Bounded Program Verification using an SMT Solver: A Case Study

Tianhai Liu, **Michael Nagel**, **Mana Taghdiri**

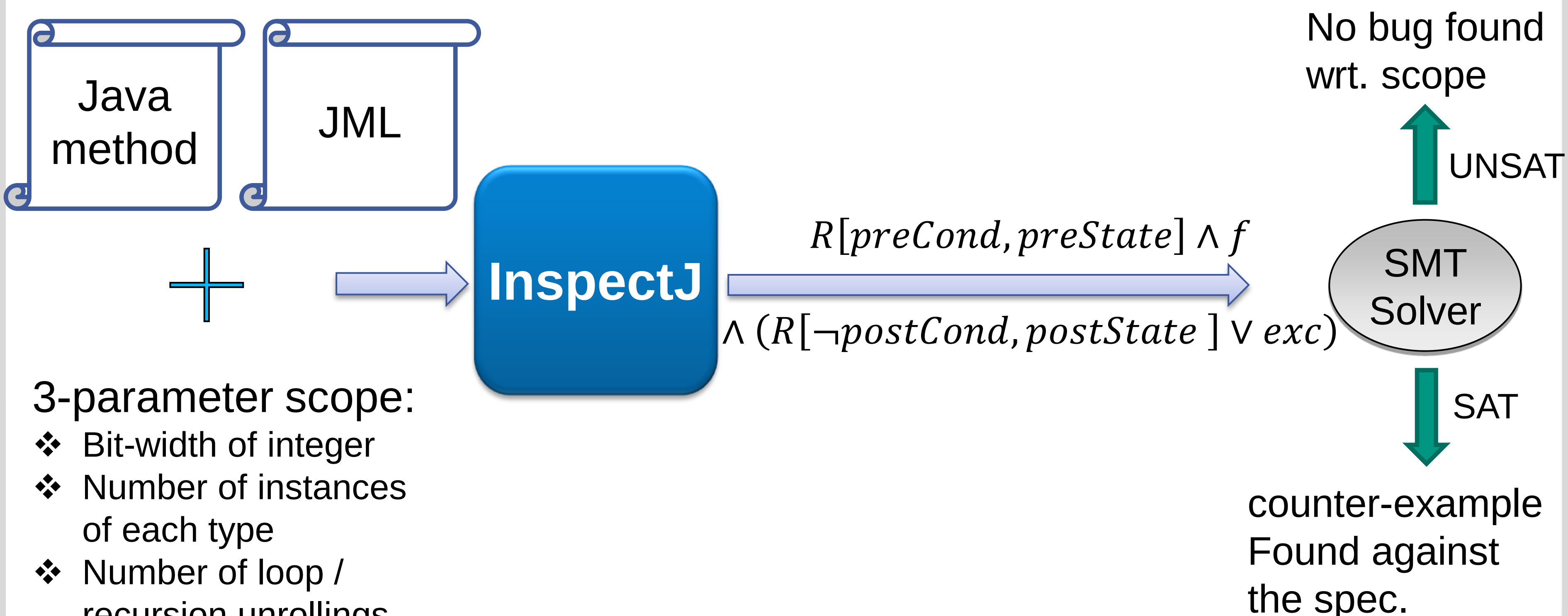
2012-04-18

AUTOMATED SOFTWARE ANALYSIS GROUP
INSTITUTE FOR THEORETICAL COMPUTER SCIENCE, DEPARTMENT OF INFORMATICS

Bounded Verification Tool: InspectJ

- Modular verification
  - Can check methods in isolation
- Rich data-structure properties of OO code
  - Arbitrarily complex object configurations in the heap
- Scalability
  - Target High-level simplications of QBVF solvers
- Usability
  - Fully automatic infrastructure
- Soundness
  - Error traces reported by InspectJ are real bugs
- Bounded completeness
  - If a bug exists wrt. bounds, InspectJ finds it
  - Only wrt. finite number of objects, and loop/recursion unrolling

