Building SMT-based Software Model Checkers: an Experience Report

Alessandro Armando

Artificial Intelligence Laboratory (AI-Lab)
Dipartimento di Informatica Sistemistica e Telematica (DIST)
University of Genova

FroCoS 2009
Trento, September 18, 2009

Outline

Software Model Checking

- Fully automatic technique for reasoning about programs
- Confluence of techniques from different fields:
  - Model checking
  - Program analysis
  - Automated theorem proving

Model Checking

- Algorithmic exploration of state space of the system
- Several advances in the past decade: symbolic model checking, symmetry reductions, partial order reductions, bounded model checking using SAT solvers
- Used by HW companies in the validation cycle
- *Strengths*
  - Rich property languages
  - Provides counterexamples
- *Weaknesses*
  - Mostly limited to finite state systems
    (but, extensions to infinite state systems exist)
  - Operates only on models, but

how do you get the model?

Model Checking

- Algorithmic exploration of state space of the system
- Several advances in the past decade: symbolic model checking, symmetry reductions, partial order reductions, bounded model checking using SAT solvers
- Used by HW companies in the validation cycle
- *Strengths*
  - Rich property languages
  - Provides counterexamples
- *Weaknesses*
  - Mostly limited to finite state systems
    (but, extensions to infinite state systems exist)
  - Operates only on models, but

how do you get the model?

Program Analysis

- Originated in optimizing compilers: constant propagation, live variable analysis, dead code elimination, loop index optimization.
- *Strengths*
  - Works on code
  - Precision efficiency tradeoffs well studied
  - Pointer aware
- *Weaknesses*
  - Abstraction hardwired
  - Property hardwired

Automated Theorem Proving

Strengths

- Handles unbounded domains naturally
- Good satisfiability procedures for
  - equality with uninterpreted function symbols
  - linear arithmetics
  - theory of arrays
  - the combination of the above
- Good integration of satisfiability procedures with modern SAT-solvers

Weaknesses

- Hard to compute fixpoints
- Requires inductive invariants
- Pre and post conditions

Model Checking, Program Analysis and Automated Theorem Proving

- Very related to each other
- Different histories
  - different emphasis
  - different tradeoffs
- Complementary, in some ways
- *Combination can be extremely powerful*

Modeling Hardware vs. Modeling Software

Hardware

- Primitive values are booleans
- States are boolean vectors of fixed size
- Modeled as finite state machines

Software

- Primitive values are more complicated
  - Floats, Pointers, Objects
- Functions
  - Function pointers, Exceptions
- States are more complicated
  - Unbounded graphs over values
- Variables are scoped
  - Locals, Shared scopes
- Much richer modularity constructs
  - Functions, Classes

My Experience

- SMT-based Bounded Model Checking
  - Bounded reachability analysis based on a reduction to SMT
  - Conceptually simple
  - Competes with SAT-based approach
- SMT-based CounterExample-Guided Abstraction Refinement (CEGAR)
  - Symbolic analysis on an abstraction of the program
  - SMT used to detect spurious execution traces and refine the abstraction
  - Superior to SW model checkers based on predicate abstraction on programs with arrays

Outline

Bounded Model Checking of Software

Bounded Model Checking [BCCZ99]

Bounded reachability problem of a system S reduced to the *satisfiability problem* of a propositional formula Φ_S^k, for $k \geq 0$.

SAT-based Bounded Model Checking of Software [KCY03]

Bounded reachability problem of a program P reduced to the *satisfiability problem* of a propositional formula Φ_P^k, for $k \geq 0$.

SMT-based Bounded Model Checking of Software [AMP06]

Bounded reachability problem of a program P reduced to the *satisfiability problem* of a quantifier-free formula Φ_P^k w.r.t. a decidable theory $\mathcal{T}$, for $k \geq 0$.

$\mathcal{T}$ axiomatizes the properties of the data structures manipulated by P (e.g. linear arithmetics, theory of arrays, or the combination thereof).

Bounded Model Checking of Software

Bounded Model Checking [BCCZ99]

Bounded reachability problem of a system S reduced to the *satisfiability problem* of a propositional formula Φ_S^k, for $k \geq 0$.

SAT-based Bounded Model Checking of Software [KCY03]

Bounded reachability problem of a program P reduced to the *satisfiability problem* of a propositional formula Φ_P^k, for $k \geq 0$.

