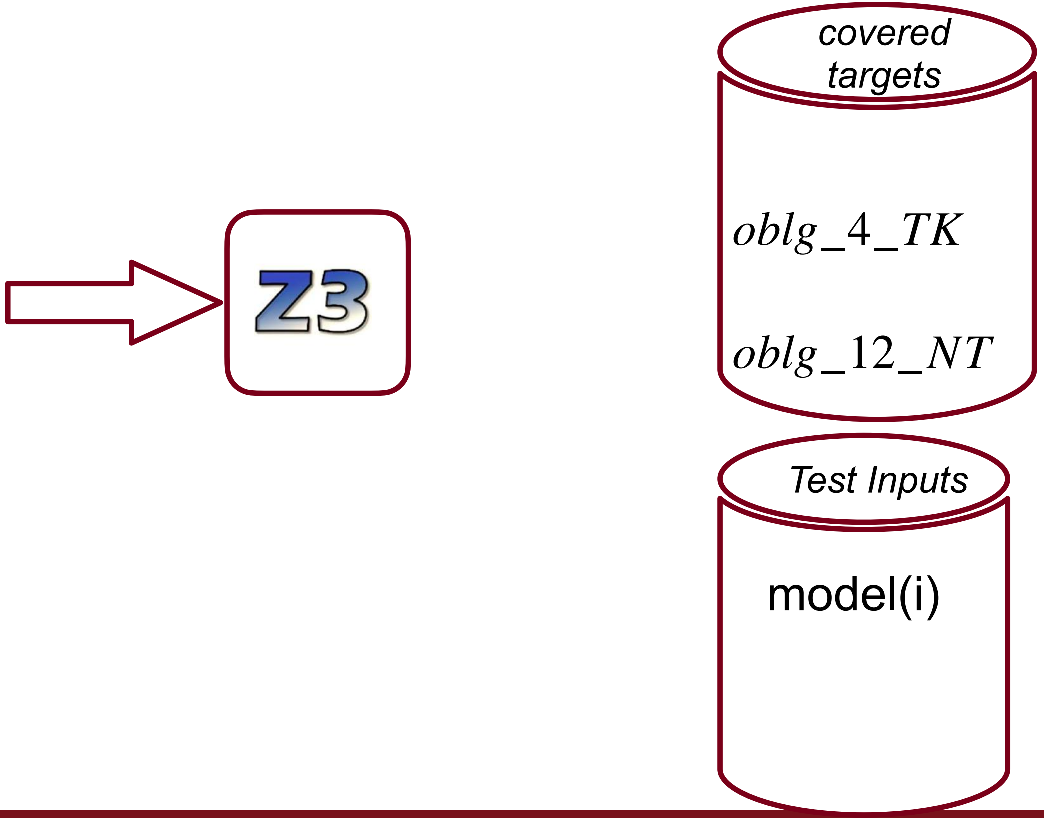
Step3: Collect Test-Inputs - Example



$$
\begin{aligned}
&PathCondition \land \\
&(\qquad oblg\_4\_NT \lor \\
&oblg\_10\_TK \lor oblg\_10\_NT \lor \\
&oblg\_12\_TK \lor \\
&oblg\_14\_TK \lor oblg\_14\_NT \lor \\
&oblg\_16\_TK \lor oblg\_16\_NT)
\end{aligned}
$$

Step3: Collect Test-Inputs - Example

$$
\begin{aligned}
&PathCondition \wedge \\
&(\quad oblg\_4\_NT \vee \\
&oblg\_10\_TK \vee oblg\_10\_NT \vee \\
&oblg\_12\_TK \vee \\
&oblg\_14\_TK \vee oblg\_14\_NT \vee \\
&oblg\_16\_TK \vee oblg\_16\_NT)
\end{aligned}
$$



