4th International KLEE Workshop on Symbolic Execution

Symbolic Execution Oriented Constraint Solving

Zhenbang Chen

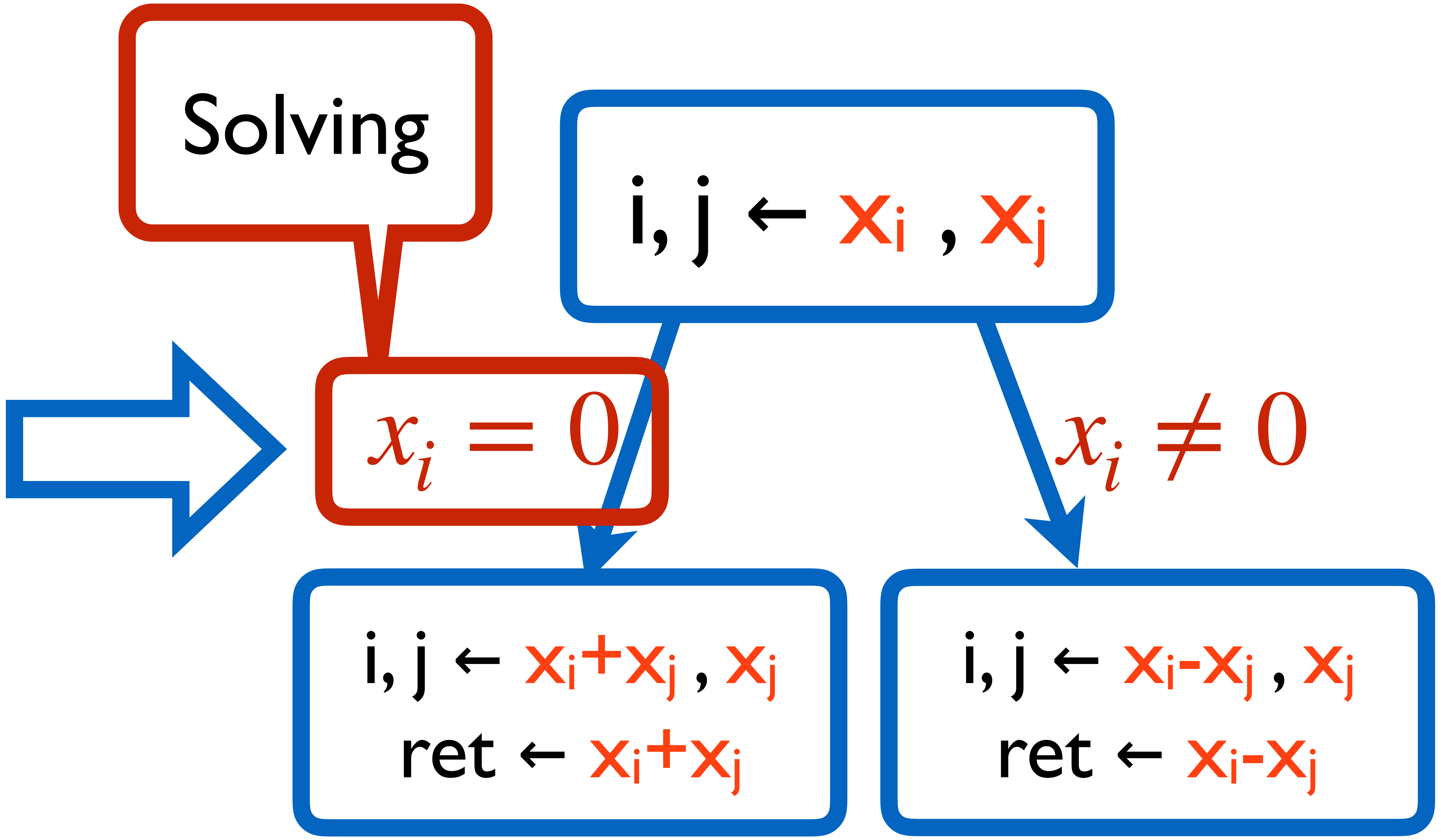
(zbchen@nudt.edu.cn)

Joint work with Ziqi Shuai, Yufeng Zhang, Zehua Chen, Guofeng Zhang, Jun Sun, Wei Dong and Ji Wang

2024.04.15

Symbolic Execution

```
int foo(int i, j) {
  if (i == 0) {
    i = i + j
  } else {
    i = i - j
  }
  return i
}
```

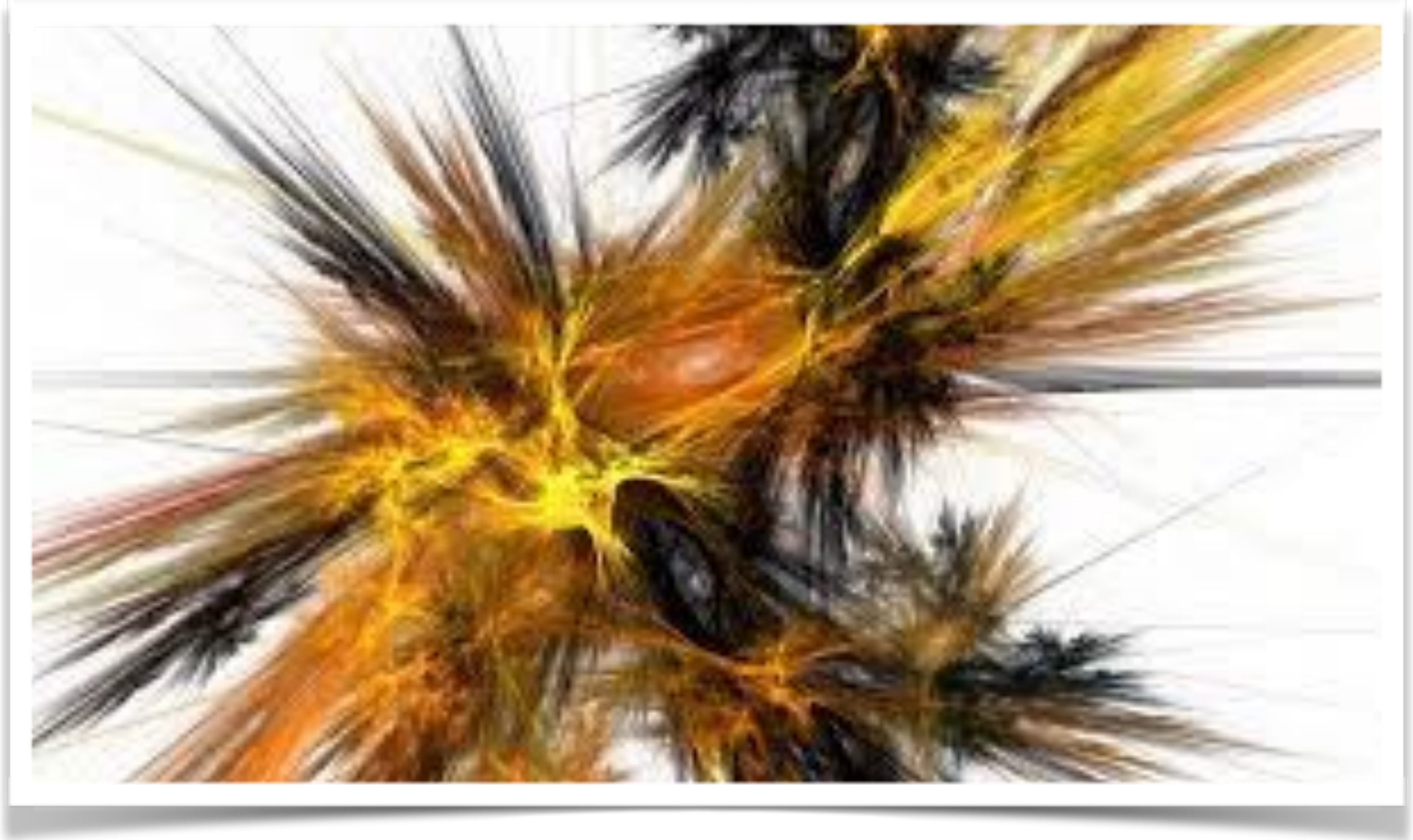


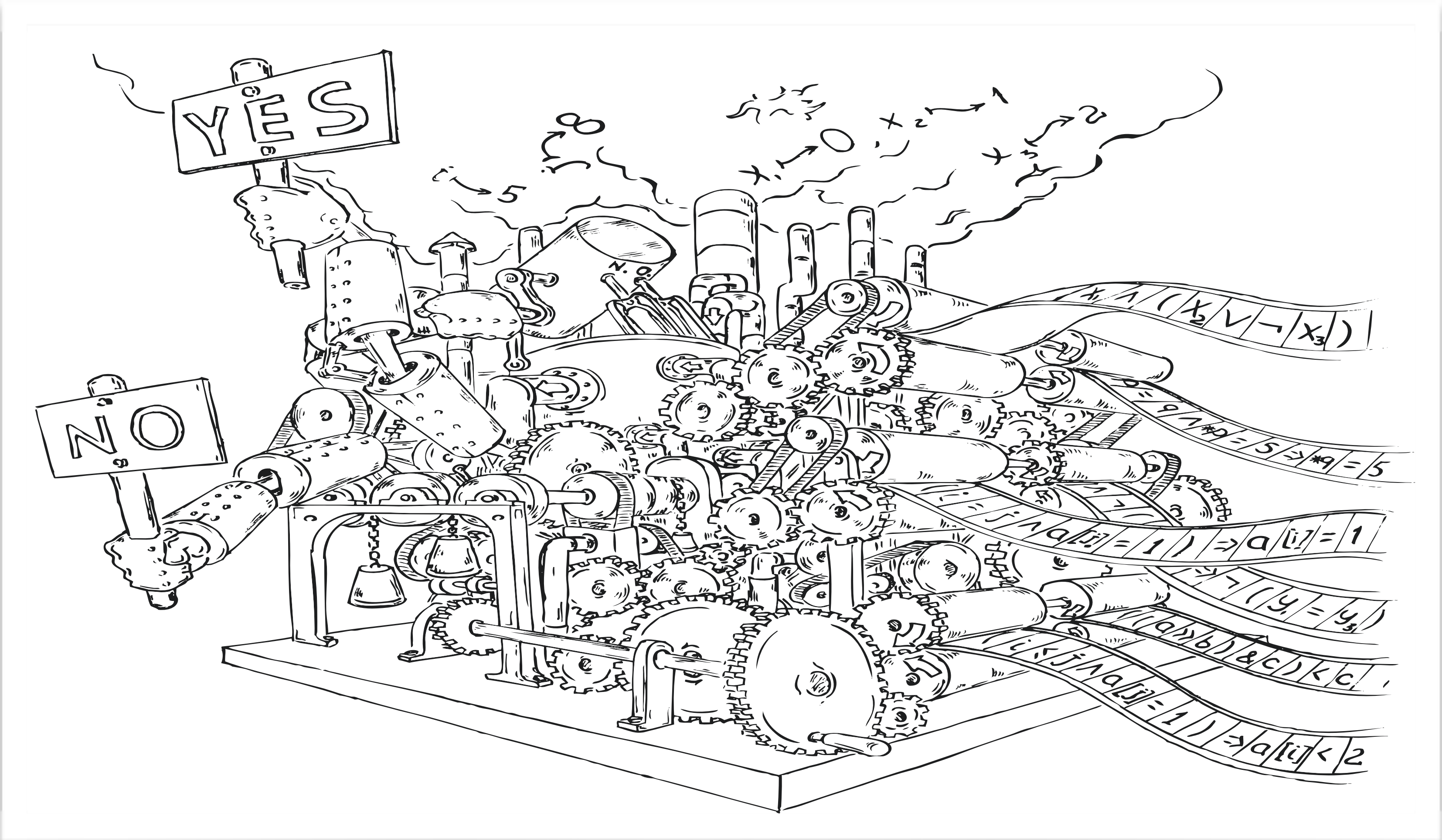
Constraint solving is the enabling technique