SMT-based Bounded Model Checking of Software [AMP06]

Bounded reachability problem of a program P reduced to the *satisfiability problem* of a quantifier-free formula Φ_P^k w.r.t. a decidable theory $\mathcal{T}$, for $k \geq 0$.

$\mathcal{T}$ axiomatizes the properties of the data structures manipulated by P (e.g. linear arithmetics, theory of arrays, or the combination thereof).

Bounded Model Checking of Software

Bounded Model Checking [BCCZ99]

Bounded reachability problem of a system S reduced to the *satisfiability problem* of a propositional formula Φ_S^k, for $k \geq 0$.

SAT-based Bounded Model Checking of Software [KCY03]

Bounded reachability problem of a program P reduced to the *satisfiability problem* of a propositional formula Φ_P^k, for $k \geq 0$.

SMT-based Bounded Model Checking of Software [AMP06]

Bounded reachability problem of a program P reduced to the *satisfiability problem* of a quantifier-free formula Φ_P^k w.r.t. a decidable theory $\mathcal{T}$, for $k \geq 0$.

$\mathcal{T}$ axiomatizes the properties of the data structures manipulated by P (e.g. linear arithmetics, theory of arrays, or the combination thereof).

Bounded Model Checking of Software

Let k be a positive integer. BMC of a program P can be done by:

1. preprocessing P:
   - removing all control statements except `while`, `if`, and `assert`,
   - unwinding `while` loops k times,
   - removing side effects by putting P in SSA Form;
2. generating the quantifier-free formula Φ_P;
3. deciding the satisfiability of Φ_P w.r.t. $\mathcal{T}$.

Bounded Model Checking of Software

Let k be a positive integer. BMC of a program P can be done by:

1. preprocessing P:
   - removing all control statements except `while`, `if`, and `assert`,
   - unwinding `while` loops k times,
   - removing side effects by putting P in SSA Form;
2. generating the quantifier-free formula Φ_P;
3. deciding the satisfiability of Φ_P w.r.t. $\mathcal{T}$.

Bounded Model Checking of Software

Let k be a positive integer. BMC of a program P can be done by:

1. preprocessing P:
   - removing all control statements except `while`, `if`, and `assert`,
   - unwinding `while` loops k times,
   - removing side effects by putting P in SSA Form;
2. generating the quantifier-free formula Φ_P;
3. deciding the satisfiability of Φ_P w.r.t. $\mathcal{T}$.

Bounded Model Checking of Software

Let k be a positive integer. BMC of a program P can be done by:

1. preprocessing P:
   - removing all control statements except `while`, `if`, and `assert`,
   - unwinding `while` loops k times,
   - removing side effects by putting P in SSA Form;
2. generating the quantifier-free formula Φ_P;
3. deciding the satisfiability of Φ_P w.r.t. $\mathcal{T}$.

Bounded Model Checking of Software

Let k be a positive integer. BMC of a program P can be done by:

1. preprocessing P:
   - removing all control statements except **`while`**, **`if`**, and **`assert`**,
   - unwinding **`while`** loops k times,
   - removing side effects by putting P in SSA Form;
2. generating the quantifier-free formula Φ_P;
3. deciding the satisfiability of Φ_P w.r.t. $\mathcal{T}$.

Bounded Model Checking of Software

Let k be a positive integer. BMC of a program P can be done by:

1. preprocessing P:
   - removing all control statements except **`while`**, **`if`**, and **`assert`**,
   - unwinding **`while`** loops k times,
   - removing side effects by putting P in SSA Form;
2. generating the quantifier-free formula Φ_P;
3. deciding the satisfiability of Φ_P w.r.t. $\mathcal{T}$.

Bounded Model Checking of Software

Let k be a positive integer. BMC of a program P can be done by:

1. preprocessing P:
   - removing all control statements except **`while`**, **`if`**, and **`assert`**,
   - unwinding **`while`** loops k times,
   - removing side effects by putting P in SSA Form;
2. generating the quantifier-free formula Φ_P;
3. deciding the satisfiability of Φ_P w.r.t. $\mathcal{T}$.