Architecture



3-parameter scope:

- Bit-width of integer
- Number of instances of each type
- Number of loop / recursion unrollings

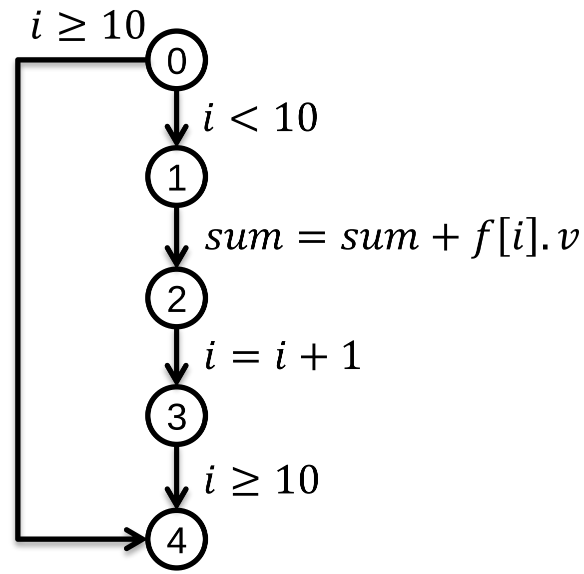
Target Logic

- Quantified bit-vector formulas (QBVF) with theory of arrays.
- QBVF were traditionally handled by flattening quantifiers using conjunctions and disjunctions.
- Recent QBVF solvers (e.g. Z3) perform several high-level simplifications before flattening quantifiers
  - skolemization
  - miniscoping
  - Rewriting
  - ... ➔ makes them more efficient!

Encoding Control Flow --- after 1 loop unrolling

Karlsruhe Institute of Technology

```
public class A {
 B[] f; int sum;
 void foo(int i){
  while(i<10){
   sum+=f[i].v;
   i++;
  }}}
class B{int v;}
```



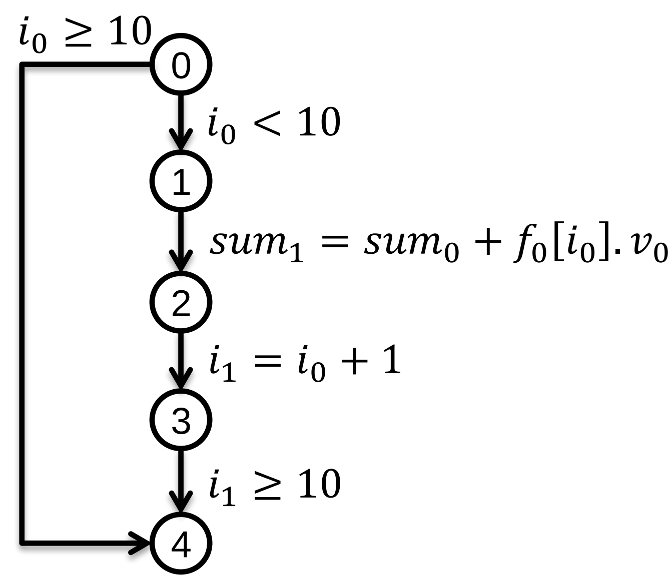
- Nodes labeled with numbers stand for states
- Edges stand for transitions or branches chosen
- CF is encoded with edge variables
  - e.g. $E_{0,1} \vee E_{0,4}$, $E_{0,1} \rightarrow E_{1,2}$
- Each edge variable is a predicate
- Predicates evaluation depends on stmt.
  - e.g. $E_{0,1} \rightarrow i < 10$

Encoding Control Flow --- after 1 loop unrolling

Karlsruhe Institute of Technology

```
public class A {
  B[] f; int sum;
  void foo(int i){
   while(i<10){
    sum+=f[i].v;
    i++;
   }}}
class B{int v;}
```