Challenges of Symbolic Execution

Path explosion

Constraint Solving



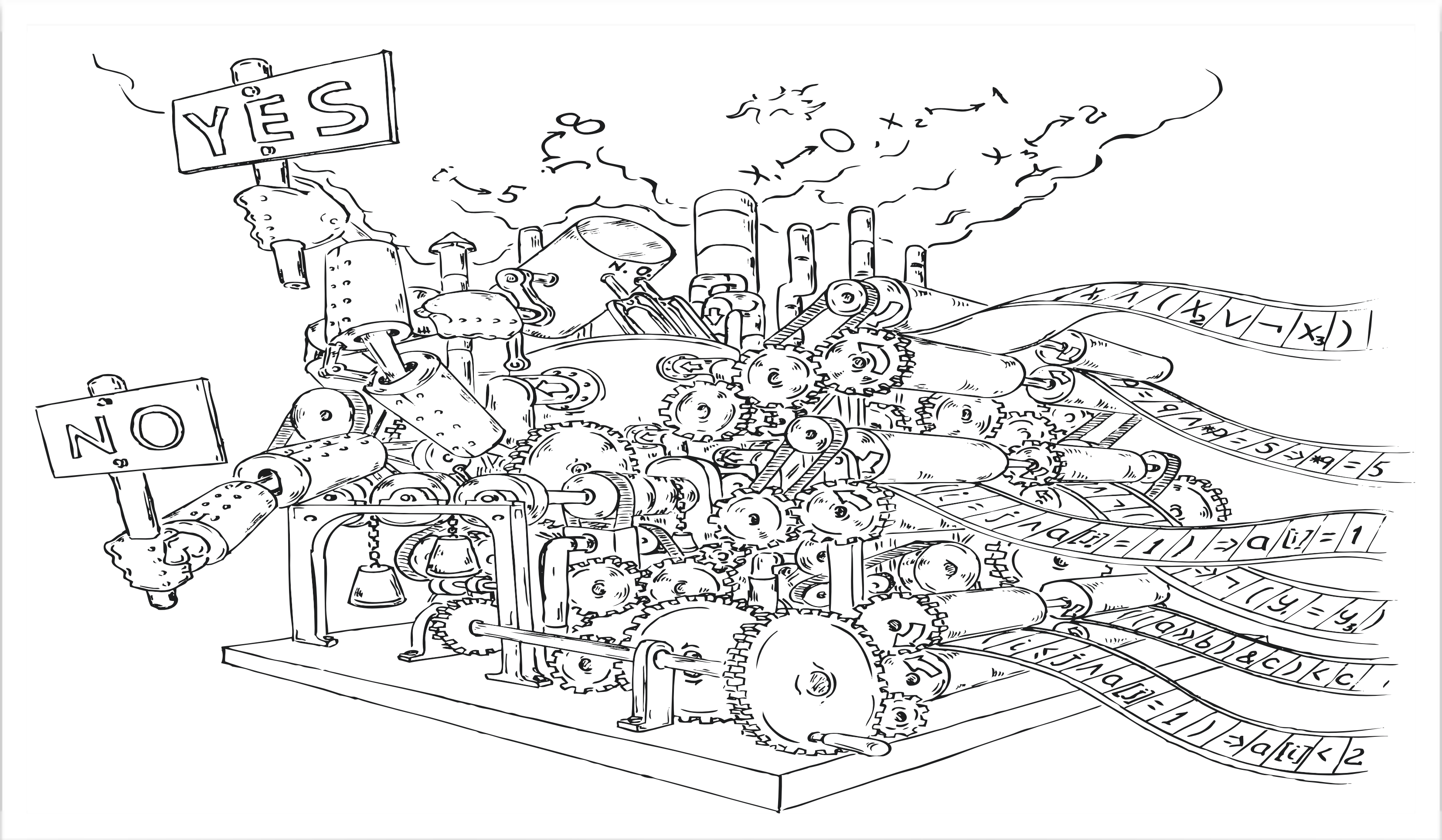


Decision Procedures An Algorithmic Point of View, Second Edition, 2016

This Talk's Target

Path explosion

Constraint Solving

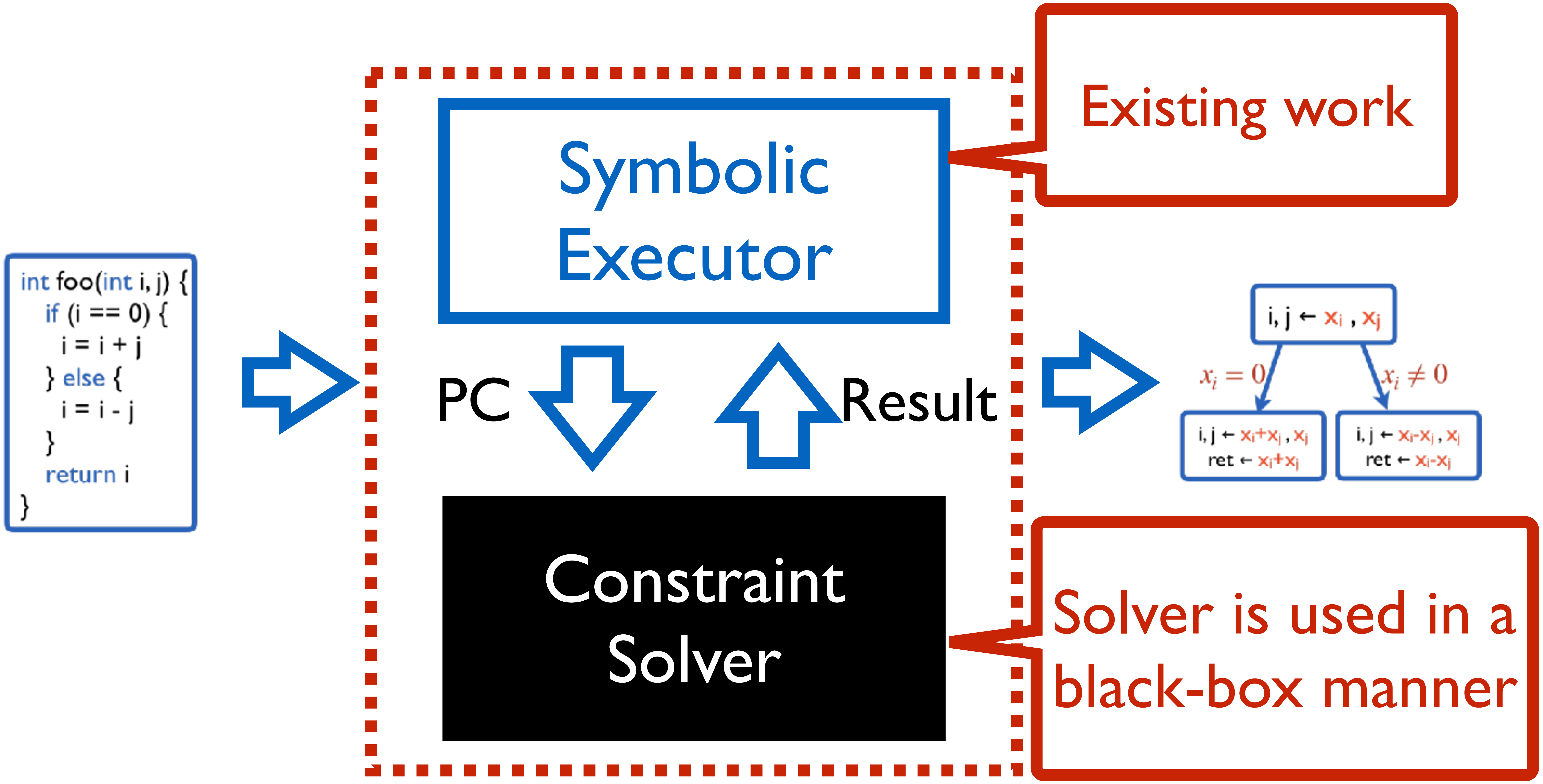


Decision Procedures An Algorithmic Point of View, Second Edition, 2016

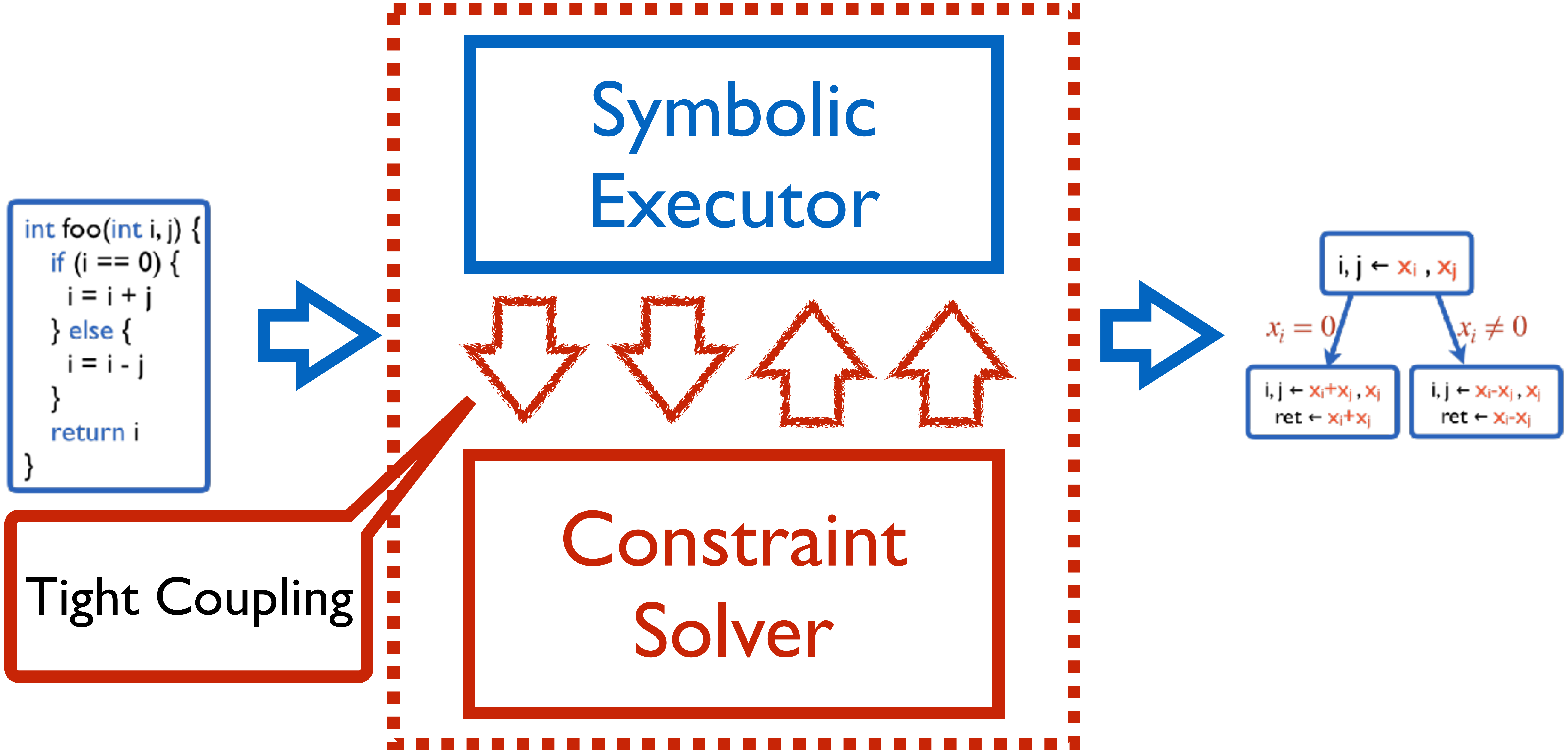
Existing Work of Optimizing Constraint Solving in SE

- Query cache (partial) and simplification
  - KLEE[OSDI'08], KLEE-Array[ISSTA'17]
- Query reduction
  - SSE[ISSRE'12], Cloud9[PLDI'12]
- Query reuse
  - Green[FSE'12], GreenTrie[ISSTA'15]

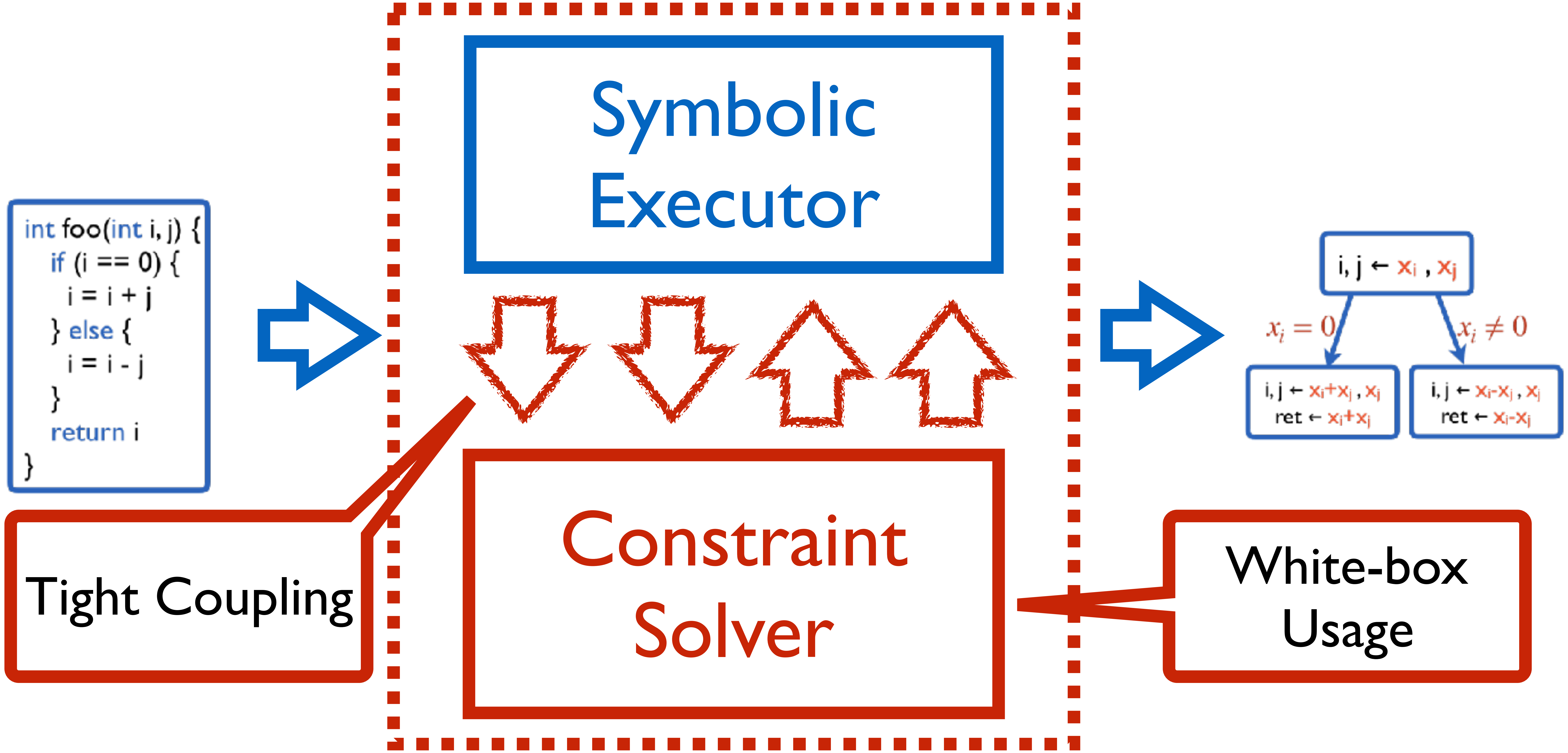
Our Observation



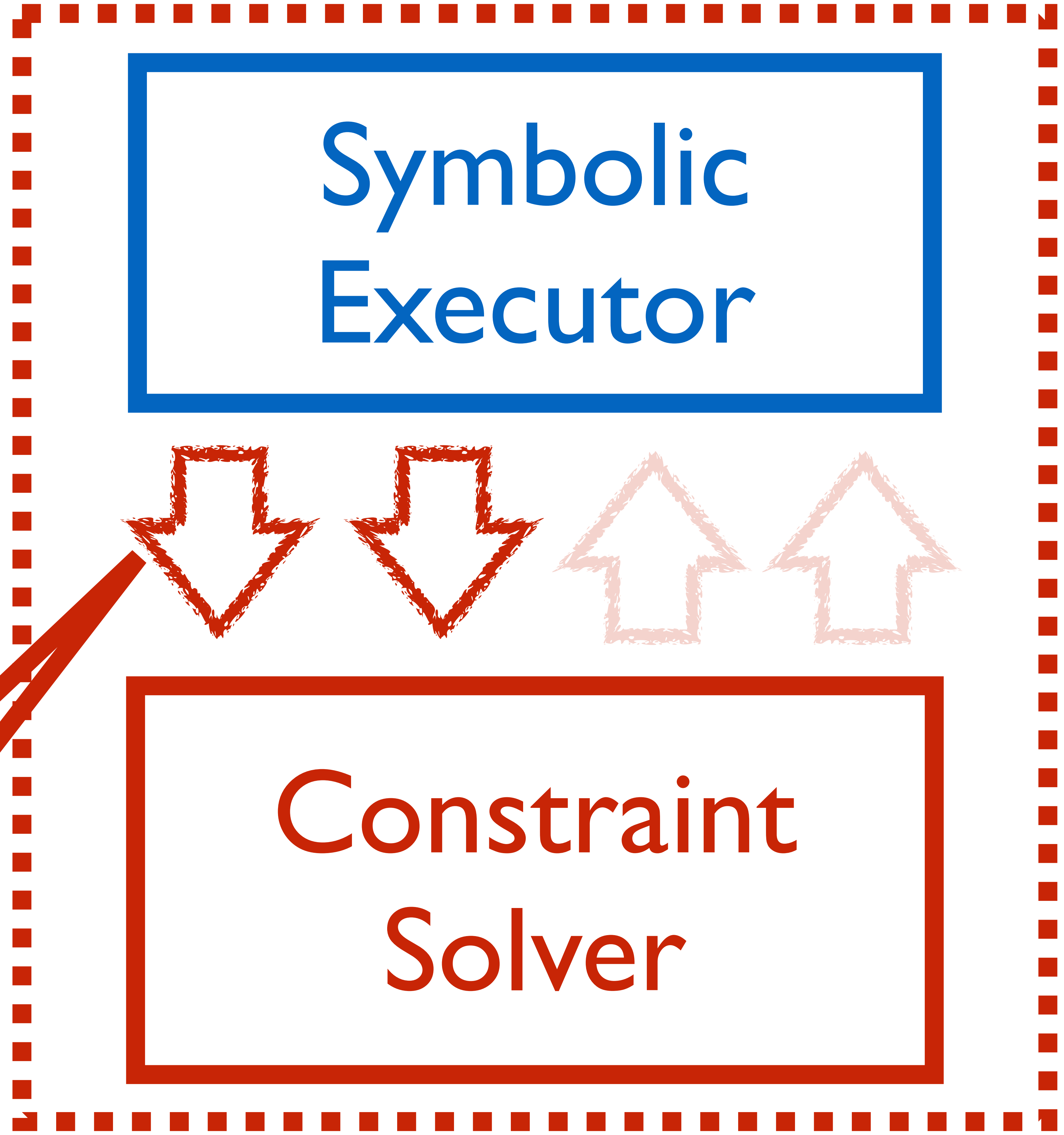
Our Argument



Our Argument

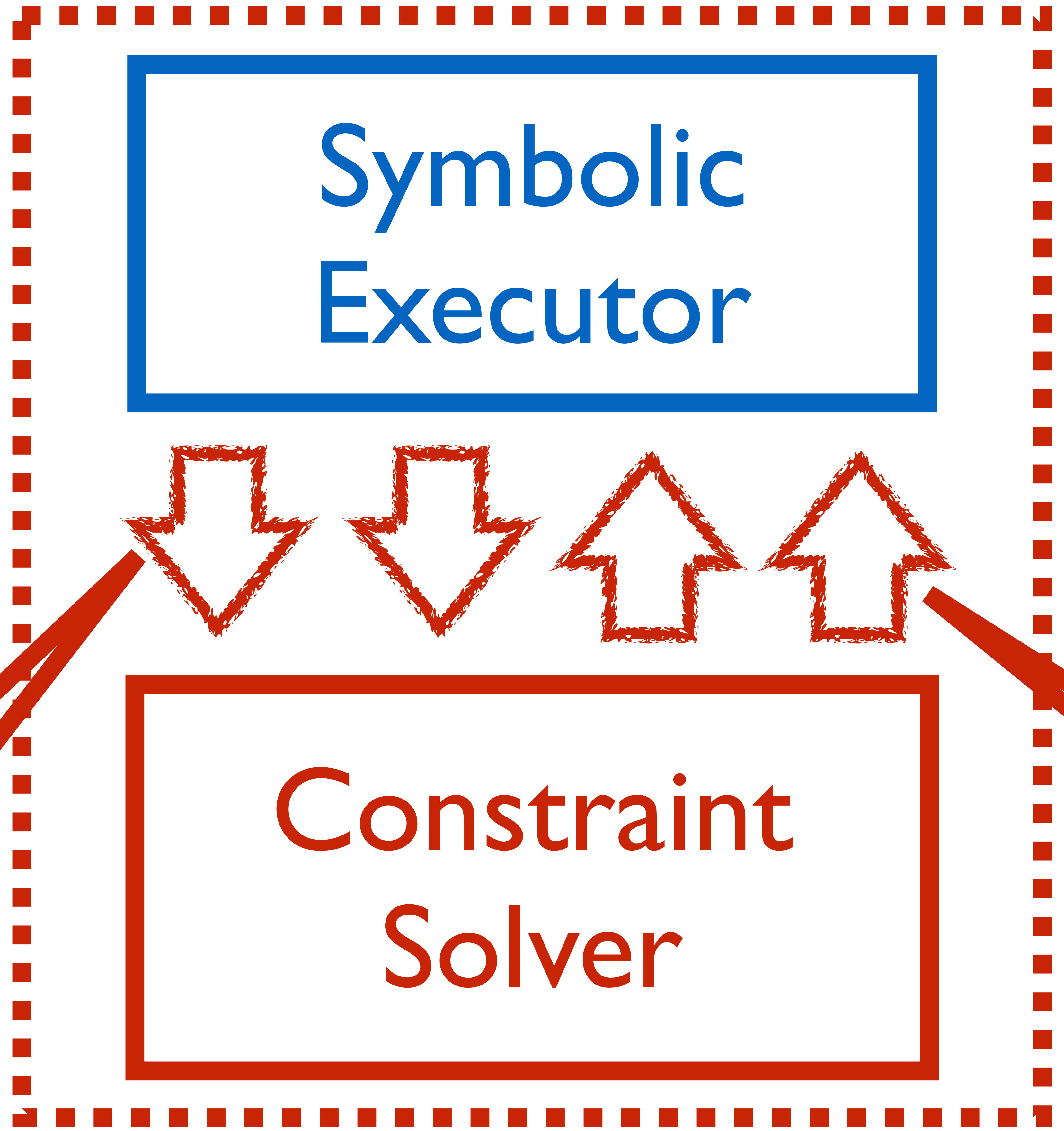


Our Recent Progress



- Type and Interval Aware Array Constraint Solving [ISSTA 2021]

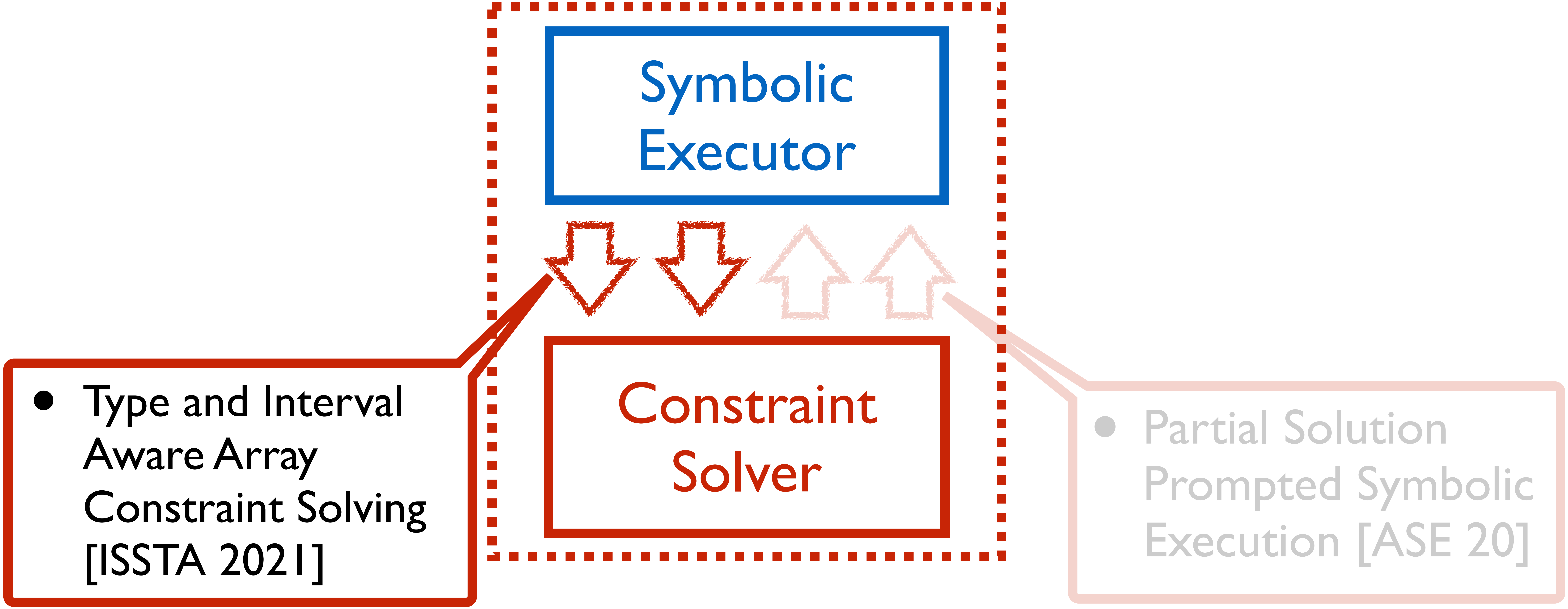
Our Recent Progress



- Type and Interval Aware Array Constraint Solving [ISSTA 2021]

- Partial Solution Prompted Symbolic Execution [ASE 20]