Preprocessing: Unwinding the Loops

Every loop is replaced by a sequence of k nested **if** statements:

```
i=0;
while(i<2) {
 a[i]=i;
 i=i+1;
}
```

$\xrightarrow{k=2}$

```
i=0;
if (i<2) {
  a[i]=i;
  i=i+1;
  if (i<2) {
    a[i]=i;
    i=i+1
    assert(!(i<2));
    }
  }
```

Preprocessing: Unwinding the Loops

Every loop is replaced by a sequence of k nested **if** statements:

```
i=0;
while(i<2) {
  a[i]=i;
  i=i+1;
}
```

$\xrightarrow{k=2}$

```
i=0;
if (i<2) {
   a[i]=i;
   i=i+1;
   if (i<2) {
      a[i]=i;
      i=i+1
      assert(!(i<2));
      }
   }
```

Unwinding Assertion $\Longrightarrow$

Turning the Program in SSA Form

Side effects are removed and variables renamed so to make them assigned at most once (SSA = Static Single Assignment):

```
i=a[0];
i=i+1;
if(x>0) {
 if(x<10)
  x=x+1;
 else
  x=x-1;
}
assert(x>0 && x<10);
a[x]=i;
```

$\Longrightarrow$

```
i1=a0[0];
i2=i1+1;
if(x0>0) {
  if(x0<10)
    x1=x0+1;
  else
    x2=x0-1;
  x3=(x0<10?x1:x2);
}
x4=(x0>0?x3:x0);
assert(x4>0 && x4<10);
a1=store(a0,x4,i2);
```

From SSA to Conditional Normal Form

```
i1=a0[0];
i2=i1+1;
if(x0>0) {
  if(x0<10)
    x1=x0+1;
  else
    x2=x0-1;
  x3=(x0<10?x1:x2);
}
x4=(x0>0?x3:x0);
assert(x4>0 && x4<10);
a1=store(a0,x4,i2);
```

$\Longrightarrow$

```
if(true) i1=a0[0];
if(true) i2=i1+1;
if(x0>0 && x0<10) x1=x0+1;
if(x0>0 && !(x0<10)) x2=x0-1;
if(x0>0 && x0<10) x3=x1;
if(x0>0 && !(x0<10)) x3=x2;
if(x0>0) x4=x3;
if(!(x0>0)) x4=x0;
if(true) assert(x4>0 && x4<10);
if(true) a1=store(a0,x4,i2);
```

Generating the formula

Each statement is finally turned into a quantifier-free formula:

```
if(true) i1=a0[0];
if(true) i2=i1+1;
if(x0>0 && x0<10) x1=x0+1;
if(x0>0 && !(x0<10)) x2=x0-1;
if(x0>0 && x0<10) x3=x1;
if(x0>0 && !(x0<10)) x3=x2;
if(x0>0) x4=x3;
if(!(x0>0)) x4=x0;
if(true) assert(x4>0 && x4<10);
if(true) a1=store(a0,x4,i2);
```

$\Longrightarrow$

$$\begin{aligned}
\text{TRUE} &\Rightarrow i^1 = \text{select}(a^0, 0) \\
\text{TRUE} &\Rightarrow i^2 = i^1 + 1 \\
(x^0 > 0 \wedge x^0 < 10) &\Rightarrow x^1 = x^0 + 1 \\
(x^0 > 0 \wedge \neg(x^0 < 10)) &\Rightarrow x^2 = x^0 - 1 \\
(x^0 > 0 \wedge x^0 < 10) &\Rightarrow x^3 = x^1 \\
(x^0 > 0 \wedge \neg(x^0 < 10)) &\Rightarrow x^3 = x^2 \\
x^0 > 0 &\Rightarrow x^4 = x^3 \\
\neg(x^0 > 0) &\Rightarrow x^4 = x^0 \\
\text{TRUE} &\Rightarrow (x^4 > 0 \wedge x^4 < 10) \\
\text{TRUE} &\Rightarrow a^1 = \text{store}(a^0, x^4, i^2)
\end{aligned}$$

Proposition

The models of $\mathcal{T} \cup \mathcal{C}_P \cup \{\neg \bigwedge \mathcal{P}_P\}$ (if any) correspond to the execution paths of P that lead to an assertion violation.

SAT- vs SMT-based Bounded Model Checking of SW

SAT-based BMC of SW

- variables of numeric type (e.g. `int`, `float`) are modelled as vectors of bits of appropriate size,
- each element of an array is treated as a different variable; hence $n \times \dim(a)$ bits per array

State-of-the-art SAT solver used to decide the resulting satisfiability problem.

SMT-based BMC of SW

- variables of numeric type (e.g. `int`, `float`) can be modeled as bit-vectors, integers, or reals;
- arrays of m elements are modelled as... arrays!