- Each variable (field, argument, local variable) is suffixed by a number **N**
- **N** means variable update times
- **N** starts from 0



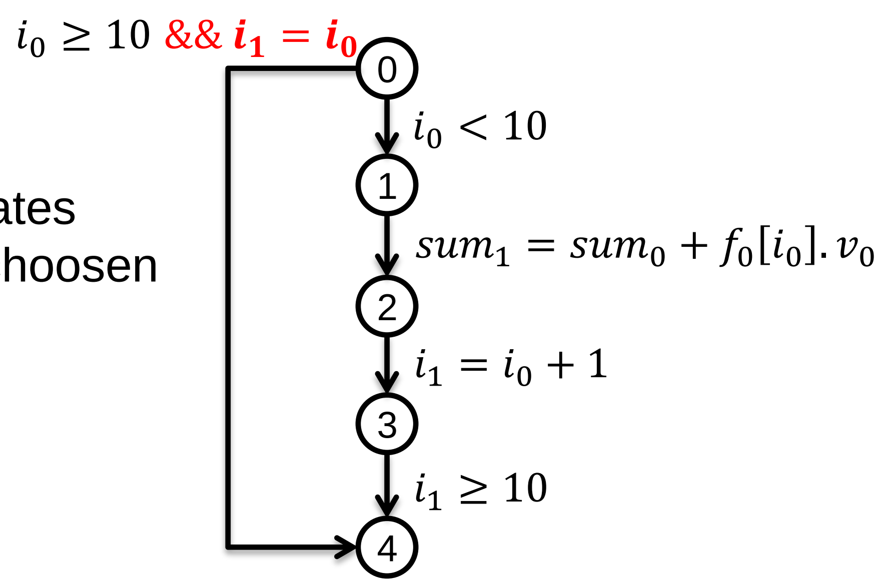
- Nodes labeled with numbers stand for states
- Edges stand for transitions or branches choosen
- CF is encoded with edge variables
  - e.g. $E_{0,1} \vee E_{0,4}$, $E_{0,1} \rightarrow E_{1,2}$
- Each edge variable is a predicate
- Predicates evaluation depends on stmt.
  - e.g. $E_{0,1} \rightarrow i_0 < 10$

Encoding Control Flow --- after 1 loop unrolling

Karlsruhe Institute of Technology

```
public class A {
 B[] f; int sum;
 void foo(int i){
  while(i<10){
   sum+=f[i].v;
   i++;
  }}}
class B{int v;}
```

- Each variable (field, argument, local variable) is suffixed by a number **N**
- **N** means variable update times
- **N** starts from 0
- Correct variable when in join nodes
  - e.g. $E_{0,4} \rightarrow i_0 \geq 10$ && $\boldsymbol{i_1 = i_0}$