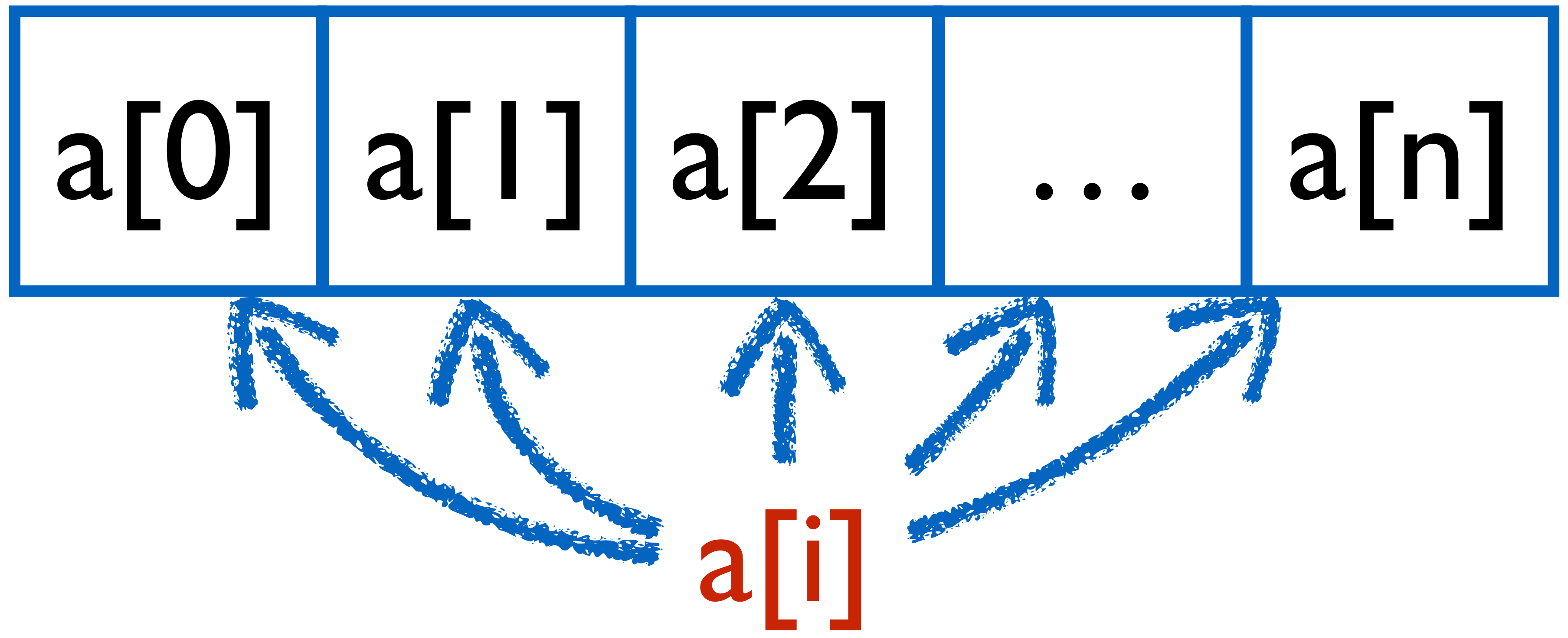
Our Recent Progress



Array Code Symbolic Execution

Arrays are ubiquitous in programs

The symbolic execution of array code is challenging



Array SMT Theory

Memory modeling in SE

- Byte-level memory reasoning in symbolic execution
  - QF_ABV SMT theory
  - KLEE、S2E、…

Memory modeling in SE

- Byte-level memory reasoning in symbolic execution
  - QF_ABV SMT theory
  - KLEE、S2E、…
- Every data is represented by a byte array
  - Many array variables in the path constraints
  - Large amount of axioms (O(n^2))

Problem

- Scalability of array constraint solving in symbolic execution
  - Byte-level array representation
  - Large number of axioms
  - …

Our Key Insights

- Many redundant axioms exist for byte array constraints
  - Array access type information
  - Array index constraint

Our Key Insights

- Many redundant axioms exist for byte array constraints
  - Array access type information
  - Array index constraint
- Unsatisfiability can be decided earlier

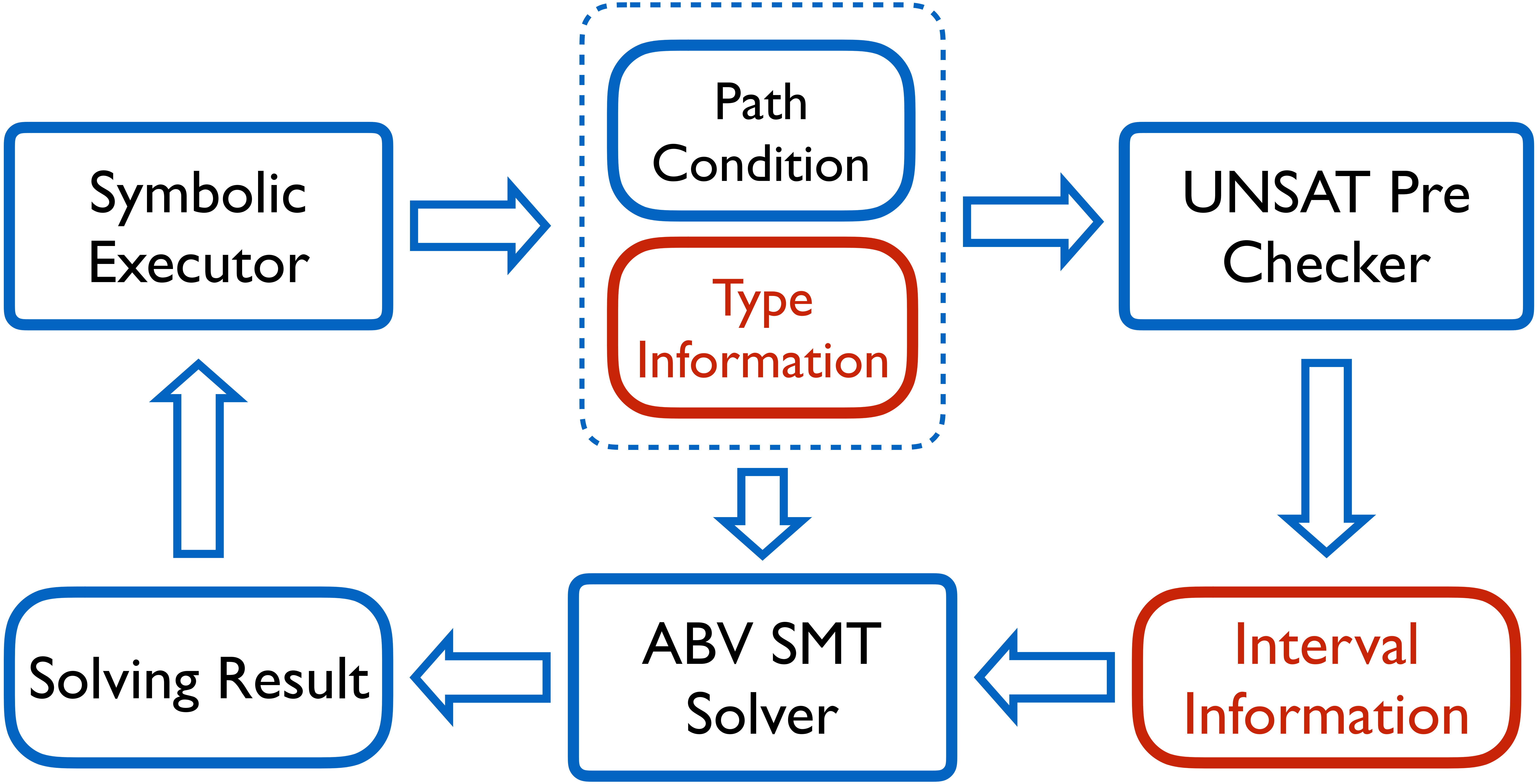
Our Key Idea

- Utilize the information calculated during symbolic execution
  - Type information of array accesses
  - Interval information of array index variables

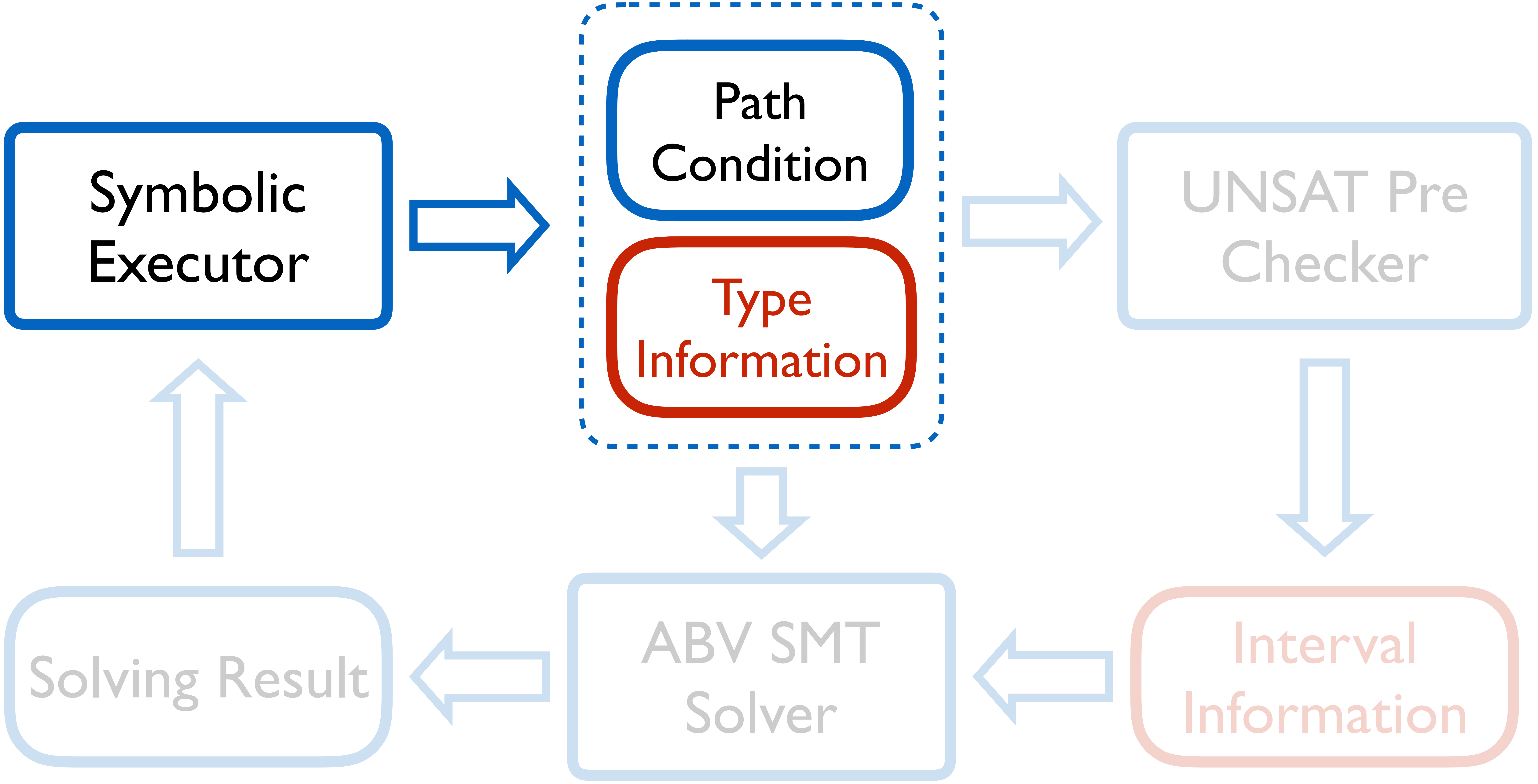
Our Key Idea

- Utilize the information calculated during symbolic execution
  - Type information of array accesses
  - Interval information of array index variables
- Check the unsatisfiability earlier
- Remove redundant axioms during solving