State-of-the-art SMT solver used to decide the resulting satisfiability problem.

SAT- vs SMT-based Bounded Model Checking of SW

SAT-based BMC of SW

- variables of numeric type (e.g. **`int`**, **`float`**) are modelled as vectors of bits of appropriate size,
- each element of an array is treated as a different variable; hence $n \times \dim(a)$ bits per array

State-of-the-art SAT solver used to decide the resulting satisfiability problem.

SMT-based BMC of SW

- variables of numeric type (e.g. **`int`**, **`float`**) can be modeled as bit-vectors, integers, or reals;
- arrays of m elements are modelled as... arrays!

State-of-the-art SMT solver used to decide the resulting satisfiability problem.

SMT vs. SAT (1)

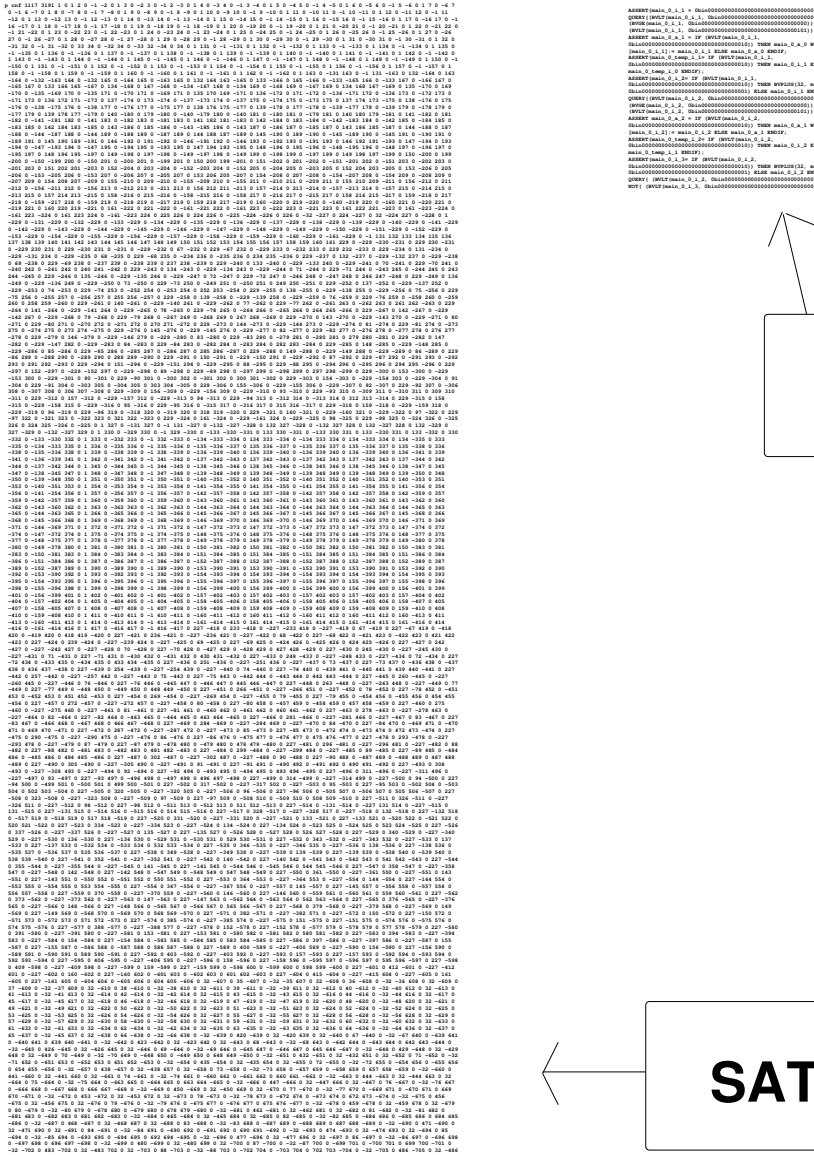
The size of the SAT encoding depends on the size of the arrays occurring in the program:

SMT	SAT
$a_k = store(a_{k-1}, e_1, e_2)$	$\bigwedge_{i=0}^{dim(a)-1} a_k^i = ((i = e_1)\ ?\ e_2 : a_{k-1}^i)$
$x_j = select(a_k, e)$	$\bigwedge_{i=0}^{dim(a)-1} (i = e \Rightarrow x_j = a_k^i)$

where $v = (c\ ?\ e_1 : e_2)$ abbreviates the formula $((c \Rightarrow v = e_1) \wedge (\neg c \Rightarrow v = e_2)$.