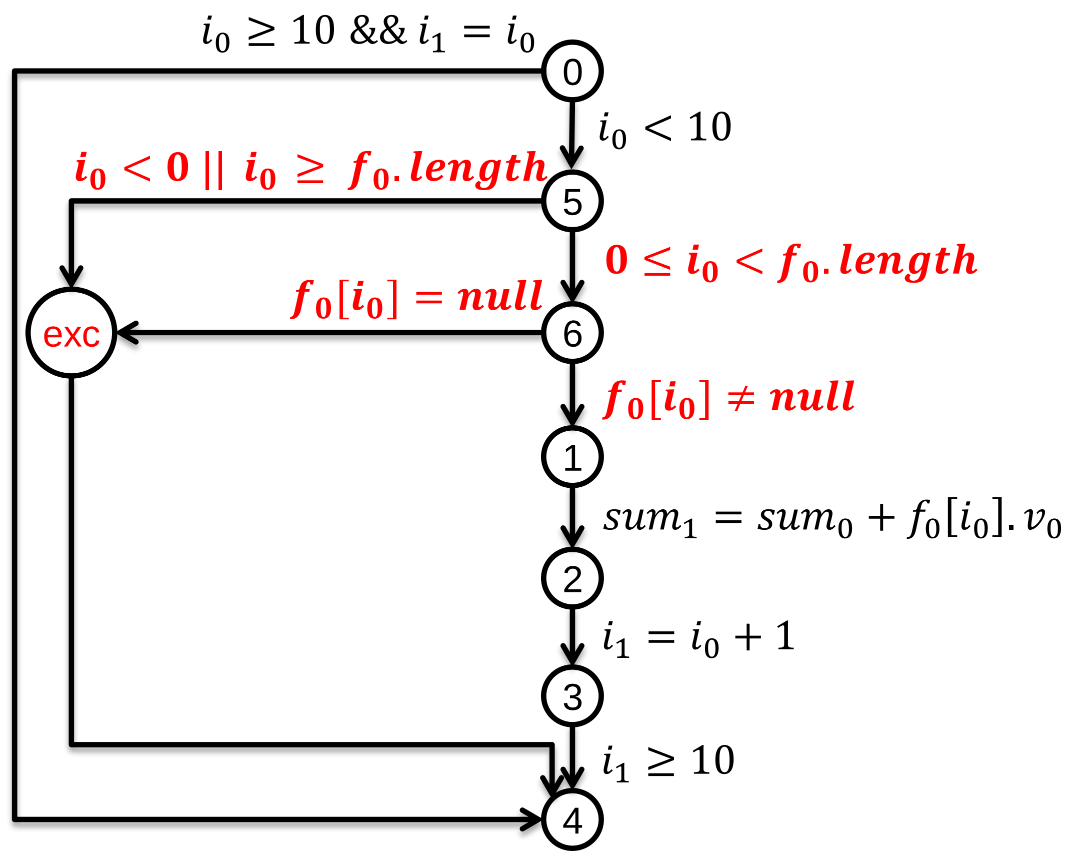


- Nodes labeled with numbers stand for states
- Edges stand for transitions or branches choosen
- CF is encoded with edge variables
  - e.g. $E_{0,1} \vee E_{0,4}$, $E_{0,1} \rightarrow E_{1,2}$
- Each edge variable is a predicate
- Predicates evaluation depends on stmt.
  - e.g. $E_{0,1} \rightarrow i_0 < 10$

Exceptions

```
class A {
 B[] f; int sum;
 void foo(int i){
  while(i<10){
   sum+=f[i].v;
   i++;
  }}}
class B{int v;}
```

Exceptions will be caught by an ***exc*** node



Encoding Classes

- Instances are bounded
- Given a bound n for a class A
  - A encoded as $(\text{define−sort A () (\_ BitVec m))}$, $m = \lceil \log(n+1) \rceil$
  - Not all values represent instances
    - value 0 stands for Java $null$, denoted by $nullA$
    - values belonging to $(n, 2^m]$ are ignored.

0	$\cdots$	n	$\cdots$	2^m

Encoding Classes (cont.)

- How to achieve bounded completeness
  - no bug exists within a bound n implies no bug exists in any bounds less than n.
- an index $\mathrm{id}xA$ is introduced to represent the last allocated object, $idxA \in [0, n]$.

0	$\cdots$	$idxA$	$\cdots$	n	$\cdots$	2^m

Encoding Classes (cont.)

Karlsruhe Institute of Technology

- in pre-state, valid range of A is $[0, idxA_0]$
- in post-state, valid range of A is $[0, idxA']$
- translation of allocation statement „A a = new A();"
  - (assert (and

$$
\begin{array}{l}
(= \mathrm{idx}A_{i+1}\ (bvadd\ idxA_i\ (\_\ bv1\ m))) \\
(= a\ idxA_{i+1}) \\
(\mathrm{bvuge}\ \mathrm{idx}A_{i+1}\ idxA_i) \\
(\mathrm{bvuge}\ \mathrm{idx}A_{i+1}\ (\_\ bv1\ m)) \\
))
\end{array}
$$