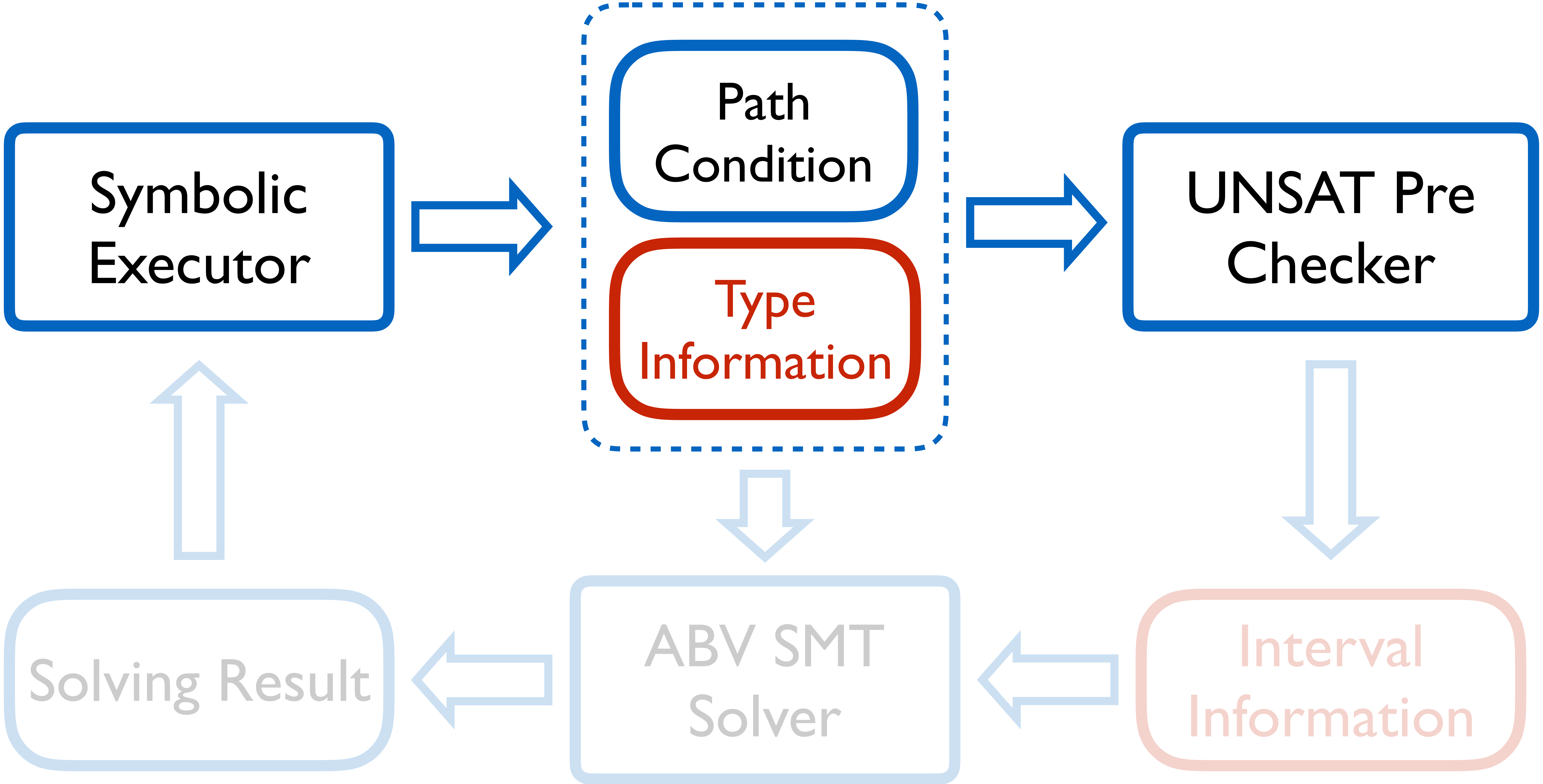
High-Level Procedure



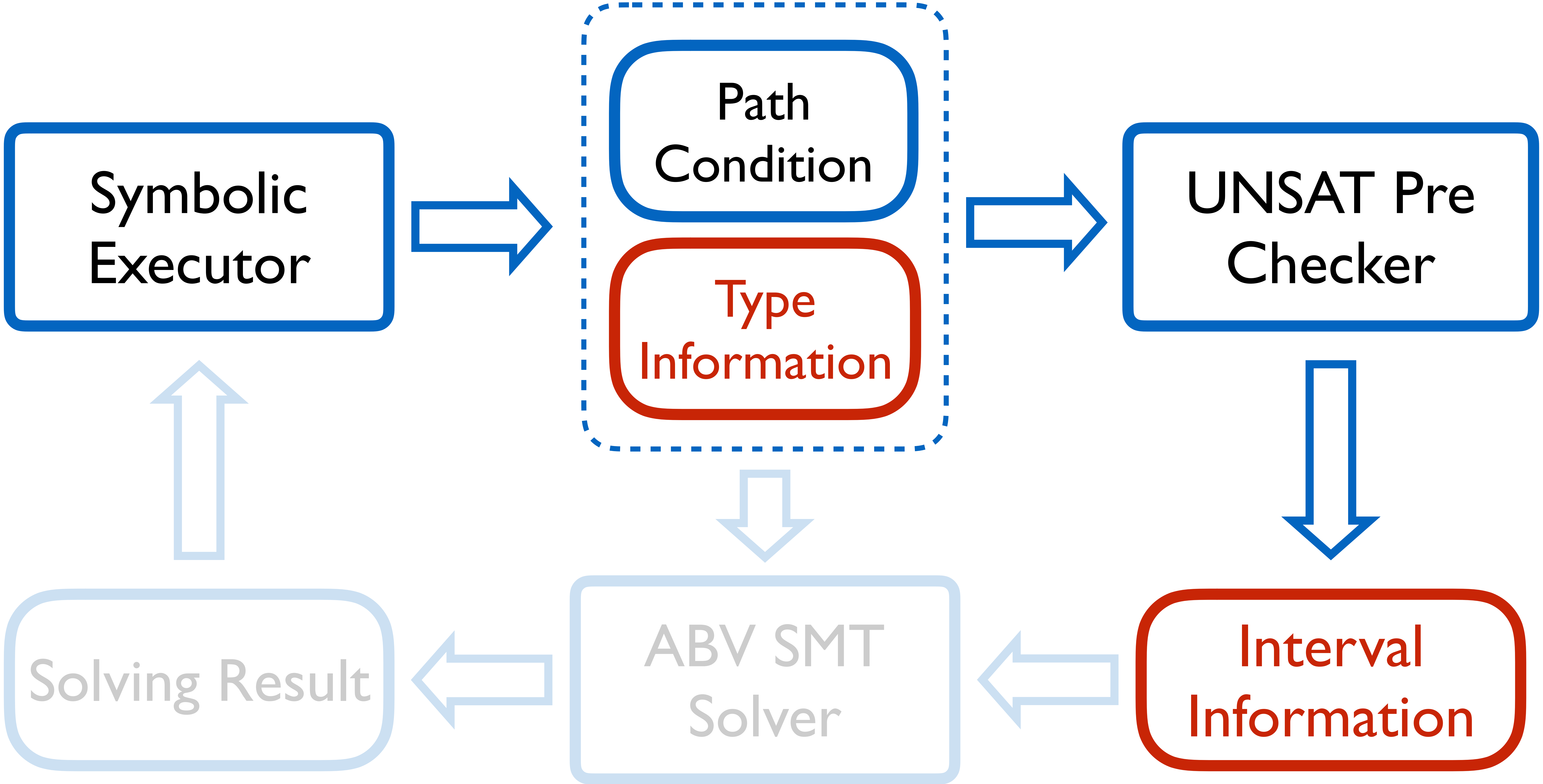
High-Level Procedure



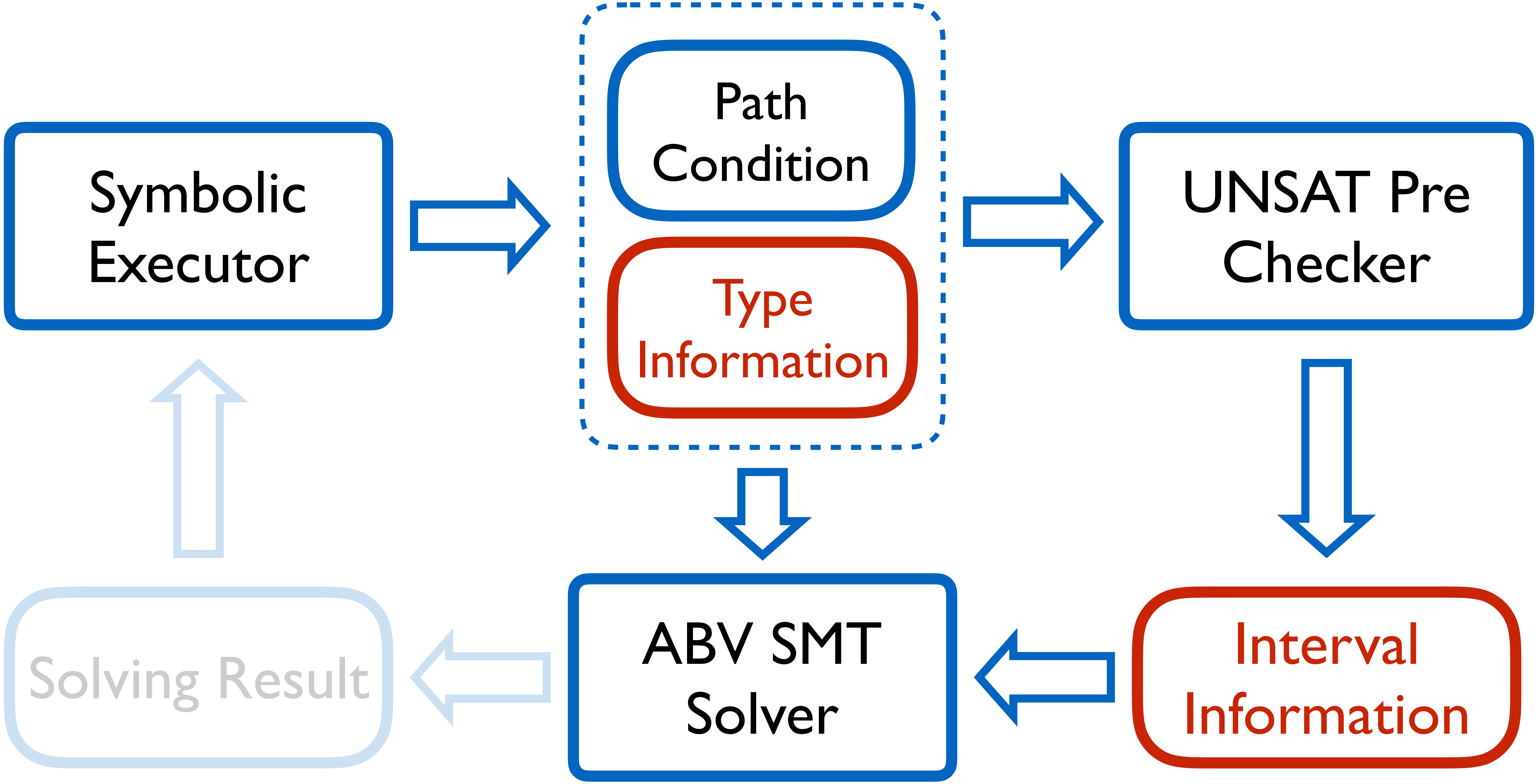
High-Level Procedure



High-Level Procedure



High-Level Procedure



Motivation Example

i, j ∈ [0, 3]

```
int foo(int i, j) {
   int a[4] = {0, 0, 0, 5}
   if (i + j > 4) {
      if (a[i] + a[j] > 10) {
         printf("Bug!!!\n")
         return 1
      }
   }
   return 0
}
```

Motivation Example

i, j ∈ [0, 3]

```
int foo(int i, j) {
  int a[4] = {0, 0, 0, 5}
  if (i + j > 4) {
    if (a[i] + a[j] > 10) {
      printf("Bug!!!\n")
      return 1
    }
  }
  return 0
}
```

$$0 \leq i \leq 3 \wedge 0 \leq j \leq 3 \wedge i + j > 4$$

$$\bigwedge$$

$$R(a,\ i) + R(a,\ j) > 10$$

$$a[4] = \{0,\ 0,\ 0,\ 5\}$$

Motivation Example

i, j ∈ [0, 3]

```
int foo(int i, j) {
   int a[4] = {0, 0, 0, 5}
   if (i + j > 4) {
      if (a[i] + a[j] > 10) {
         printf("Bug!!!\n")
         return 1
      }
   }
   return 0
}
```

UNSAT Pre-check

$$0 \leq i \leq 3 \wedge 0 \leq j \leq 3 \wedge i + j > 4$$

Index Constraints

$$\bigwedge$$

$$R(a,\ i) + R(a,\ j) > 10$$

$$a[4] = \{0,\ 0,\ 0,\ 5\}$$

Array Constraint

Motivation Example

i, j ∈ [0, 3]

```
int foo(int i, j) {
  int a[4] = {0, 0, 0, 5}
  if (i + j > 4) {
    if (a[i] + a[j] > 10) {
      printf("Bug!!!\n")
      return 1
    }
  }
  return 0
}
```

UNSAT Pre-check

$$0 \leq i \leq 3 \wedge 0 \leq j \leq 3 \wedge i + j > 4$$

Index Constraints

ILP

Motivation Example

i, j ∈ [0, 3]

```
int foo(int i, j) {
    int a[4] = {0, 0, 0, 5}
    if (i + j > 4) {
        if (a[i] + a[j] > 10) {
            printf("Bug!!!\n")
            return 1
        }
    }
    return 0
}
```

UNSAT Pre-check

Index Constraints

$$0 \le i \le 3 \land 0 \le j \le 3 \land i + j > 4$$

ILP

$$2 \le i \le 3 \land 2 \le j \le 3$$

Motivation Example

i, j ∈ [0, 3]

```
int foo(int i, j) {
  int a[4] = {0, 0, 0, 5}
  if (i + j > 4) {
    if (a[i] + a[j] > 10) {
      printf("Bug!!!\n")
      return 1
    }
  }
  return 0
}
```

UNSAT Pre-check

Index Constraints

$$0 \leq i \leq 3 \wedge 0 \leq j \leq 3 \wedge i + j > 4$$

ILP

$$2 \leq i \leq 3 \wedge 2 \leq j \leq 3$$

$$a[4] = \{0,\ 0,\ 0,\ 5\}$$

Motivation Example

i, j ∈ [0, 3]

```
int foo(int i, j) {
   int a[4] = {0, 0, 0, 5}
   if (i + j > 4) {
      if (a[i] + a[j] > 10) {
         printf("Bug!!!\n")
         return 1
      }
   }
   return 0
}
```

UNSAT Pre-check

$$0 \leq i \leq 3 \wedge 0 \leq j \leq 3 \wedge i + j > 4$$

Index Constraints

ILP

$$2 \leq i \leq 3 \wedge 2 \leq j \leq 3$$

$$a[4] = \{0,\ 0,\ 0,\ 5\}$$

$$0 \leq R(a,\ i) \leq 5 \wedge 0 \leq R(a,\ j) \leq 5$$

Motivation Example

i, j ∈ [0, 3]

```
int foo(int i, j) {
  int a[4] = {0, 0, 0, 5}
  if (i + j > 4) {
    if (a[i] + a[j] > 10) {
      printf("Bug!!!\n")
      return 1
    }
  }
  return 0
}
```

UNSAT Pre-check

$0 \leq i \leq 3 \wedge 0 \leq j \leq 3 \wedge i + j > 4$

Index Constraints

ILP

$2 \leq i \leq 3 \wedge 2 \leq j \leq 3$

$a[4] = \{0,\ 0,\ 0,\ 5\}$

$0 \leq R(a,\ i) \leq 5 \wedge 0 \leq R(a,\ j) \leq 5$

Over-approximation

Motivation Example

$i, j \in [0, 3]$

```
int foo(int i, j) {
   int a[4] = {0, 0, 0, 5}
   if (i + j > 4) {
      if (a[i] + a[j] > 10) {
➡     printf("Bug!!!\n")
        return 1
      }
   }
   return 0
}
```

UNSAT Pre-check

$$0 \leq R(a,\, i) \leq 5 \wedge 0 \leq R(a,\, j) \leq 5$$

$$\bigwedge$$

$$R(a,\, i) + R(a,\, j) > 10$$

ILP ➡ ✖

Motivation Example

i, j ∈ [0, 3]

```
int foo(int i, j) {
   int a[4] = {0, 0, 0, 5}
   if (i + j > 4) {
      if (a[i] + a[j] > 10) {
         printf("Bug!!!\n")
         return 1
      }
   }
   return 0
}
```

UNSAT Pre-check

$$0 \leq R(a,\, i) \leq 5 \wedge 0 \leq R(a,\, j) \leq 5$$

$$\bigwedge$$

$$R(a,\, i) + R(a,\, j) > 10$$

ILP

Unsatisfiable!!!

Motivation Example

i, j ∈ [0, 3]

```
int foo(int i, j) {
   int a[4] = {0, 0, 0, 9}
   if (i + j > 4) {
      if (a[i] + a[j] > 10) {
         printf("Bug!!!\n")
         return 1
      }
   }
   return 0
}
```

Motivation Example

i, j ∈ [0, 3]

```
int foo(int i, j) {
   int a[4] = {0, 0, 0, 9}
   if (i + j > 4) {
      if (a[i] + a[j] > 10) {
         printf("Bug!!!\n")
         return 1
      }
   }
   return 0
}
```

$$0 \leq i \leq 3 \wedge 0 \leq j \leq 3 \wedge i + j > 4$$

$$\bigwedge$$

$$R(a, i) + R(a, j) > 10$$

$$a[4] = \{0, 0, 0, 9\}$$

Motivation Example

i, j ∈ [0, 3]

UNSAT Pre-check

```
int foo(int i, j) {
  int a[4] = {0, 0, 0, 9};
  if (i + j > 4) {
    if (a[i] + a[j] > 10) {
      printf("Bug!!!\n")
      return 1
    }
  }
  return 0
}
```

$$0 \leq R(a,\, i) \leq 9 \wedge 0 \leq R(a,\, j) \leq 9$$

$$\wedge$$

$$R(a,\, i) + R(a,\, j) > 10$$

Motivation Example

$i, j \in [0, 3]$

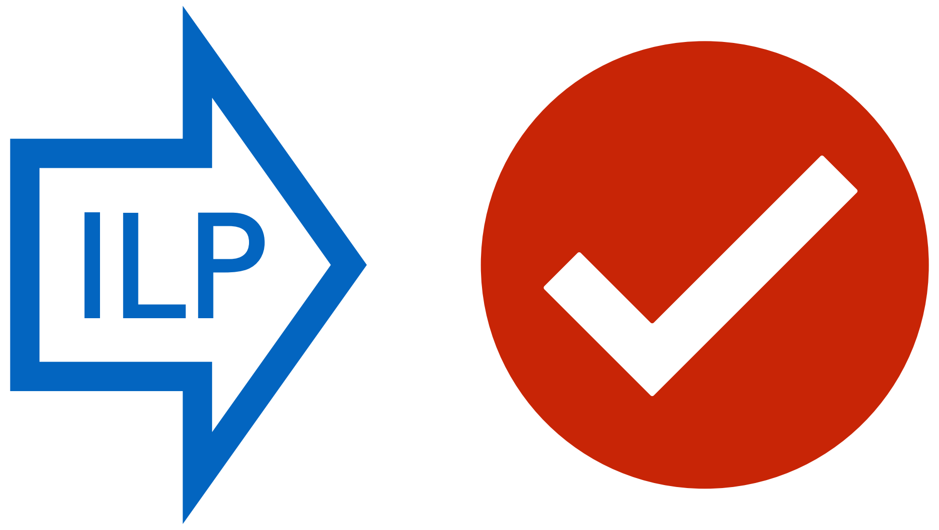
```
int foo(int i, j) {
  int a[4] = {0, 0, 0, 9};
  if (i + j > 4) {
    if (a[i] + a[j] > 10) {
      printf("Bug!!!\n")
      return 1
    }
  }
  return 0
}
```

UNSAT Pre-check

$$0 \le R(a, i) \le 9 \wedge 0 \le R(a, j) \le 9$$

$$\wedge$$

$$R(a, i) + R(a, j) > 10$$



Motivation Example

i, j ∈ [0, 3]

```
int foo(int i, j) {
  int a[4] = {0, 0, 0, 9};
  if (i + j > 4) {
    if (a[i] + a[j] > 10) {
      printf("Bug!!!\n")
      return 1
    }
  }
  return 0
}
```

UNSAT Pre-check

$0 \leq R(a, i) \leq 9 \wedge 0 \leq R(a, j) \leq 9$

$\wedge$

$R(a, i) + R(a, j) > 10$

ILP

Satisfiable??? Not sure!!!

Motivation Example

i, j ∈ [0, 3]

```
int foo(int i, j) {
  int a[4] = {0, 0, 0, 9};
  if (i + j > 4) {
    if (a[i] + a[j] > 10) {
      printf("Bug!!!\n")
      return 1
    }
  }
  return 0
}
```

Axiom elimination

Motivation Example

i, j ∈ [0, 3]

```
int foo(int i, j) {
  int a[4] = {0, 0, 0, 9};
  if (i + j > 4) {
    if (a[i] + a[j] > 10) {
      printf("Bug!!!\n")
      return 1
    }
  }
  return 0
}
```

Axiom elimination

- Interval info computed in pre-check

$$0 \leq i \leq 3 \wedge 0 \leq j \leq 3 \wedge i + j > 4$$

ILP

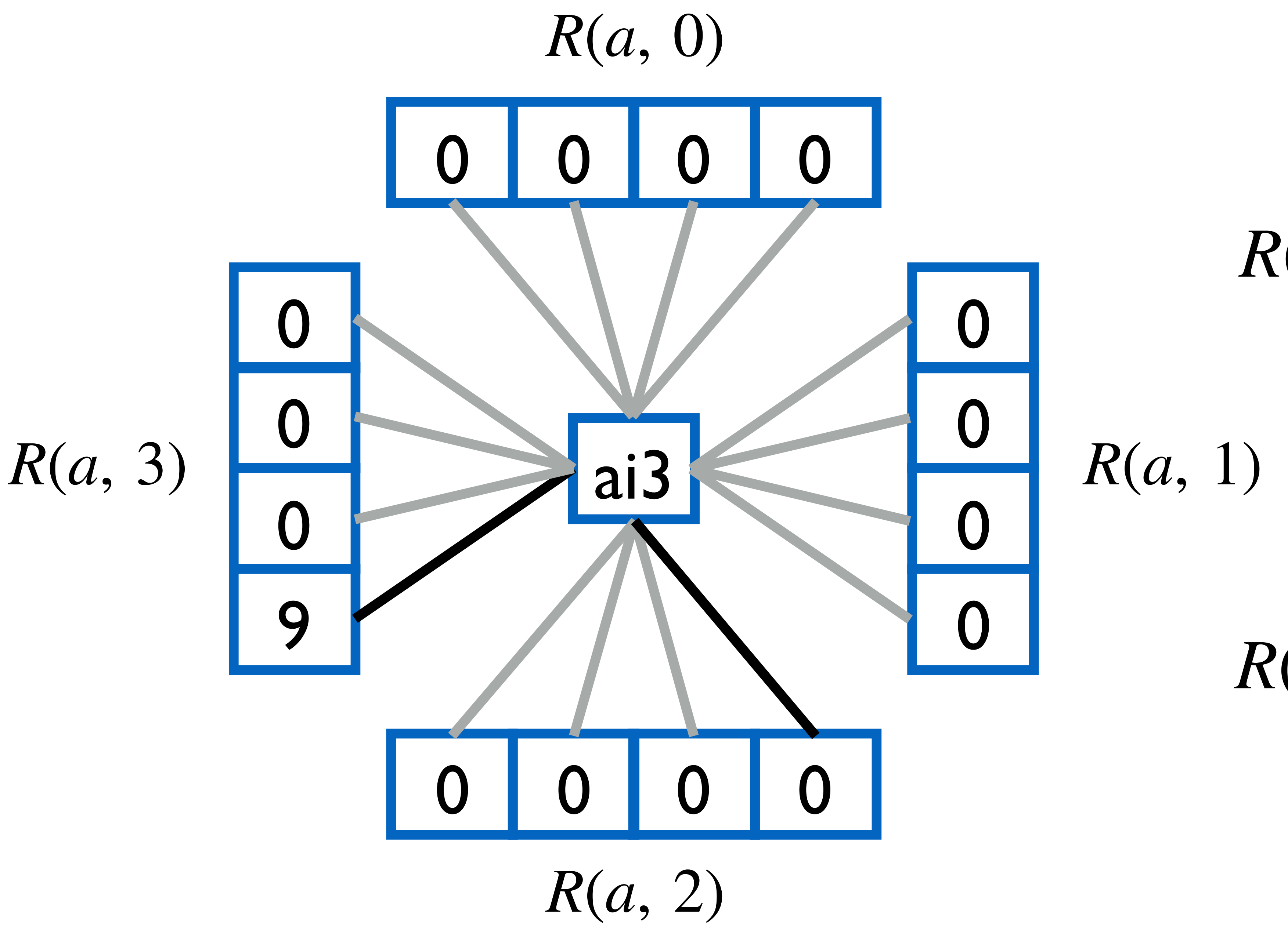
$$2 \leq i \leq 3 \wedge 2 \leq j \leq 3$$