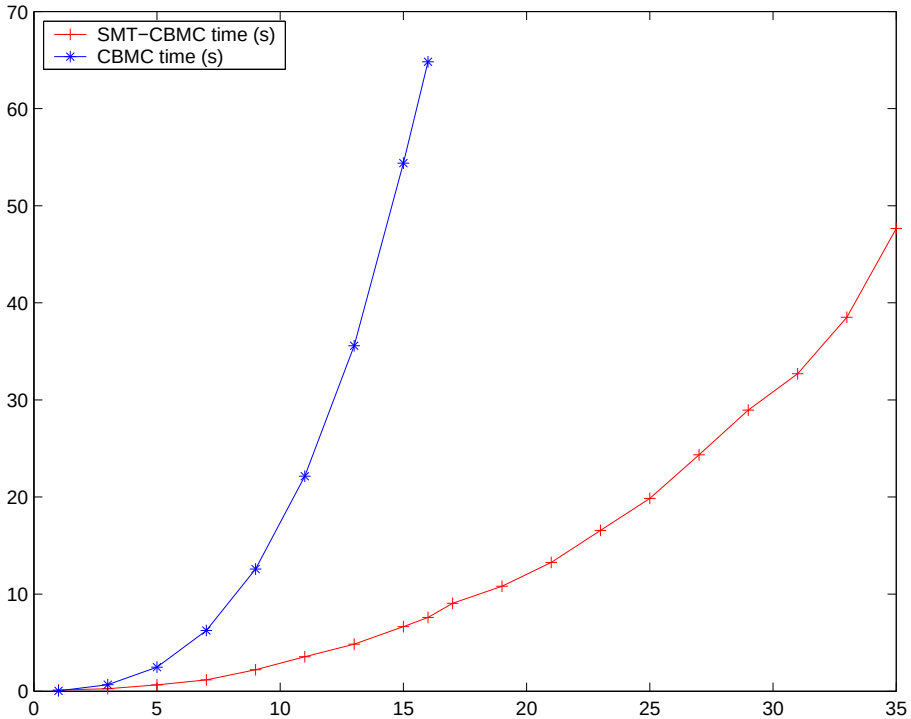
SMT vs. SAT (3)

```
for(i=0;i<2;i++)
  a[i]=i;
```

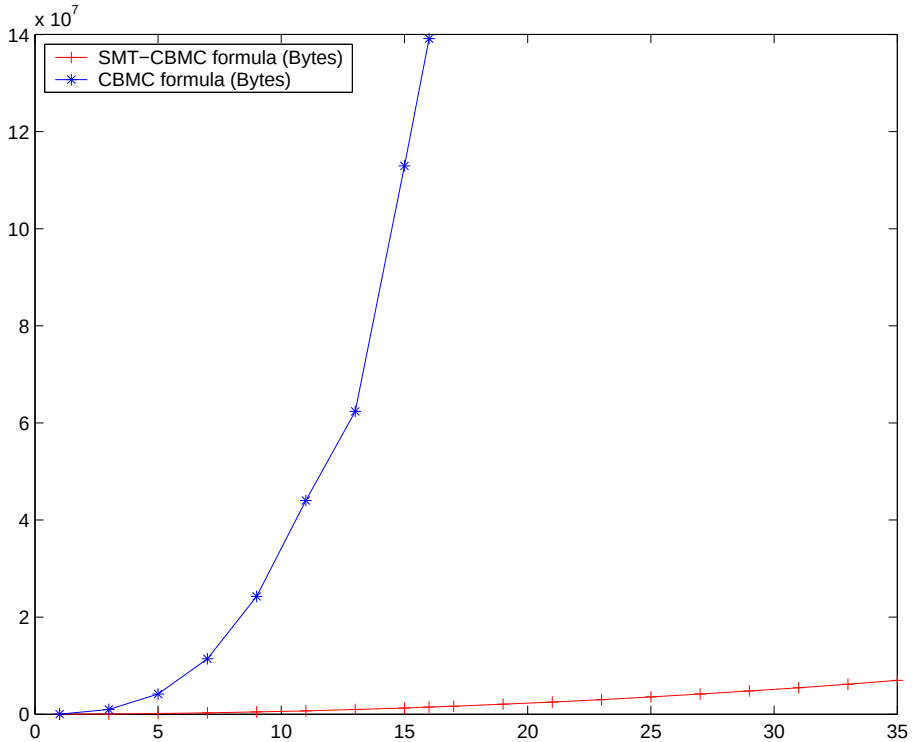


Experimental Results: SMT-CBMC vs. CBMC

Bubblesort(N)