Encoding Fields

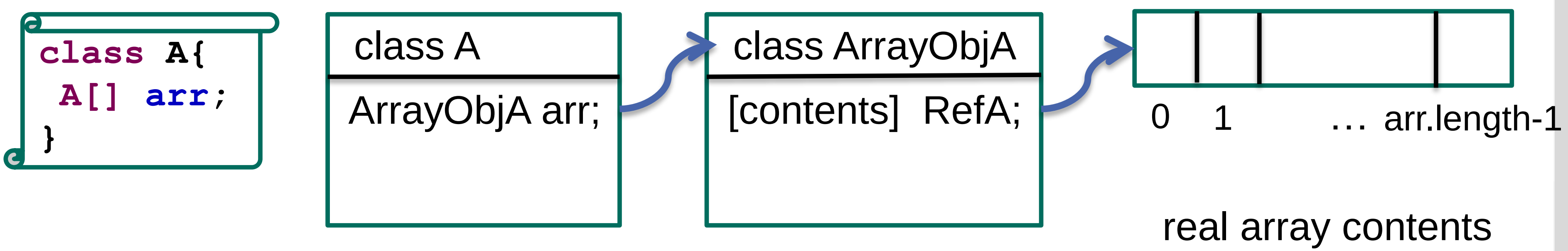
- Encoded as arrays over bit-vectors
  - (declare−fun f () (Array A B))
- Using theory of array
  - Read o.f : (select f o)
  - Write o.f = b : (store f o b)

```
class A {
    B f;
}
```

- Values of all fields must be valid in pre-state
  - (assert (forall (x A)
    (=> (and (not (= x nullA)) (bvule x idxA))
    (bvule (select f_0 x) idxB))))

Encoding Arrays

- array objects of type $A[\,]$ are encoded by introducing a new type $ArrayObjA$ and a reference $RefA$ from $ArrayObjA$ to their contents.
  - (define−sort ArrayObjA (_ BitVec t))
  - (declare−fun RefA () (Array ArrayObjA (Array integer A)))



Encoding Arrays --- bitwidth 5, instance 3

(define-sort int () (_ BitVec 5))
(define-sort A () (_ BitVec 2))
(define-sort ArrayObjA () (_ BitVec 2))

```
class A {
 A[] arr;
 void foo(){
  A elem = arr[0];
  int len = arr.length
 }
}
```

Define types

Encoding Arrays --- bitwidth 5, instance 3

```
class A {
 A[] arr;
 void foo(){
  A elem = arr[0];
  int len = arr.length
 }
}
```

(define-sort int () (_ BitVec 5))
(define-sort A () (_ BitVec 2))
(define-sort ArrayObjA () (_ BitVec 2))

(declare-fun this () A)
(declare-fun elem () A)
(declare-fun len () int)

Define local variables

Encoding Arrays --- bitwidth 5, instance 3

```
class A {
 A[] arr;
 void foo(){
  A elem = arr[0];
  int len = arr.length
 }
}
```

(define-sort int () (_ BitVec 5))
(define-sort A () (_ BitVec 2))
(define-sort ArrayObjA () (_ BitVec 2))

(declare-fun this () A)
(declare-fun elem () A)
(declare-fun len () int)

(declare-fun arr (A) ArrayObjA)
(declare-fun RefA (ArrayObjA) (Array int A))
(assert (= elem
  (select (select RefA (select arr this)) (_ bv0 5)))

Define array fields and access array

Encoding Arrays --- bitwidth 5, instance 3

```
class A {
 A[] arr;
 void foo(){
  A elem = arr[0];
  int len = arr.length
 }
}
```

```
(define-sort int () (_ BitVec 5))
(define-sort A () (_ BitVec 2))
(define-sort ArrayObjA () (_ BitVec 2))

(declare-fun this () A)
(declare-fun elem () A)
(declare-fun len () int)

(declare-fun arr (A) ArrayObjA)
(declare-fun RefA (ArrayObjA) (Array int A))
(assert (= elem
   (select (select RefA (select arr this)) (_ bv0 5)))

(declare-fun length () (Array ArrayObjA int))
(assert (= len
   (select length (select arr this))))
```