covered targets

$oblg\_4\_TK$

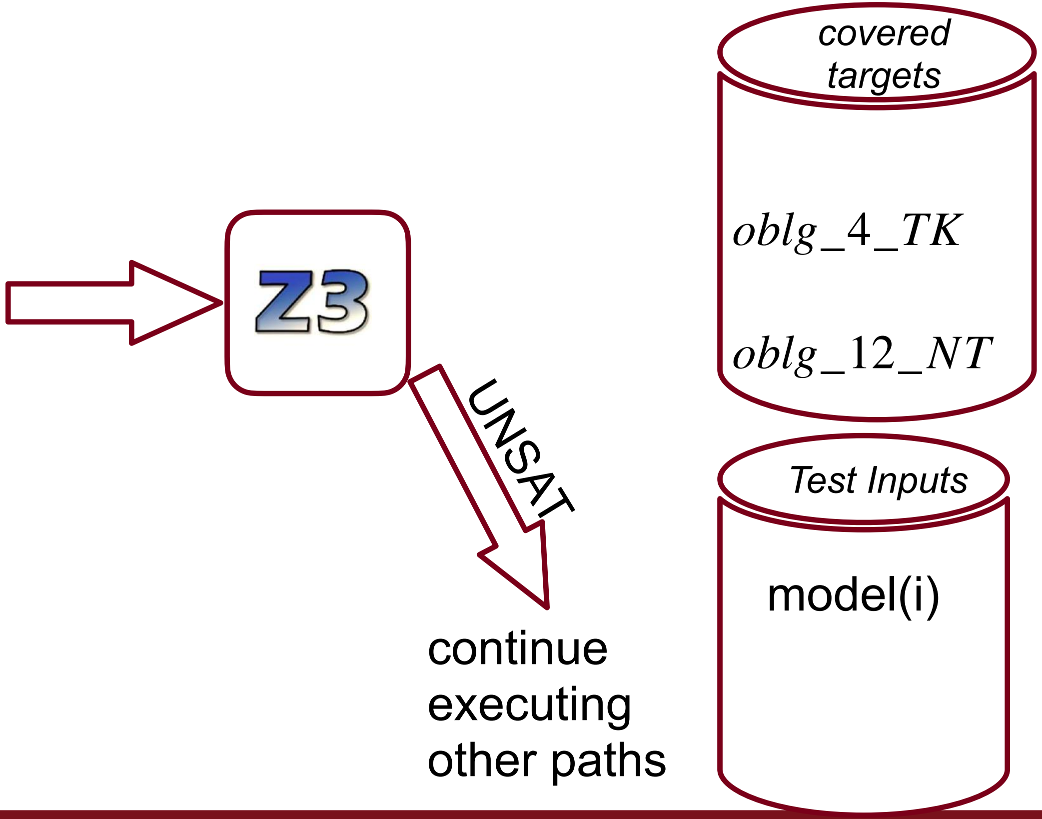
$oblg\_12\_NT$

Test Inputs

model(i)

Step3: Collect Test-Inputs - Example

$$\begin{gathered} PathCondition \wedge \\ (\qquad oblg\_4\_NT \vee \\ oblg\_10\_TK \vee oblg\_10\_NT \vee \\ oblg\_12\_TK \vee \\ oblg\_14\_TK \vee oblg\_14\_NT \vee \\ oblg\_16\_TK \vee oblg\_16\_NT) \end{gathered}$$



continue executing other paths

covered targets

$oblg\_4\_TK$

$oblg\_12\_NT$

Test Inputs

model(i)

Evaluation

- Implemented our extension in Java Ranger, and compared it with SPF.

- **RQ1:** Compared to SPF, what is the JR's s run-time overhead for generating branch adequate test inputs?
- **RQ2:** What is the effectiveness of the coverage over time for each technique?
- **RQ3:** Are SE and PM-SE complementary to one another?

Benchmark	SLOC	# classes	#methods
ApacheCLI	3612	18	183
NanoXML	4610	17	129
TCAS	300	1	12
WBS	265	1	3
Schedule	306	4	27
Siena	1256	10	94
PrintTokens	570	4	30
replace	795	1	19

RQ1: Compared to SPF, what is the JR’s s run-time overhead for generating branch adequate test inputs?

- We configured each benchmark with the maximum number of symbolic inputs such that all runs terminate in less than an hour.

- We confirmed that they covering the same obligations.

RQ1: Compared to SPF, what is the JR’s s run-time overhead for generating branch adequate test inputs?

benchmark	analysis	merges	queries	tests	avg query	time
ApachiCLI	SPF	-	139262	8	0.012	1737.9
	JR	7296	4286	8	0.091	412.2
NanoXML	SPF	-	91554	46	0.014	1391.6
	JR	1602	19060	44	0.021	428
WBS	SPF	-	27646	12	0.012	349.9
	JR	35	7	6	1.276	11.7
TCAS	SPF	-	1256	21	0.012	17.4
	JR	4	17	16	0.761	15.6
Schedule	SPF	-	33612	7	0.012	415.9
	JR	0	33612	7	0.012	427.8
Siena	SPF	-	52780	6	0.012	698.9
	JR	0	52780	6	0.012	703.3
PrintTokens	SPF	-	10616	43	0.013	141.7
	JR	451	9179	43	0.019	186.26
Replace	SPF	-	11304	27	0.011	131.8
	JR	252	35136	36	0.066	2358.7