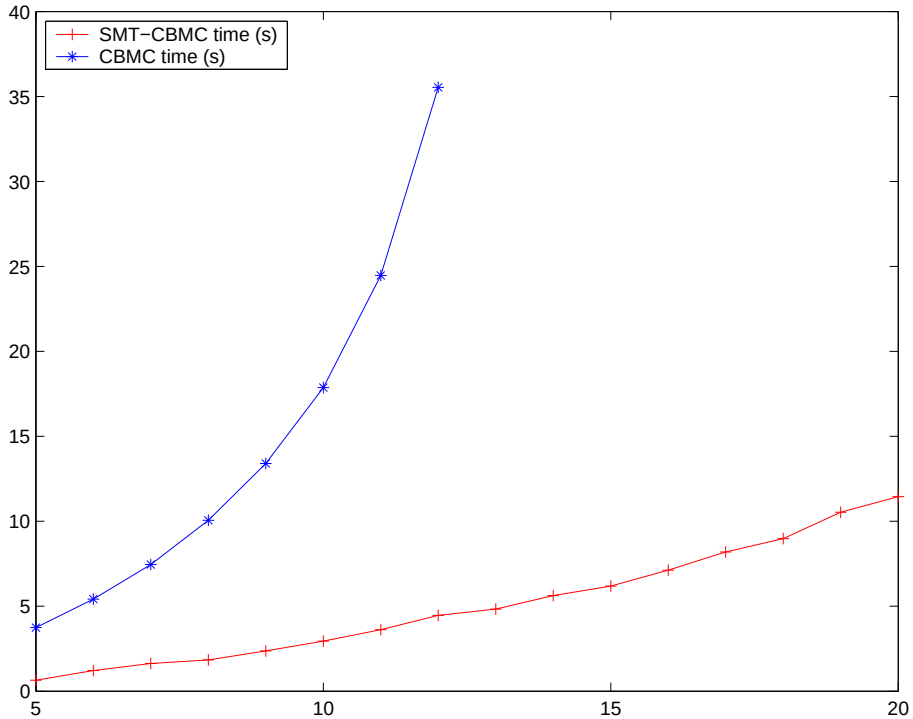
Overall time (s)



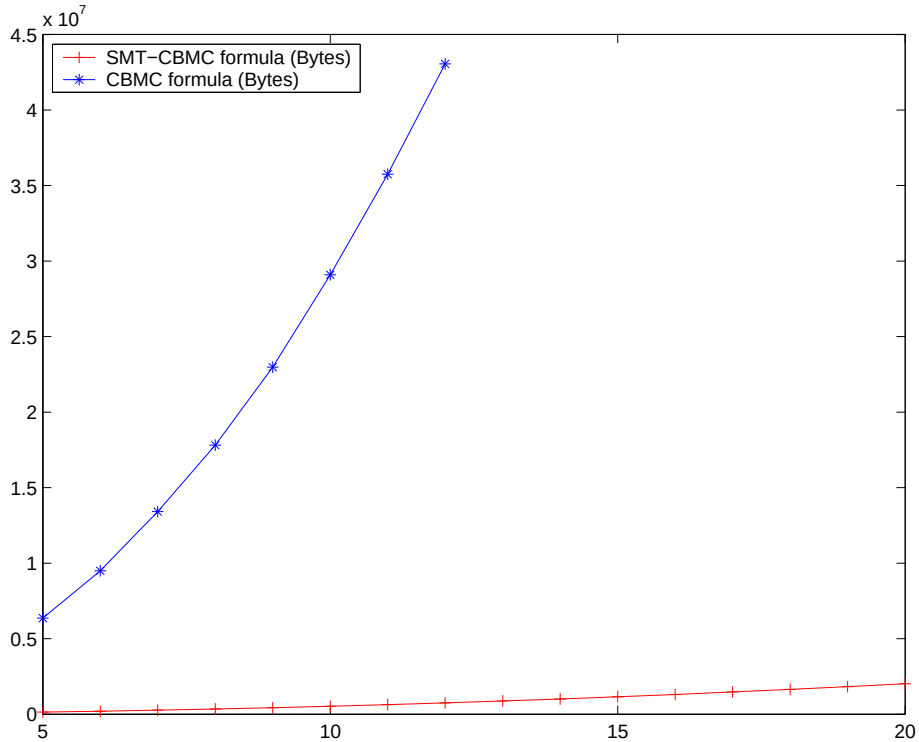
Size of the formula (Bytes)