Define array length

Encoding JML Specifications

- Standard JML plus the \reach clause
- Simply transform to FOL formulas except...
  - Constraint variables of a reference type A must be in A's instance range.

```
class A{
    B f;
    //@ invariants \forall o A; o.f == null;
    void foo(){}
}
```

transform

```
(assert (forall ((o A))  (=> (and (not (= o nullA)) (bvule o idxA))
                              (= (select f o) nullA))))
```

Reachability

- expressed as $\text{\reach}(x, T, f)$
- Generally Transitive Closure encoded as (inspired by Claessen)

  1) $\forall x, y. xRy \Leftrightarrow P(x, y) = 1$

Reachability

- expressed as $\text{\reach}(x, T, f)$
- Generally Transitive Closure encoded as (inspired by Claessen)
  1) $\forall x, y. xRy \Leftrightarrow P(x, y) = 1$
  2) $\forall x, y, z. P(x, y) > 0 \,\&\&\, P(x, z) > 0 \Rightarrow P(x, z) > 0$

Reachability

- expressed as $\text{\reach}(x, T, f)$
- Generally Transitive Closure encoded as (inspired by Claessen)
  1) $\forall x, y.\, xRy \Leftrightarrow P(x, y) = 1$
  2) $\forall x, y, z.\, P(x, y) > 0 \;\&\&\; P(x, z) > 0 \Rightarrow P(x, z) > 0$
  3) $\forall x, y.\, P(x, y) > 1 \Rightarrow \exists \mathrm{w}.\, (P(x, w) = 1 \;\&\&\; P(x, y) = P(w, y) + 1)$

Reachability

- expressed as $\backslash\mathrm{reach}(x, T, f)$
- Generally Transitive Closure encoded as (inspired by Claessen)
  1) $\forall x, y. xRy \Leftrightarrow P(x, y) = 1$
  2) $\forall x, y, z. P(x, y) > 0 \;\&\&\; P(x, z) > 0 \Rightarrow P(x, z) > 0$
  3) $\forall x, y. P(x, y) > 1 \Rightarrow \exists \mathrm{w}. (P(x, w) = 1 \;\&\&\; P(x, y) = P(w, y) + 1)$
- Additional constraints in Java context
  1) $\forall x. P(null, x) = 0$