RQ1: Compared to SPF, what is the JR’s s run-time overhead for generating branch adequate test inputs?

benchmark	analysis	merges	queries	tests	avg query	time
ApachiCLI	SPF	-	139262	8	0.012	1737.9
	JR	7296	4286	8	0.091	412.2
NanoXML	SPF	-	91554	46	0.014	1391.6
	JR	1602	19060	44	0.021	428
WBS	SPF	-	27646	12	0.012	349.9
	JR	35	7	6	1.276	11.7
TCAS	SPF	-	1256	21	0.012	17.4
	JR	4	17	16	0.761	15.6
Schedule	SPF	-	33612	7	0.012	415.9
	JR	0	33612	7	0.012	427.8
Siena	SPF	-	52780	6	0.012	698.9
	JR	0	52780	6	0.012	703.3
PrintTokens	SPF	-	10616	43	0.013	141.7
	JR	451	9179	43	0.019	186.26
Replace	SPF	-	11304	27	0.011	131.8
	JR	252	35136	36	0.066	2358.7

RQ1: Compared to SPF, what is the JR’s s run-time overhead for generating branch adequate test inputs?

benchmark	analysis	merges	queries	tests	avg query	time
ApachiCLI	SPF	-	139262	8	0.012	1737.9
	JR	7296	4286	8	0.091	412.2
NanoXML	SPF	-	91554	46	0.014	1391.6
	JR	1602	19060	44	0.021	428
WBS	SPF	-	27646	12	0.012	349.9
	JR	35	7	6	1.276	11.7
TCAS	SPF	-	1256	21	0.012	17.4
	JR	4	17	16	0.761	15.6
Schedule	SPF	-	33612	7	0.012	415.9
	JR	0	33612	7	0.012	427.8
Siena	SPF	-	52780	6	0.012	698.9
	JR	0	52780	6	0.012	703.3
PrintTokens	SPF	-	10616	43	0.013	141.7
	JR	451	9179	43	0.019	186.26
Replace	SPF	-	11304	27	0.011	131.8
	JR	252	35136	36	0.066	2358.7

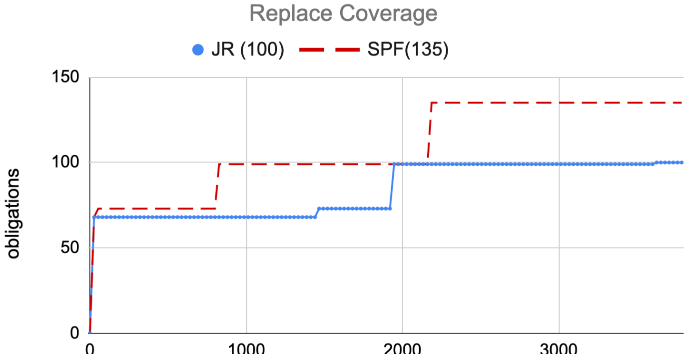
RQ1: Compared to SPF, what is the JR’s s run-time overhead for generating branch adequate test inputs?

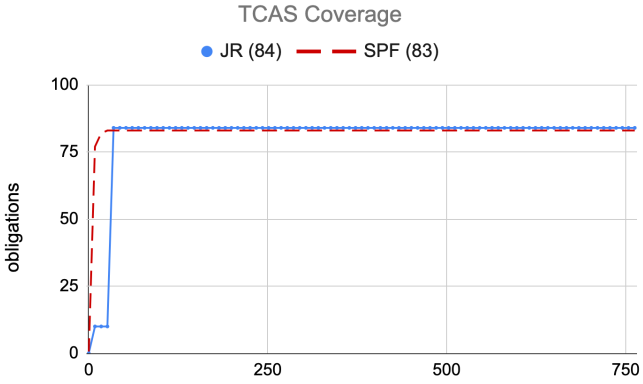
benchmark	analysis	merges	queries	tests	avg query	time
ApachiCLI	SPF	-	139262	8	0.012	1737.9
	JR	7296	4286	8	0.091	412.2
NanoXML	SPF	-	91554	46	0.014	1391.6
	JR	1602	19060	44	0.021	428
WBS	SPF	-	27646	12	0.012	349.9
	JR	35	7	6	1.276	11.7
TCAS	SPF	-	1256	21	0.012	17.4
	JR	4	17	16	0.761	15.6
Schedule	SPF	-	33612	7	0.012	415.9
	JR	0	33612	7	0.012	427.8
Siena	SPF	-	52780	6	0.012	698.9
	JR	0	52780	6	0.012	703.3
PrintTokens	SPF	-	10616	43	0.013	141.7
	JR	451	9179	43	0.019	186.26
Replace	SPF	-	11304	27	0.011	131.8
	JR	252	35136	36	0.066	2358.7