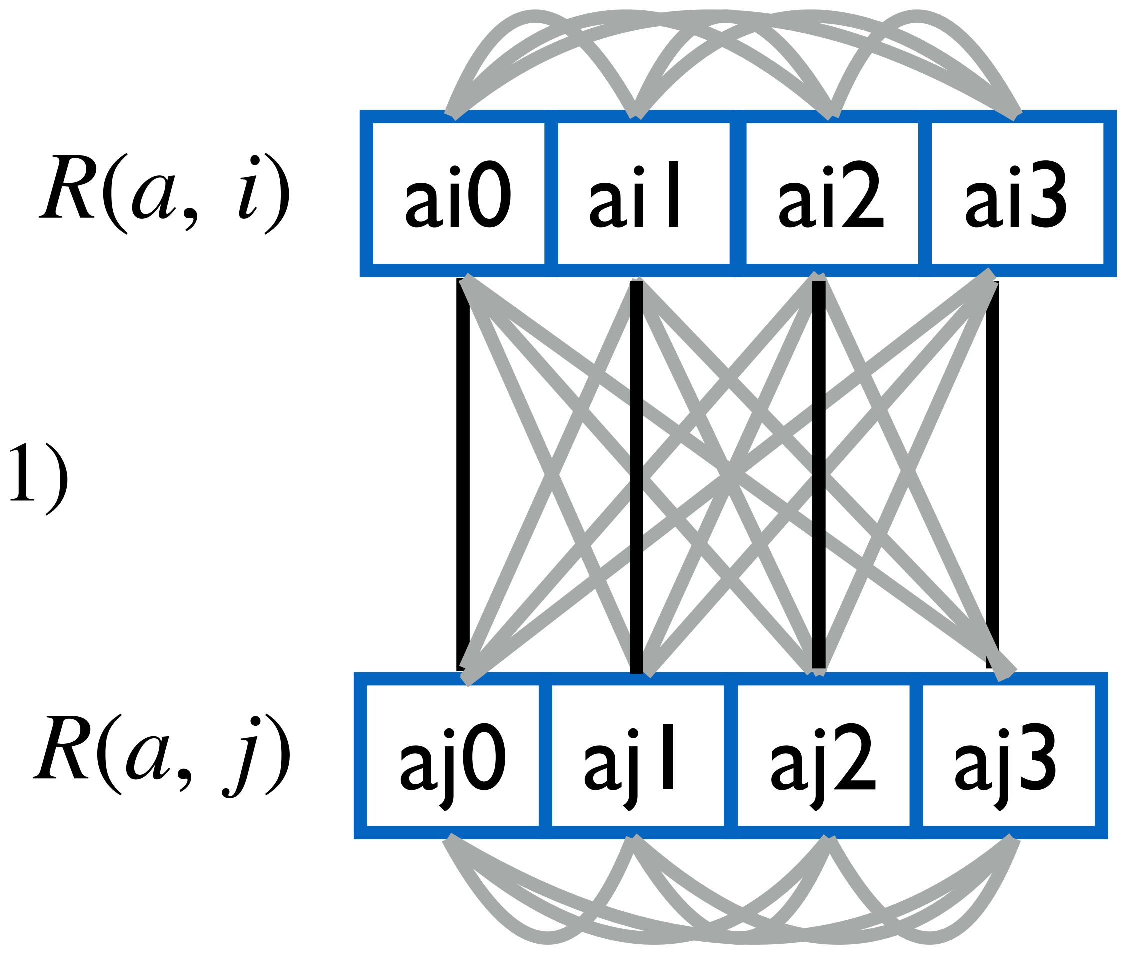
- Type info collected in SE (int)

Motivation Example

Array Memory Layout

offset	0	1	2	3
$R(a, i)$	ai0	ai1	ai2	ai3
$R(a, j)$	aj0	aj1	aj2	aj3
$R(a, 0)$	0	0	0	0
$R(a, 1)$	0	0	0	0
$R(a, 2)$	0	0	0	0
$R(a, 3)$	0	0	0	9





$$0 \leq i \leq 3 \wedge 0 \leq j \leq 3 \wedge i + j > 4 \wedge R(a, i) + R(a, j) > 10$$

156 axioms → 20 axioms

Evaluation

- Research Questions
  - Effectiveness
  - Relevance of either optimization
  - Comparison with KLEE-Array

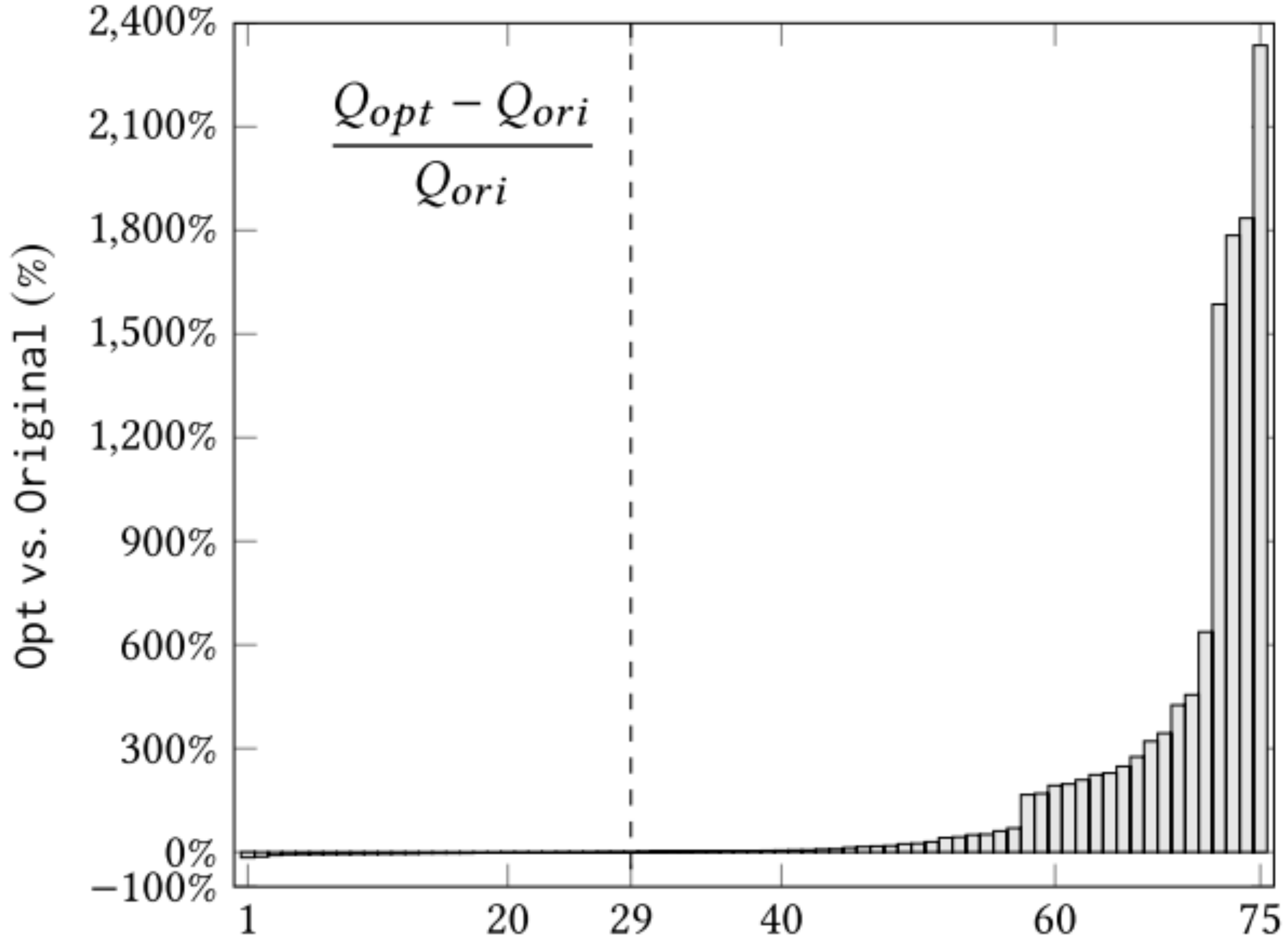
Evaluation

- Implementation
  - KLEE with STP
  - PPL solver for ILP solving

- Real-world programs as benchmark
  - Coreutils programs (62)
  - Lexer programs of various grammars (13)