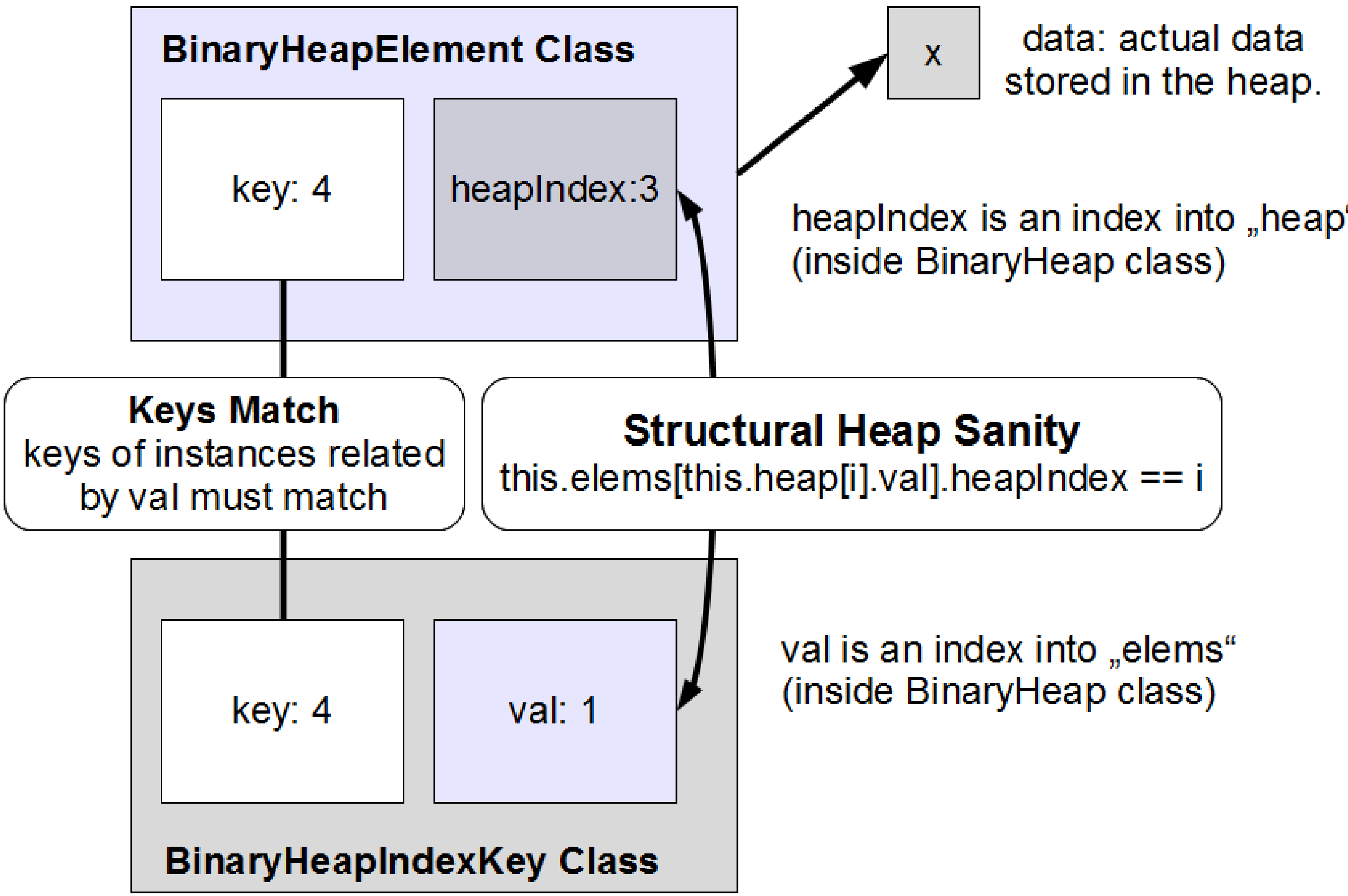
Reachability

- expressed as $\backslash\text{reach}(x, T, f)$
- Generally Transitive Closure encoded as (inspired by Claessen)
    1) $\forall x, y.\, xRy \Leftrightarrow P(x, y) = 1$
    2) $\forall x, y, z.\, P(x, y) > 0 \;\&\&\; P(x, z) > 0 \Rightarrow P(x, z) > 0$
    3) $\forall x, y.\, P(x, y) > 1 \Rightarrow \exists \text{w}.\, (P(x, w) = 1 \;\&\&\; P(x, y) = P(w, y) + 1)$
- Additional constraints in Java context
    1) $\forall x.\, P(null, x) = 0$
    2) $\forall x.\, xRx \Rightarrow \forall y.\, (x \neq y) \Rightarrow (P(x, y) = 0)$

Evaluation Benchmark

- Dijkstra algorithem implemented using BinaryHeap data structure in Java
  - 7 classes
  - 346 Java source lines
  - 37 methods
  - 27 lines of JML specification, which checks binary heap data structure internal intergrity.
  - runtime compared with JForge