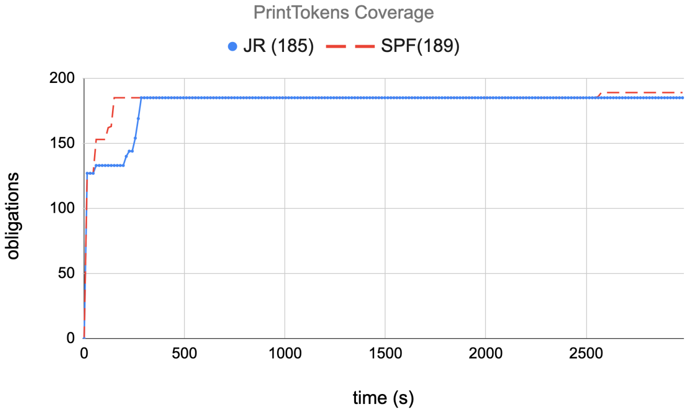
RQ1: Compared to SPF, what is the JR's s run-time overhead for generating branch adequate test inputs?

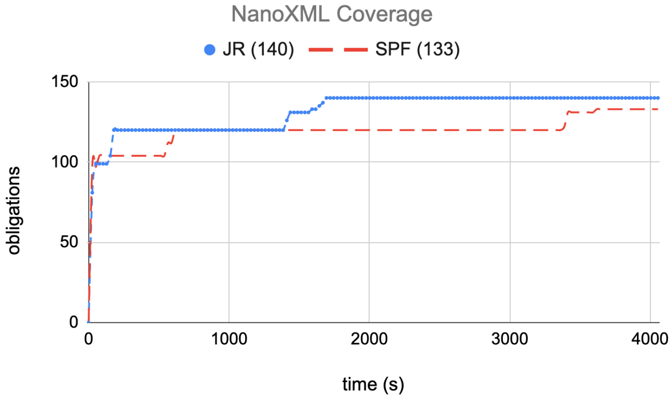
benchmark	analysis	merges	queries	tests	avg query	time
ApachiCLI	SPF	-	139262	8	0.012	1737.9
	JR	7296	4286	8	0.091	412.2
NanoXML	SPF	-	91554	46	0.014	1391.6
	JR	1602	19060	44	0.021	428
WBS	SPF	-	27646	12	0.012	349.9
	JR	35	7	6	1.276	11.7
TCAS	SPF	-	1256	21	0.012	17.4
	JR	4	17	16	0.761	15.6
Schedule	SPF	-	33612	7	0.012	415.9
	JR	0	33612	7	0.012	427.8
Siena	SPF	-	52780	6	0.012	698.9
	JR	0	52780	6	0.012	703.3
PrintTokens	SPF	-	10616	43	0.013	141.7
	JR	451	9179	43	0.019	186.26
Replace	SPF	-	11304	27	0.011	131.8
	JR	252	35136	36	0.066	2358.7

RQ2: What is the effectiveness of the coverage over time for each technique?


Replace Coverage


TCAS Coverage


PrintTokens Coverage


NanoXML Coverage

RQ3: Are SE and PM-SE complementary to one another?

- **Results:**
  - Both SPF and JR can reach complementary obligations.

bench	common	SPF extras	JR extras
ApacheCLI	89	0	0
NanoXML	133	0	7
TCAS	67	0	0
WBS	83	0	1
Schedule	61	0	0
Siena	39	0	0
PrintTokens	185	4	0
replace	99	36	1

RQ3: Are SE and PM-SE complementary to one another?

- **Results:**
  - Both SPF and JR can reach complementary obligations.
  - Notably, in replace despite the much worst performance (2358.7 ms vs. 131.8 ms), JR found an extra obligation.
  - Suggesting JR is a complementary technique for SPF.

bench	common	SPF extras	JR extras
ApacheCLI	89	0	0
NanoXML	133	0	7
TCAS	67	0	0
WBS	83	0	1
Schedule	61	0	0
Siena	39	0	0
PrintTokens	185	4	0
replace	99	36	1

Conclusion

- This paper showed how coverage and test inputs can be generated for path-merged symbolic execution.
- We applied this technique to create test inputs for branch coverage.
- We implemented the technique as an extension to Java Ranger.
- Experiments showed mixed complementary results, indicating the importance of both techniques
- In the future, we plan to investigate and create heuristics that use path merging when it can yield overall better performance.