Results of Effectiveness

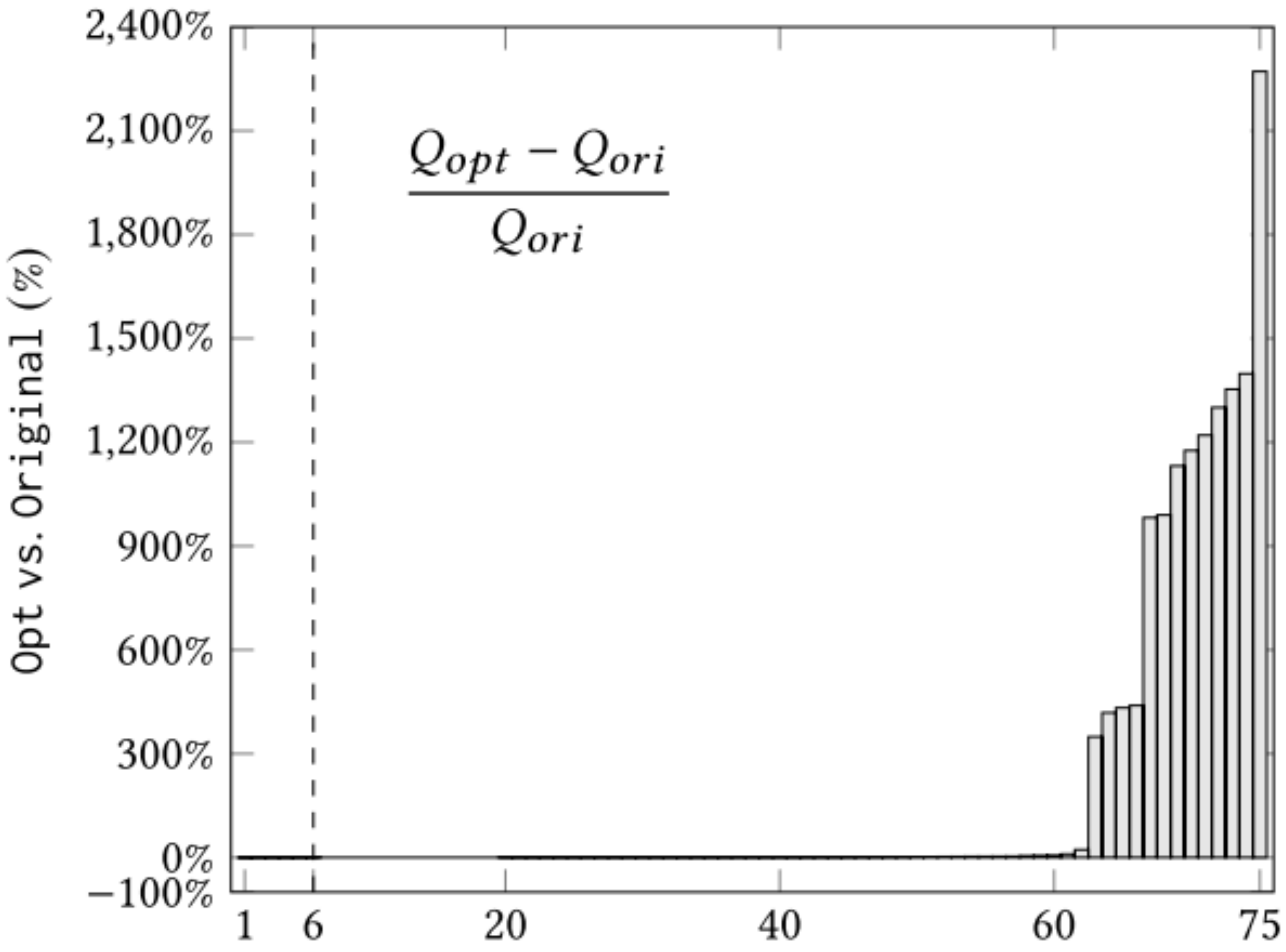
Queries without KLEE opt



Improves the queries for 46 programs, 160.52% on average

Results of Effectiveness

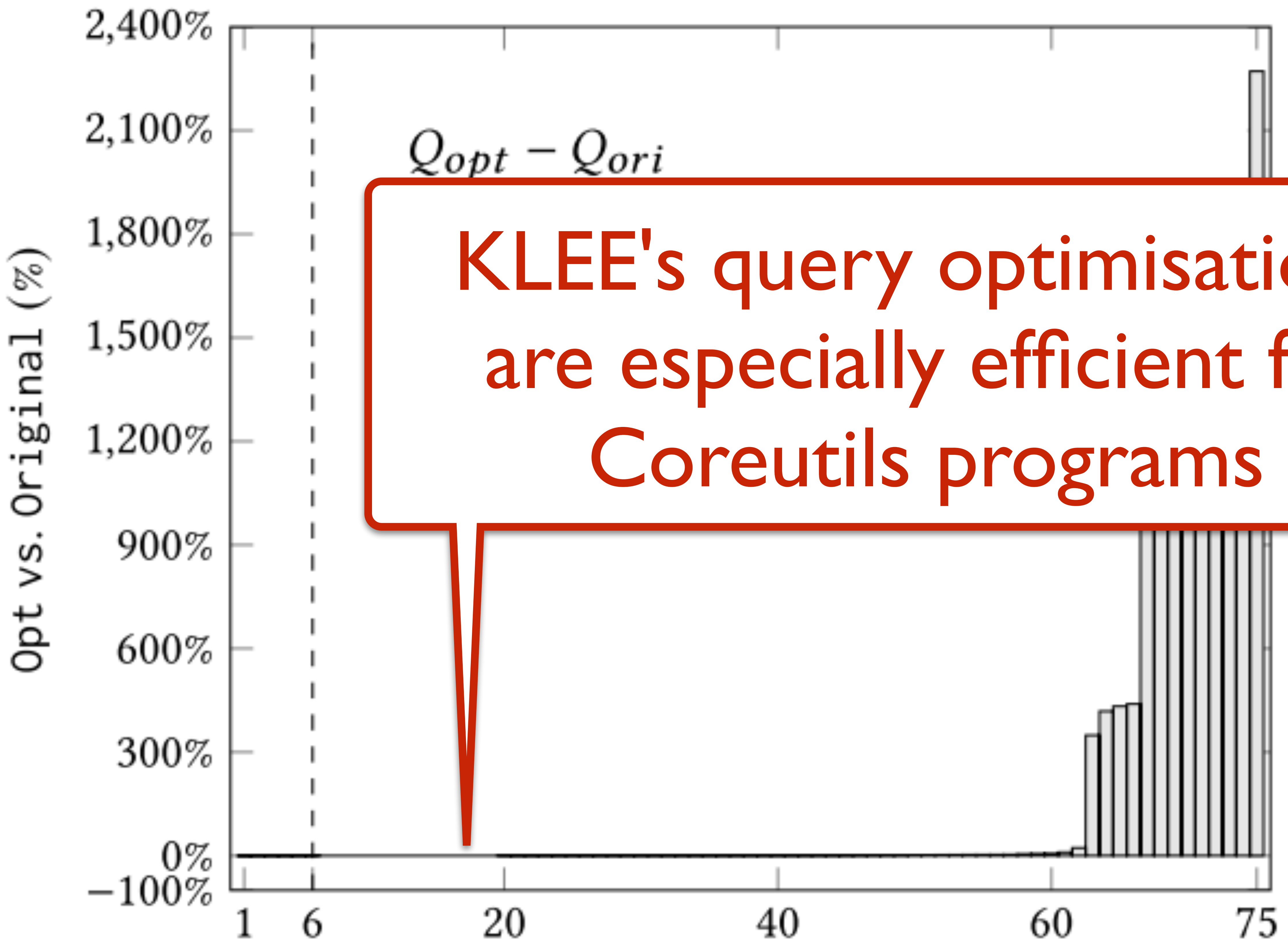
Queries with KLEE opt



Improves the queries for 56 programs, 182.56% on average

Results of Effectiveness

Queries with KLEE opt

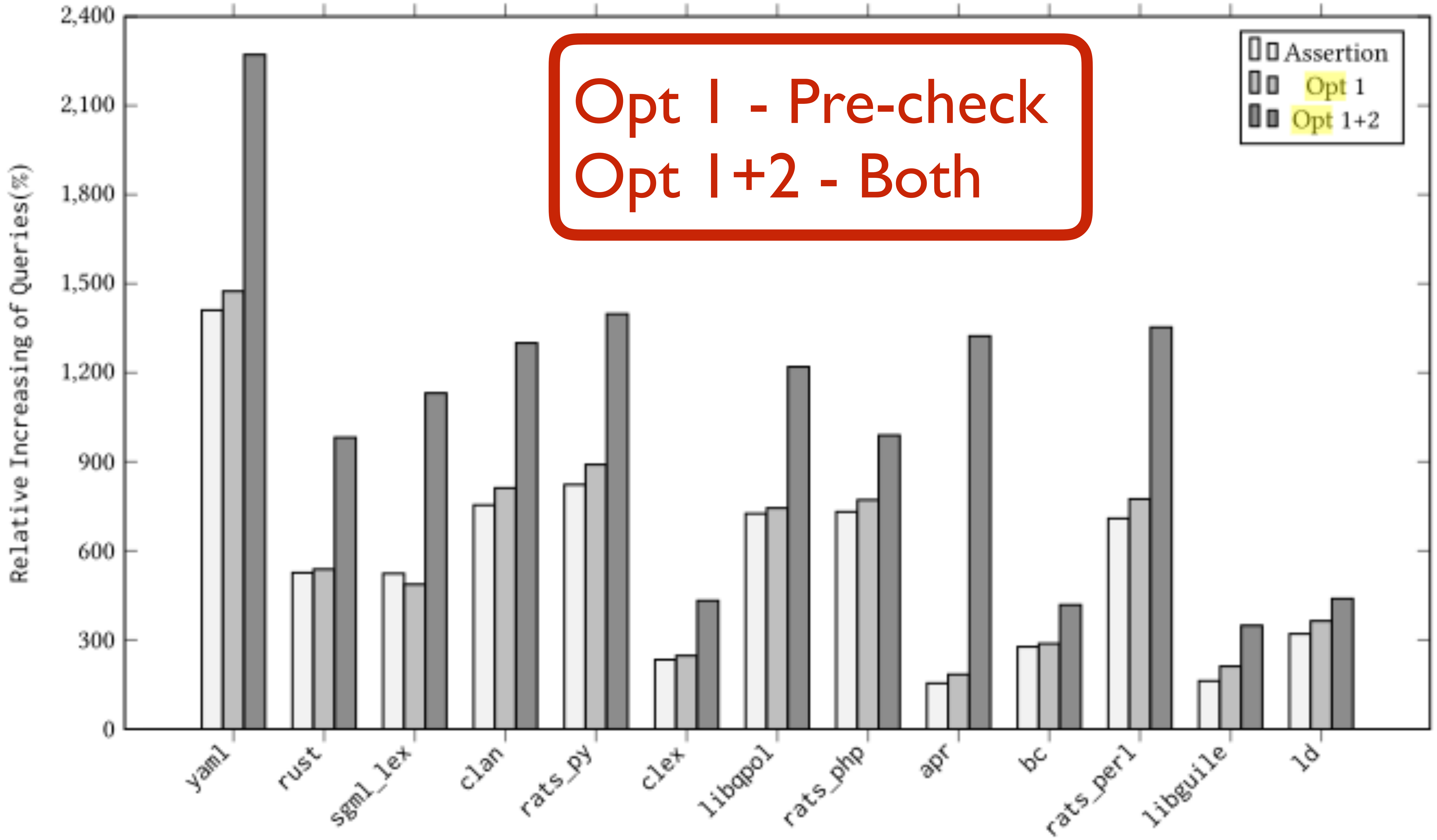


KLEE's query optimisations are especially efficient for Coreutils programs

Improves the queries for 56 programs, 182.56% on average

Results of Relevance

Queries
with
KLEE opt



Opt 1 - Pre-check
Opt 1+2 - Both

Opt 2 is more significant, while Opt 1 can generate useful information for Opt 2

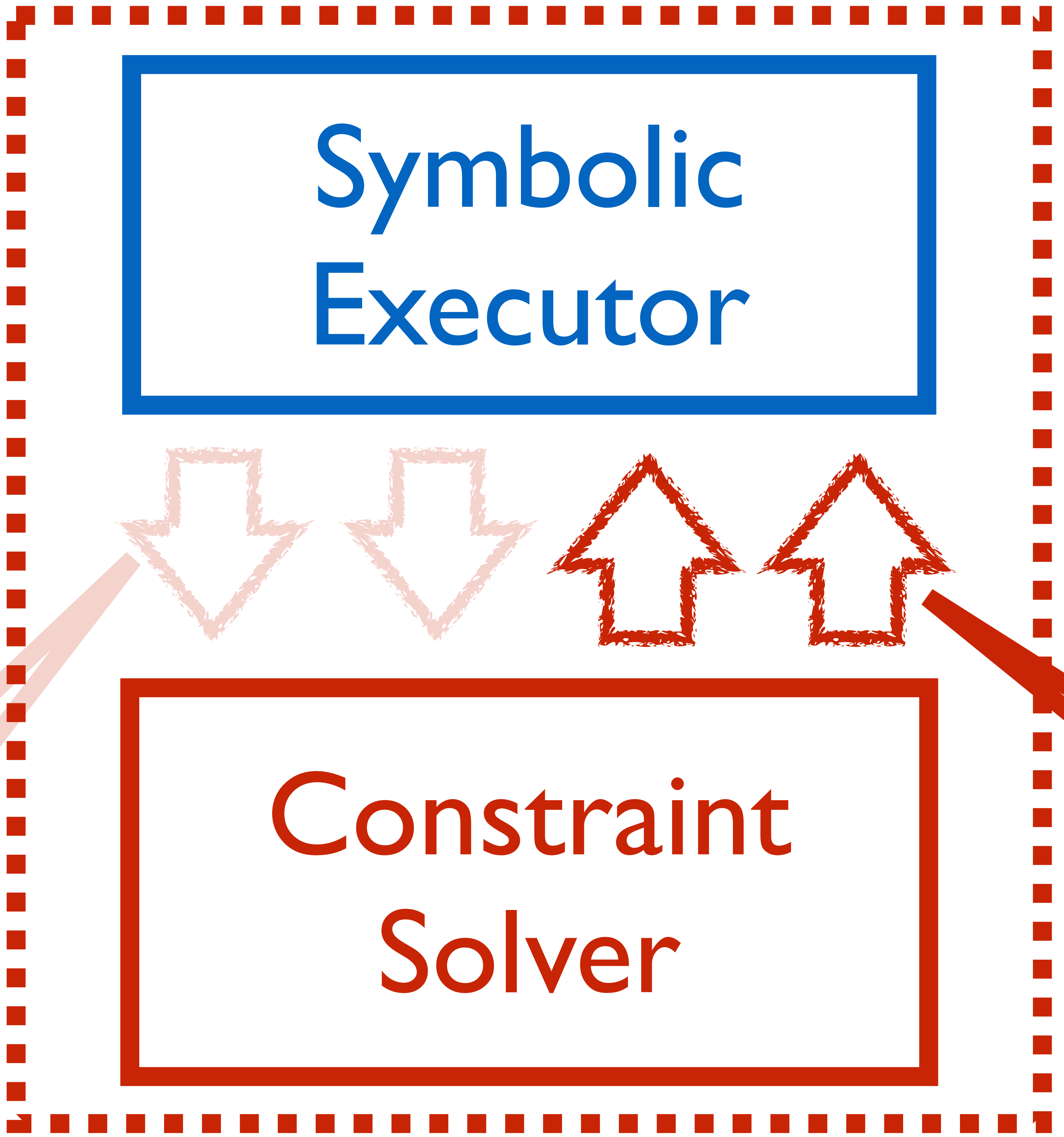
Comparison with KLEE-Array

With
KLEE opt

Programs	KLEE-Array		Our Method	
	#Instrs	#Paths	#Instrs	#Paths
yaml	71687	29	63864	28
rust	38892	24	53921	38
sgml_lex	599397	184	523956	165
clan	69777	66	89288	86
rats_py	353230	342	417394	401
clex	87322	87	115455	124
libqpol	35871	22	45190	35
rats_php	5221268	1554	14514660	4479
apr	637629	3456	880674	5542
bc	340874	36	440008	43
rats_perl	325398	338	379466	402
libguile	665723	337	750713	421
ld	373181619	489	373304921	584

Our method increases the number of paths and instructions by 30.31% and 40.39%, respectively

Our Recent Progress

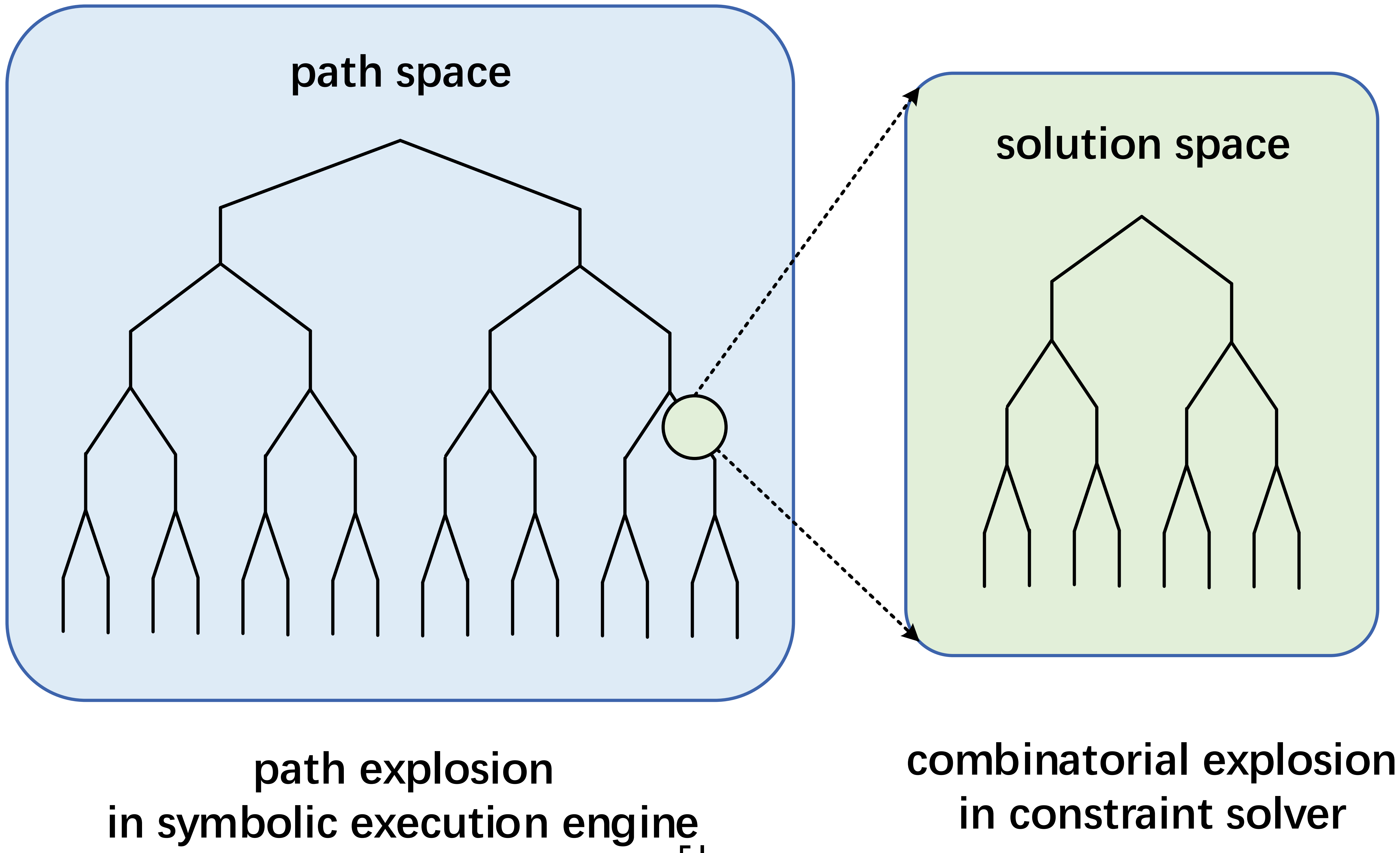


- Type and Interval Aware Array Constraint Solving [ISSTA 2021]

- Partial Solution Prompted Symbolic Execution [ASE 20]

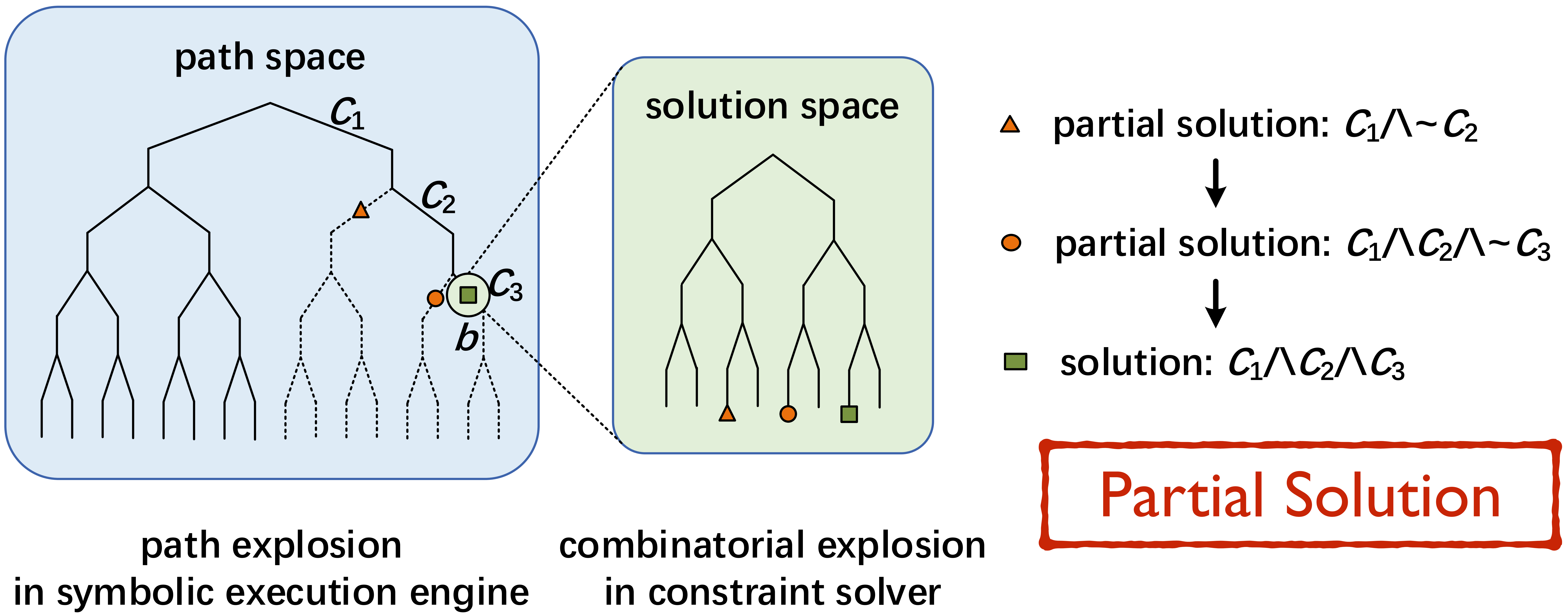
Multiplex Symbolic Execution

- Double explosions in symbolic execution



Multiplex Symbolic Execution

- Generate multiple test inputs by solving once

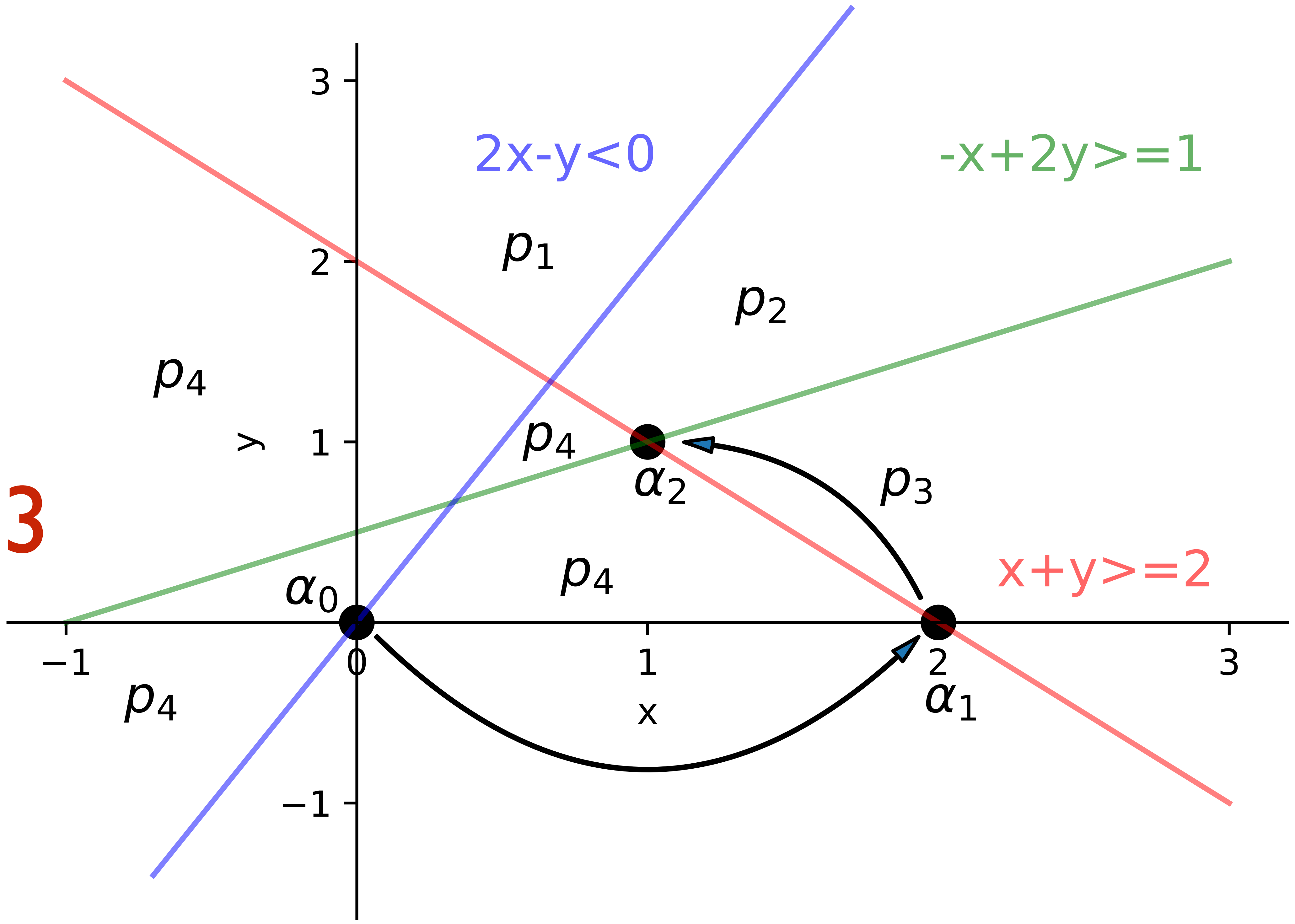


Motivation Example

```
public void start(int x,int y){
   if (x + y >= 2) {
      if(2 * y - x >= 1) {
         if(2 * x - y >= 0) {
            System.out.println("#2");
         } else {
            System.out.println("#1");
         }
      } else {
         System.out.println("#3");
      }
   } else {
      System.out.println("#4");
   }
}
```

Initial input: x = 1, y = 3

$$x + y \geq 2 \land 2y - x \geq 1 \land 2x - y \geq 0$$



Path space and solution space are related for the program's input space

Motivation Example

```java
public void start(int x,int y){
   if (x + y >= 2) {
      if(2 * y - x >= 1) {
         if(2 * x - y >= 0) {
            System.out.println("#2");
         } else {
            System.out.println("#1");
         }
      } else {
         System.out.println("#3");
      }
   } else {
      System.out.println("#4");
   }
}
```

First solving

x = 0, y = 0

Pivot

x = 2, y = 0

Pivot

x = 1, y = 1

$$x + y \geq 2 \wedge 2y - x \geq 1 \wedge 2x - y \geq 0$$

Motivation Example

```
public void start(int x,int y){
   if (x + y >= 2) {
      if(2 * y - x >= 1) {
         if(2 * x - y >= 0) {
            System.out.println("#2");
         } else {
            ...ntln("#1");
            ...n("#3");
      }
   } else {
      System.out.println("#4");
   }
}
```

Only need one time of solving

First solving

$x = 0, y = 0$

Pivot

$x = 2, y = 0$

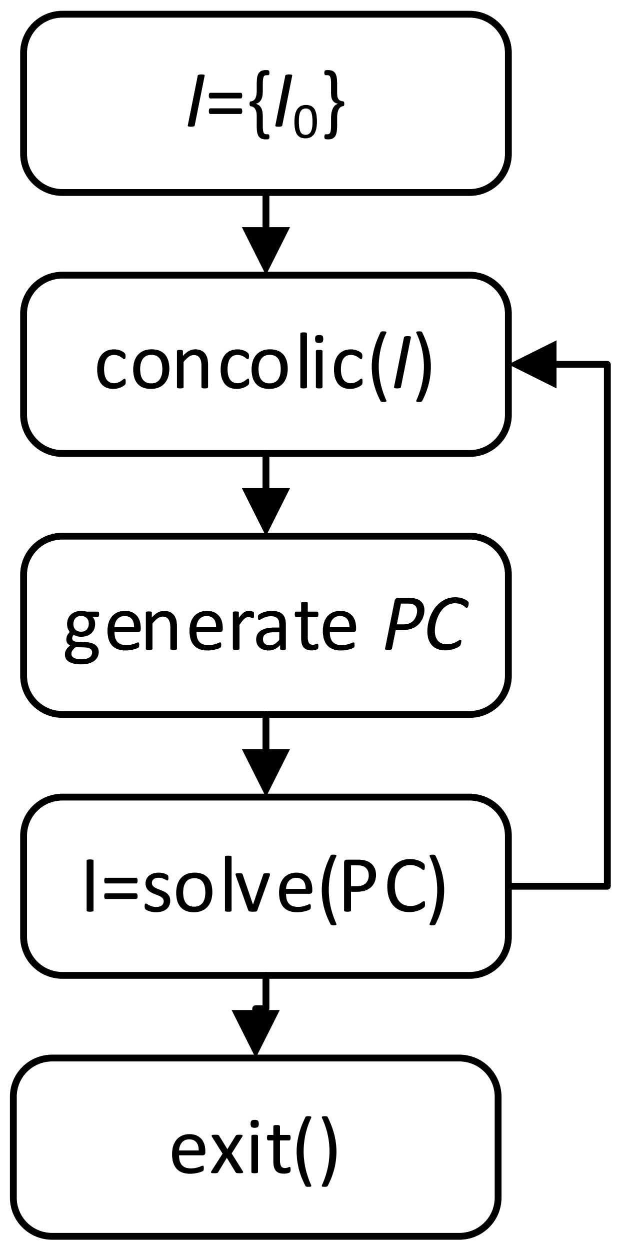
Pivot

$x = 1, y = 1$

$$x + y \geq 2 \wedge 2y - x \geq 1 \wedge 2x - y \geq 0$$

Multiplex DSE (MuSE)