Properties Checked



Bugs found

Karlsruhe Institute of Technology

Copy by reference bug

```
/*@ invariant
 @(\forall int i; i >= 0 && i < this.heap.len
 @ ==> this.elems[this.heap[i].val].key ==
 @ this.heap[i].key)
 @*/
 // VERSION WITH BUG
 heap[index2] = heap[index1];
 heap[index2].key = k;

 // VERSION WITHOUT BUG
 heap[index2].key = heap[index1].key;
 heap[index2].val = heap[index1].val;
 heap[index2].key = k;
```

null pointer dereference

```
// VERSION WITH BUG
 this.dropHeap();
 x = heap[1];
 ....

// VERSION WITHOUT BUG
 x = heap[1];
 this.dropHeap();
 ....
```

Runtime Evaluation Results

Method	Bit	Obj	Loop	JForge				InspectJ			
				PrePro.	Z3	Total	Result	Result	PrePro.	Z3	Total
decreaseKey	3	3	3	0.6	61.8	62.4	unsat	unsat	1.5	0.4	**1.9**
	4	4	4	0.7	82.5	83.2	unsat	unsat	1.5	8.7	**10.3**
	5	5	5	1.8	TO	TO	-	unsat	1.5	31.3	**32.8**
	7	7	6	66.0	TO	TO	-	unsat	1.6	507.5	**509.1**
deleteMin	3	3	3	0.5	0.6	**1.1**	unsat	unsat	1.7	0.2	1.9
	4	4	4	1.5	36.4	37.9	unsat	unsat	1.7	3.4	**5.0**
	5	5	5	4.8	TO	TO	-	unsat	1.7	52.5	**54.2**
	6	6	6	29.5	TO	TO	-	unsat	1.7	133.4	**135.1**
insert	3	3	3	0.5	0.5	**1.0**	unsat	unsat	1.6	0.4	1.9
	4	4	4	1.5	14.8	15.6	unsat	unsat	1.6	5.4	**7.0**
	5	5	5	2.1	409.8	411.9	unsat	unsat	1.6	86.8	**88.4**
	6	6	6	11.3	TO	TO	-	unsat	1.6	110.0	**111.6**
minElement	4	4	4	0.5	0.2	**0.7**	unsat	unsat	1.4	0.0	1.4
	7	7	7	49.5	16.6	66.1	unsat	unsat	1.4	0.0	**1.4**
	8	8	8	TO	-	-	-	unsat	1.4	0.0	**1.4**
run	3	3	1	9.6	2.2	11.8	sat	sat	3.2	0.7	**3.9**
	4	4	1	16.7	4.3	21.0	sat	sat	3.2	6.9	**10.0**
	7	7	1	371.1	299.0	TO	-	sat	3.2	2.4	**5.6**
	3	3	2	TO	-	-	-	sat	5.0	52.7	**57.7**

SMT-based program checking

- ESC/Java, ESC/Java2
  - Unrolling loops bounded only
  - Undecidable target logics
- Armando et al.[09], Cordeiro et al. [09], Ganai et al. [06], Sinz et al. [10] and LAV
  - Quantifier-free target logics
  - Check finite-state-machine properties
  - No data-structure properties checked
- Boogie
  - Undecidable target logics
  - Loop invariants required
  - Spurious counterexamples

Rich-Data-Structure checkings

- Bounded verification approaches
  - SAT solver used and fully bounded
    - JAlloy, JForge, TACO, Miniatur, Karun and MemSAT
  - SMT solver used and only loops are bounded
    - ESC/Java and ESC/Java2
  - Dynamic checking with bounded heap
    - TestEra and Korat
  - Java PathFinder + Korat
- Deductive verification
  - Key, LOOP

Conclusion

- Main contribution
  - First attempt to use SMT solver on bounded data-structure-rich program verification.
  - Present a translation from subset of Java to QBVF with theory of arrays.
- Future
  - incorporating optimizations to reduce the burden of the underlying solver
  - finding relationship between the number of objects and loop unrollings