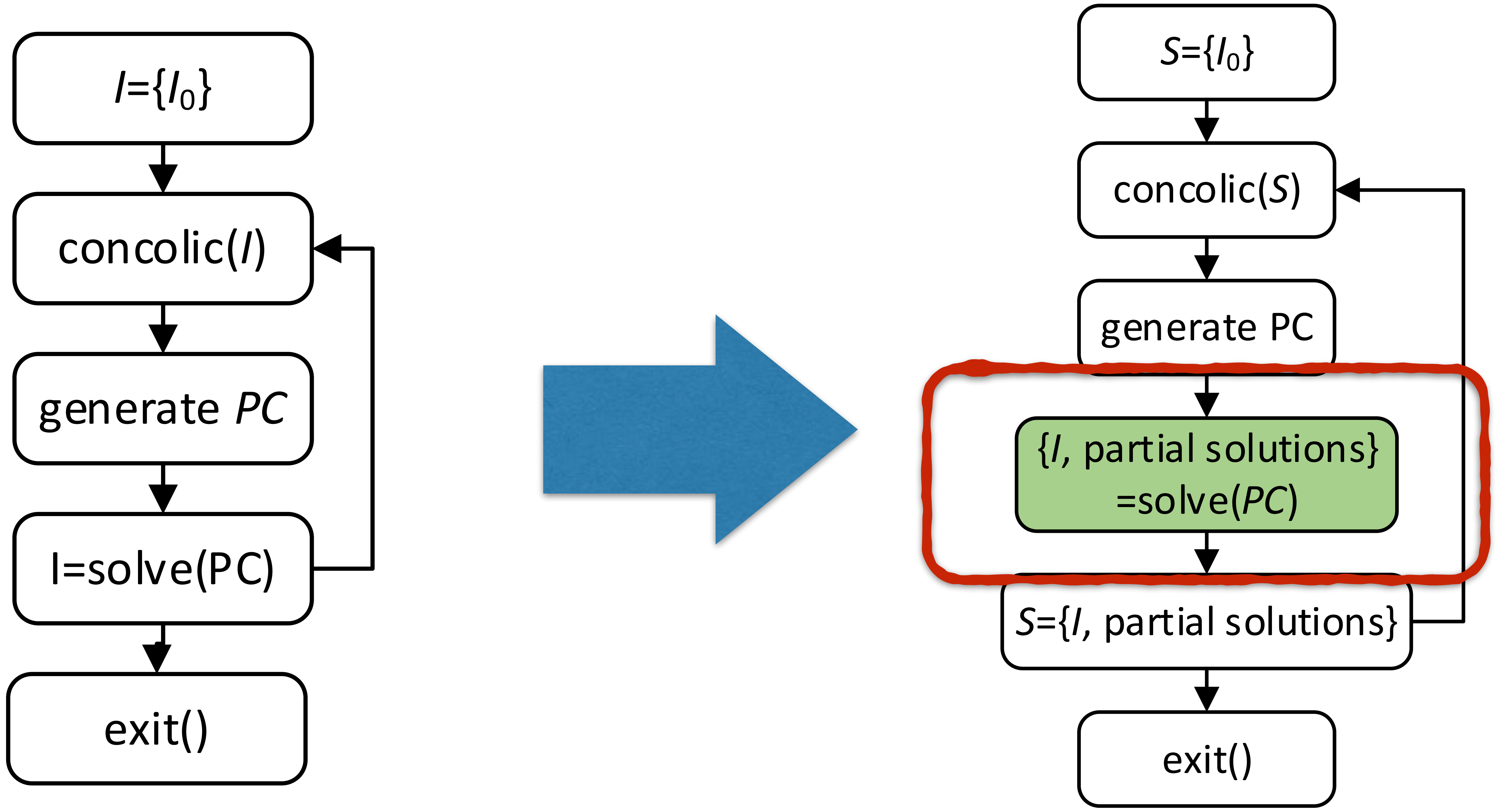
Utilize partial solutions for generating multiple tests by solving once during DSE



Vanilla Symbolic Execution

Multiplex DSE (MuSE)

Utilize partial solutions for generating multiple tests by solving once during DSE



Vanilla Symbolic Execution

MuSE

Partial Solutions are Ubiquitous

- CDCL/DPLL framework for SAT
- DPLL(T) framework for SMT
- JFS: coverage-guided fuzzing for FP constraints
- …

Partial Solution Support

- What we have done
  - QF_LIA: Simplex-based
  - QF_ABV: CEGAR-based
  - Optimization-based floating-point solving

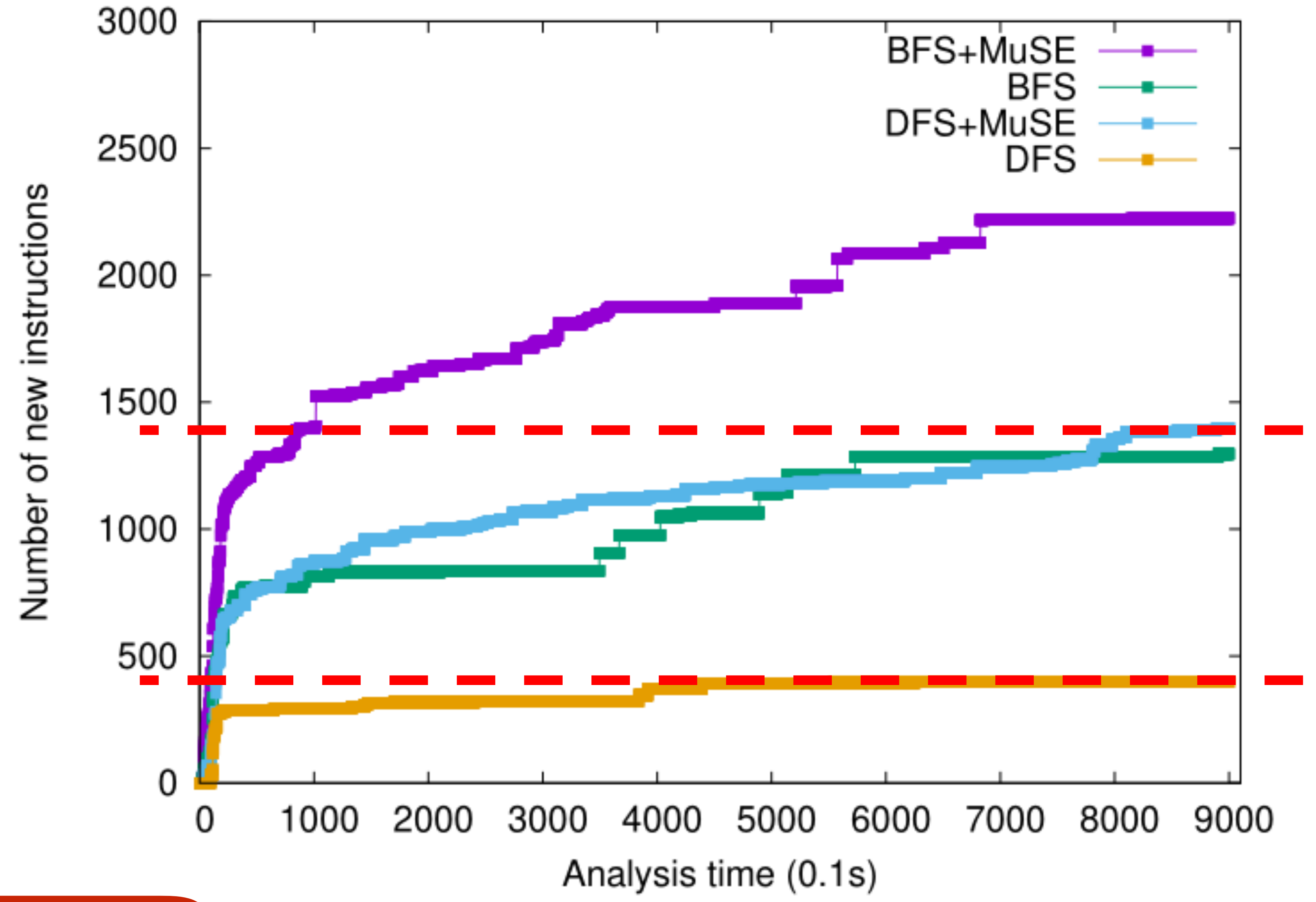
Evaluation - Implementation

- Solvers with partial solution support
  - QF_LIA on Z3
  - QF_ABV on STP
  - Optimization-based floating-point solving (Simulated annealing-based Java implementation)
- C programs: Concolic KLEE + QF_ABV(STP)
- Java programs: JFuzz + QF_LIA/QF_FP

Evaluation - Result (1/3)

- Simplex-baed QF_LIA solving

Programs	DFS+P		DFS		BFS+P		BFS	
	#T	#NI	#T	#NI	#T	#NI	#T	#NI
BMPDecorder	1125	134	5	0	3746	84	104	40
AviParser	340	117	144	46	1732	101	114	0
GifParser	721	25	60	5	1905	64	960	48
BMPParser	1203	52	8	0	4458	126	102	18
PGMParser	264	1	263	1	4736	188	7362	178
ImgParserPCX	387	38	81	20	2596	76	65	0
ImgParserBMP	458	314	114	21	1784	528	135	198
JaadParser	2083	64	134	0	2692	64	2835	59
Schroeder	1149	23	235	20	2267	29	402	22
JMP3Parser	214	286	37	198	319	653	279	646
Toba	1836	344	117	87	1670	311	179	87
Average	889	127	108	36	2536	202	1139	117



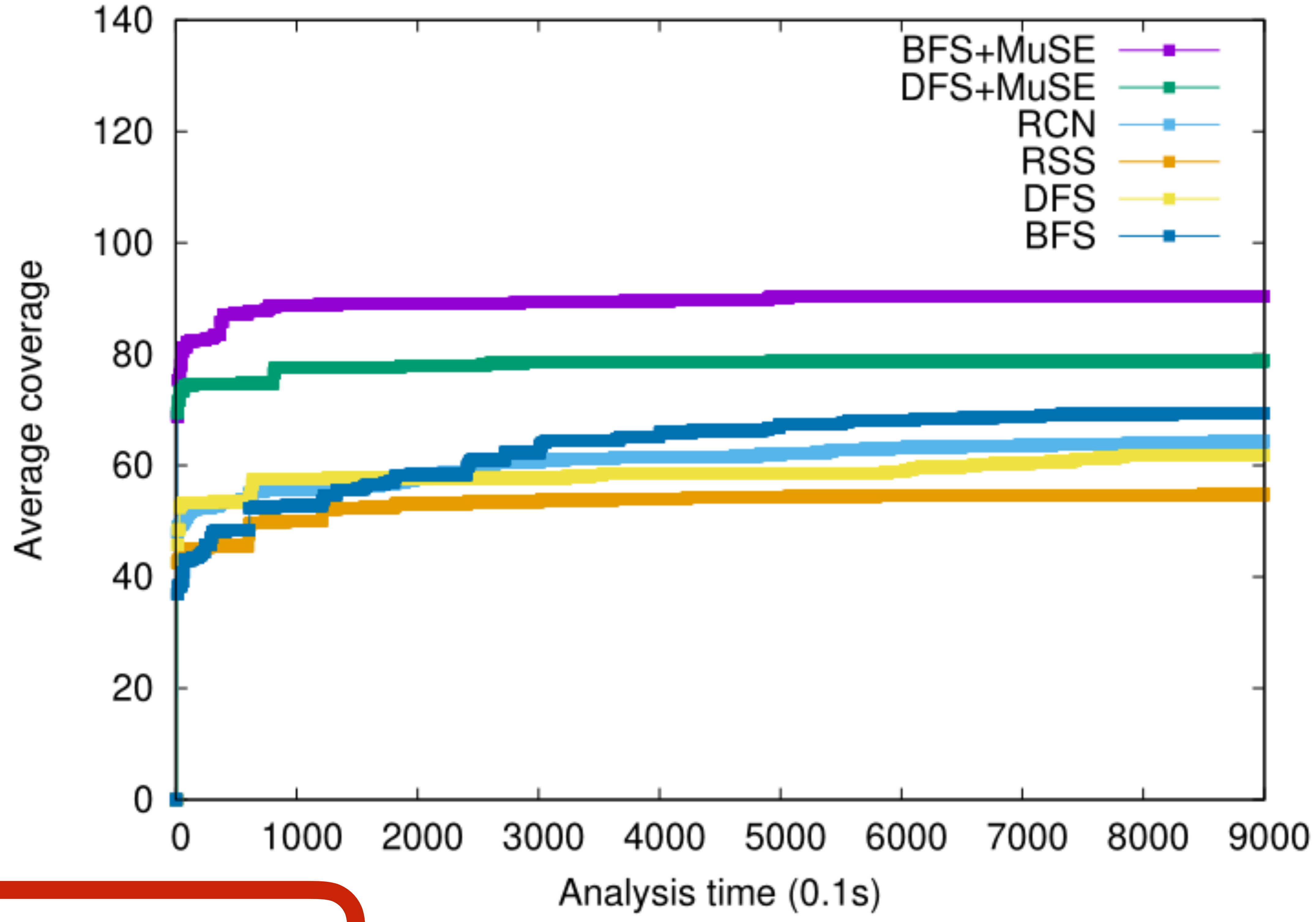
D(B)FS+P: D(B)FS + partial solution
#T: the number of test inputs
#NI: the number of new instructions covered after the first path

MuSE can cover more instructions

Evaluation - Result (2/3)

- CEGAR-based QF_ABV

Programs	DFS+P		BFS+P		Other Stategies			
	#PS	COV	#PS	COV	RCN	RSS	DFS	BFS
akimaei	1	64.7	514	76.1	76.5	67.2	65.3	64.9
bilinea	305	71.6	172	80.8	79.0	77.4	59.1	65.4
find	177	96.9	156	96.7	91.3	40.0	91.5	97.7
eigengs	19	73.5	118	98.0	67.6	51.6	61.1	82.8
fft-rrt	1015	46.8	350	99.5	39.6	38.6	46.5	11.3
h2d-ps	4	95.7	130	98.6	47.5	47.5	95.7	98.6
sort	18	100.0	9	100.0	89.7	82.2	83.7	44.6
sum-lu	29	76.5	129	88.6	70.8	50.7	70.1	43.1
linear-ed	13	63.1	1015	82.8	79.9	78.7	56.3	63.8
linear-ei	3	73.5	376	80.5	77.5	71.2	64.2	72.6
solve-ct	135	93.4	33	94.4	26.6	22.3	13.8	93.5
solve-ctn	32	94.2	2	96.0	21.7	19.2	30.0	95.5
steffen-ei	18	74.8	253	83.4	68.7	65.6	67.4	68.8
Average	136	78.8	250	90.4	64.3	54.8	61.9	69.4



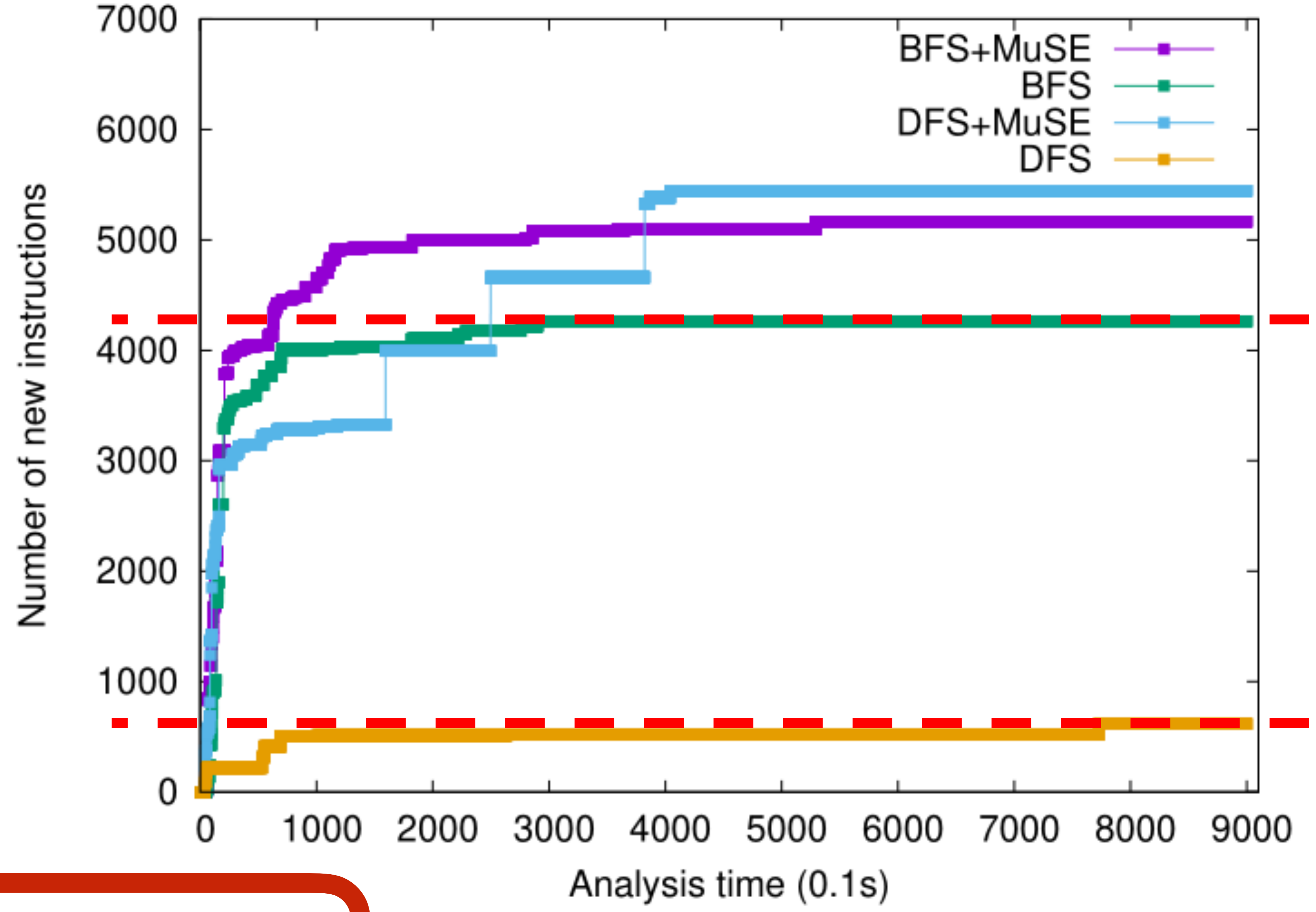
D(B)FS+P: D(B)FS + partial solution
#PS: the number of partial solutions
COV: LLVM code coverage

MuSE can achieve higher coverage

Evaluation - Result (3/3)

- Optimization-based Floating-point Solving

Programs	DFS+P		DFS		BFS+P		BFS	
	#T	#NI	#T	#NI	#T	#NI	#T	#NI
EigenD	3	244	1	0	477	1028	20	965
JacobiS	1424	13	43	6	1151	13	43	6
CholeskyD	1376	1335	43	4	1116	8	42	8
LeastS	169	2000	1	0	573	2246	43	2196
SquareR	1541	166	43	4	1240	8	44	8
EDAnalysis	8	418*	3	3*	8	392*	3	3*
Mutil	10	7*	4	0*	10	7*	4	0*
RankAnalysis	255	406	15	180	325	427*	20	427*
SVDAnalysis	204	427*	38	418*	276	427*	19	427*
TVSAnalysis	343	430	1	0	484	612	7	225
Average	484	495	17	55	514	469	22	387

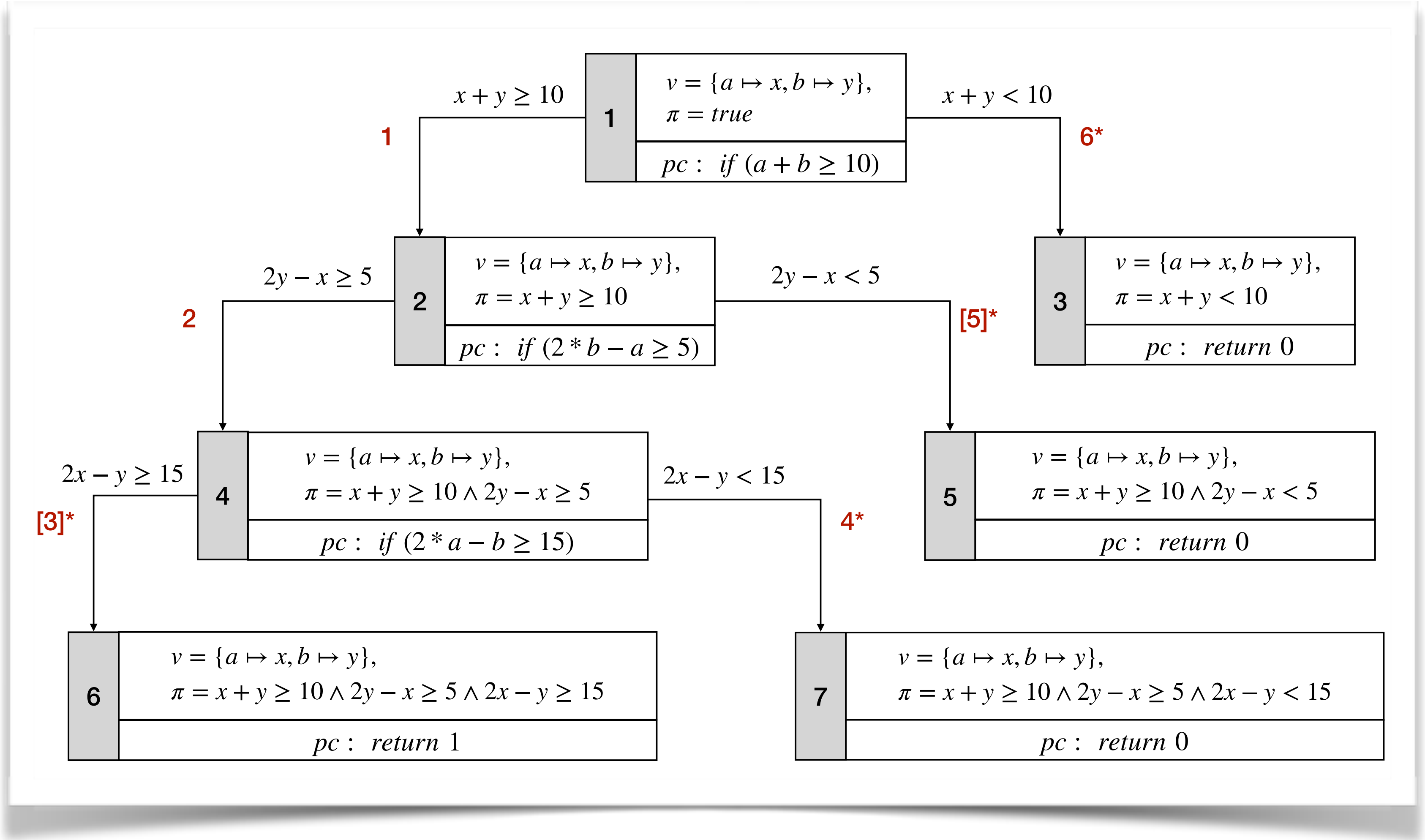


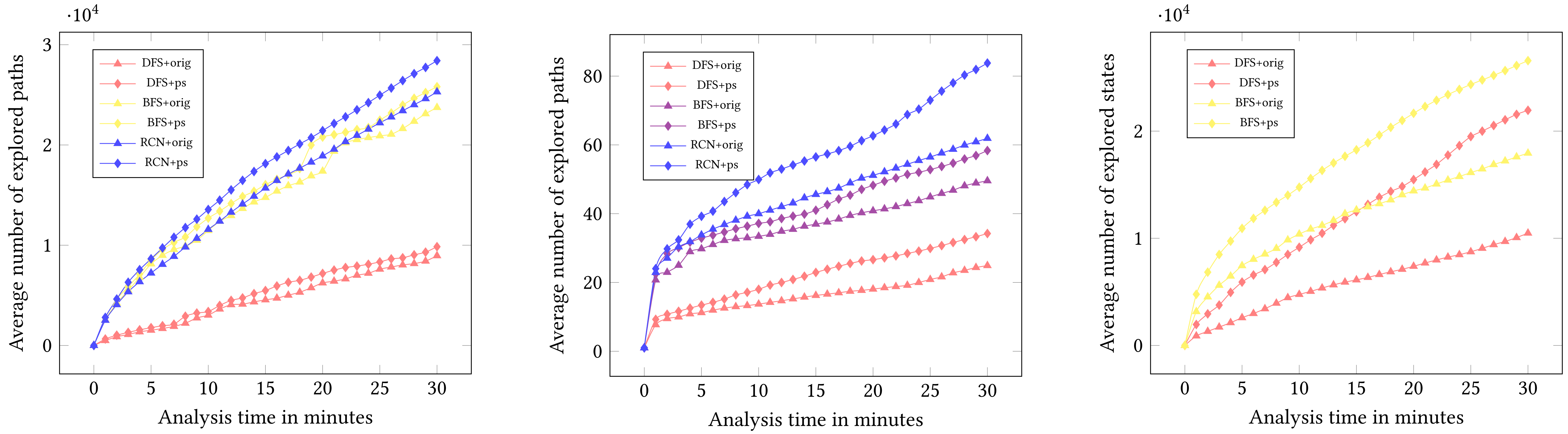
D(B)FS+P: D(B)FS + partial solution
#T: the number of test inputs
#NI: the number of new instructions covered after the first path

MuSE can cover more instructions

Follow-up Work

Partial solution-based constraint solving cache for symbolic execution

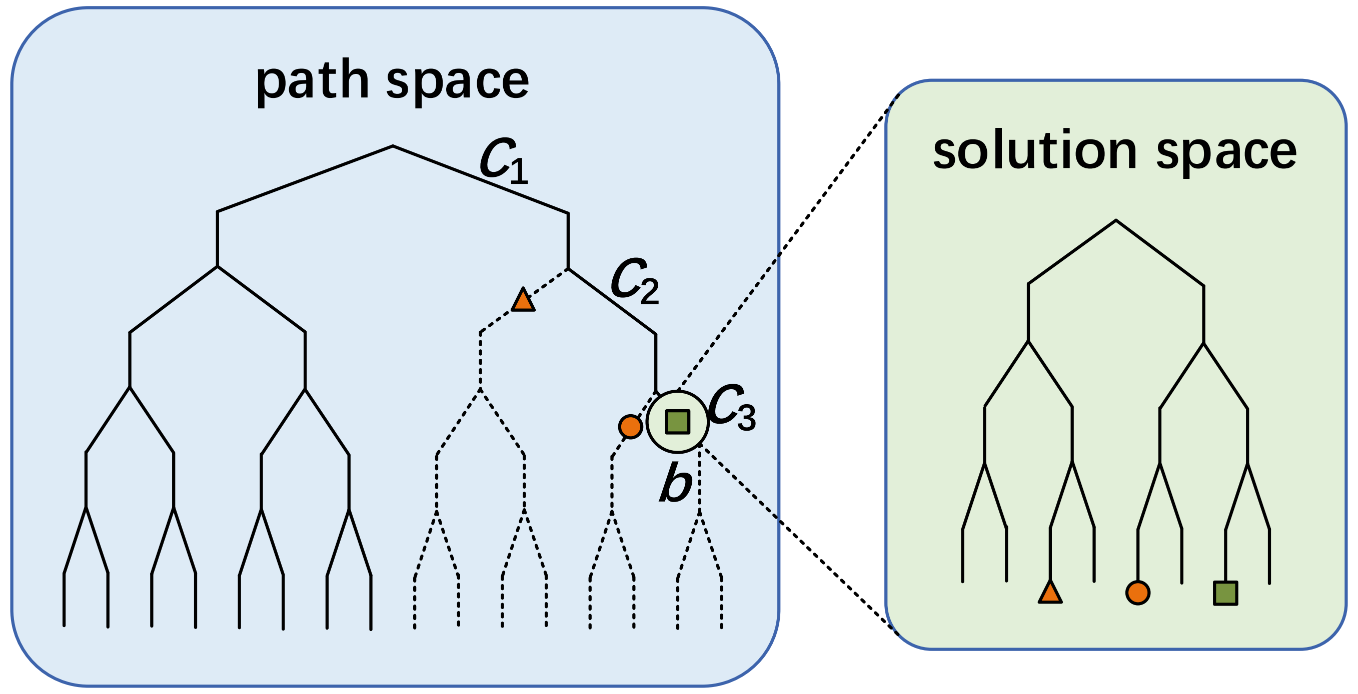


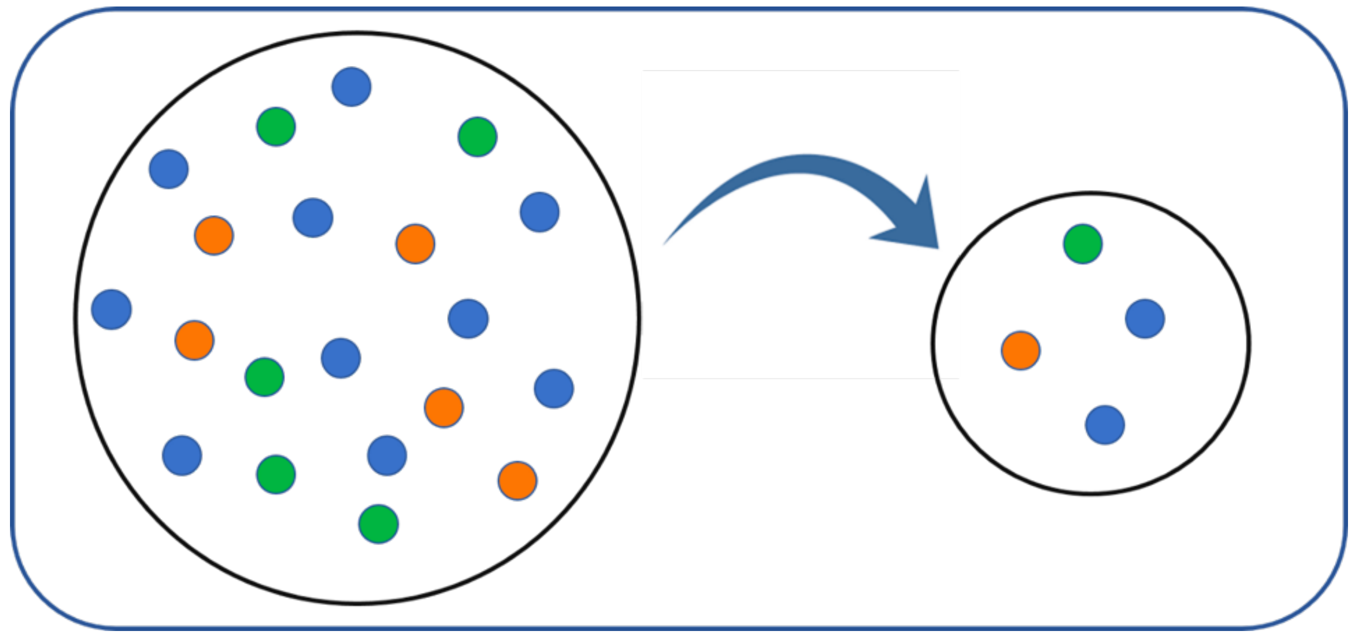


Utilize partial solutions to enrich solving cache and improve cache hit

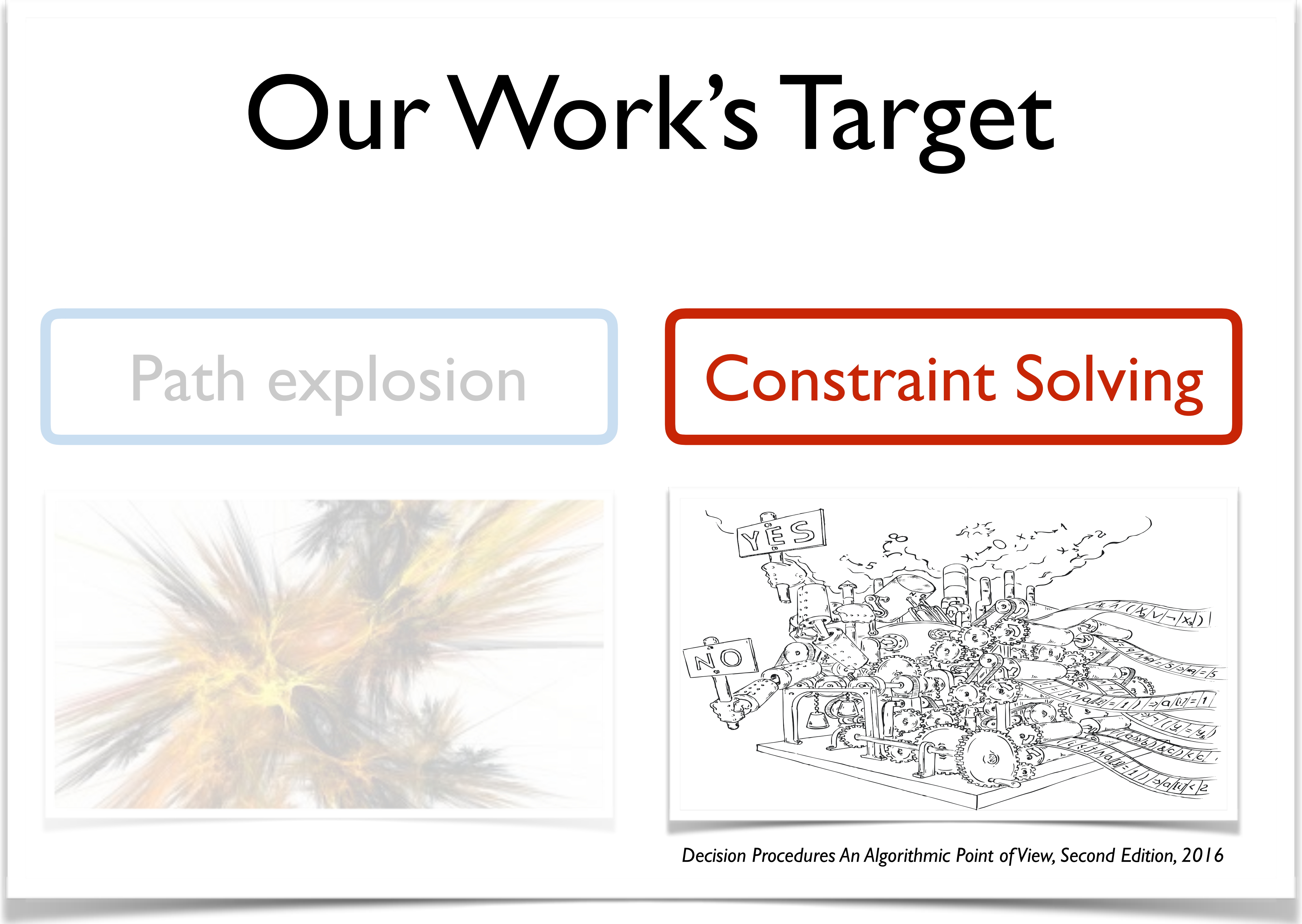
Discussion

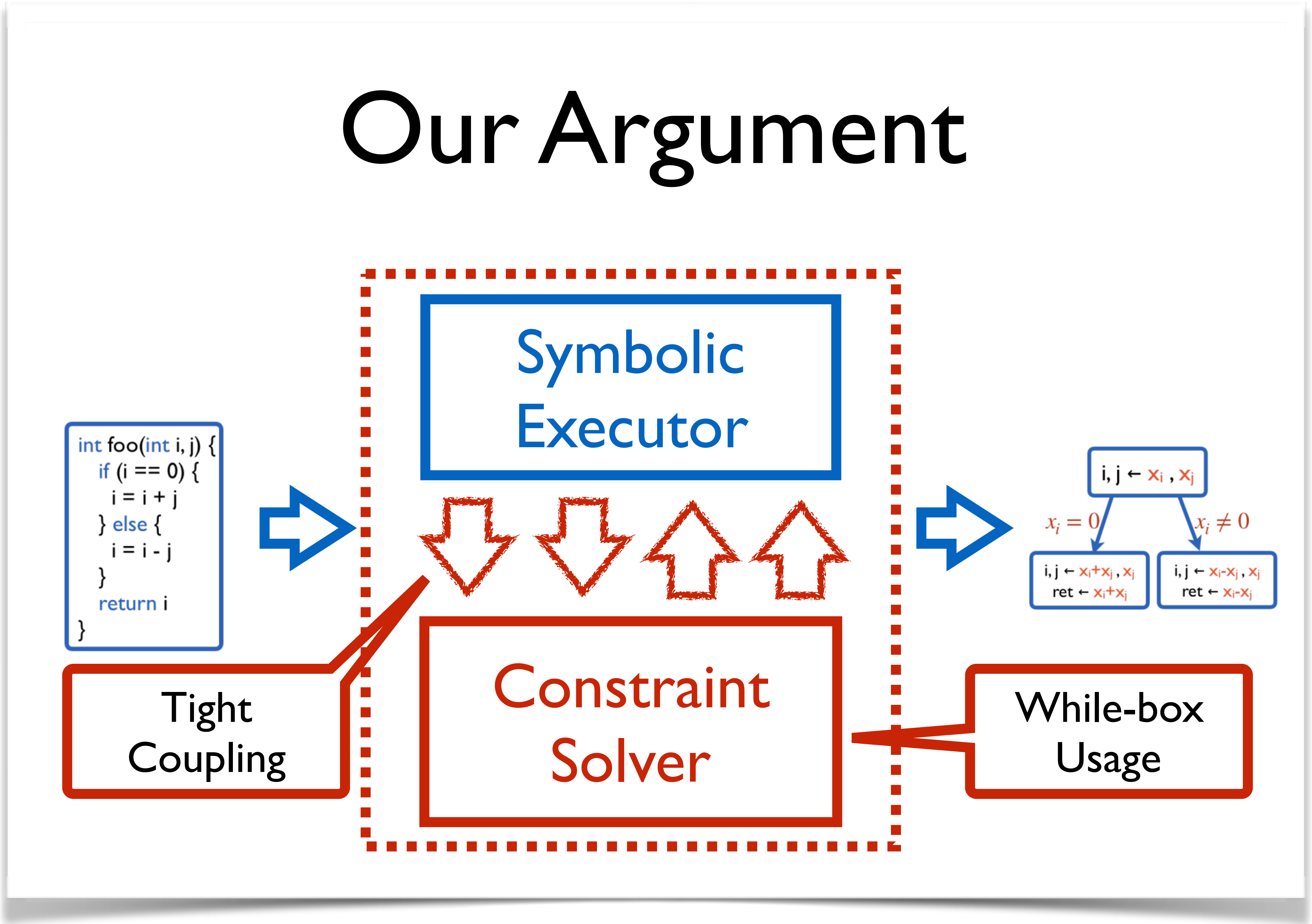
- Challenges
  - How to unify the explorations of the path space and the solution space?
  - How to sample the solving procedure?
  - …





Summary





- Type and Interval Aware Array Constraint Solving [ISSTA'21]
- Partial Solution Promoted Symbolic Execution [ASE'20]

Thank you!

Q&A

Zhenbang Chen

(zbchen@nudt.edu.cn)

Joint work with Ziqi Shuai, Yufeng Zhang, Zehua Chen, Guofeng Zhang, Jun Sun. Wei Dong and Ji Wang