Experimental Results: SMT-CBMC vs. CBMC

BellmannFord(N)



Overall time (s)



Size of the formula (Bytes)

Experimental Results: SMT-CBMC vs. CBMC

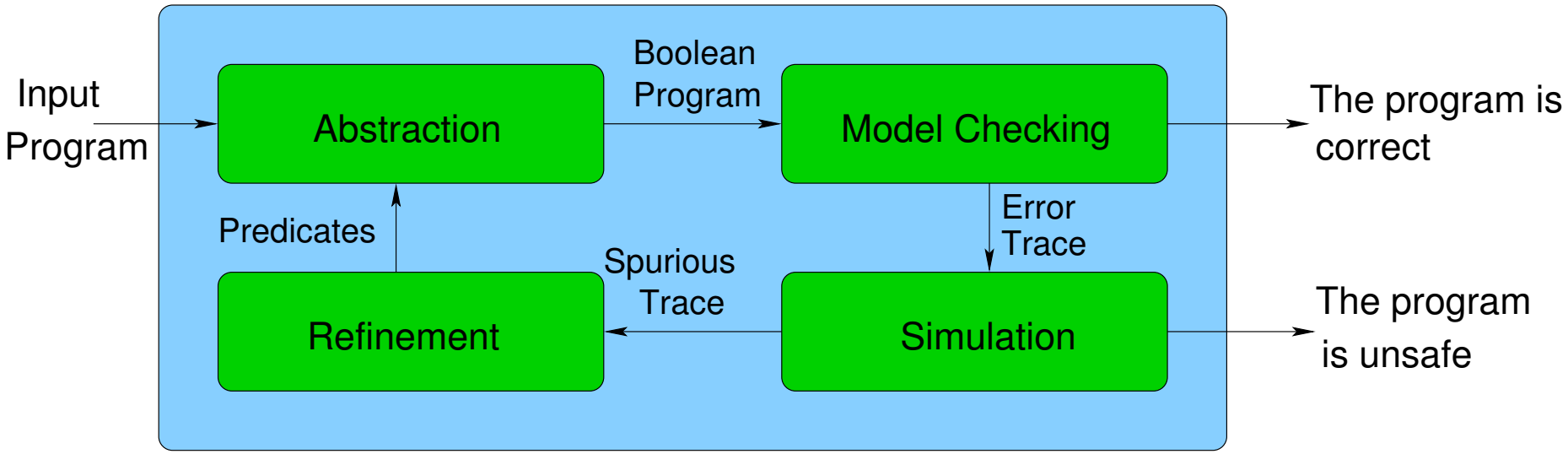
Prim's algorithm

	Size (Bytes)		Time (s)	
N	SMT-CBMC	CBMC	SMT-CBMC	CBMC
4	376,016	14,269,837	7.15	9.27
5	692,767	31,453,119	12.93	28.08
6	1,153,156	59,675,705	21.87	43.66
7	1,783,069	108,045,992	34.20	151.08
8	2,611,469	–	49.95	–
9	3,668,203	–	71.05	–
10	4,983,407	–	98.44	–
11	6,589,143	–	134.05	–
12	8,504,471	–	178.06	–

Outline

Outline

CounterExample-Guided Abstraction Refinement



Boolean Programs

A *Boolean Program* is a program in a fragment of the C programming language with

- expressions in conditional statements and assignments involve *boolean* combinations of *boolean-valued* variables and the symbol $\mathfrak{U}$ (standing for *undefined*).
- the usual control-flow constructs (**if**, **else**, **while**, **assert**),
- an additional **assume** statement,
- procedural abstraction with call-by-value parameter passing and recursion,

```
bool nU0;

void getUnit() {
 bool cE=F;
 if(nU0) {
  if(𝔘) {
   ;
   nU0=F;
   cE=T;
  }
 } else cE=T;
 if(cE) {
  if(nU0)
   ERROR: ;
   else
   ;
 }
}
```

Model Checking Boolean Programs

Model checking of Boolean programs can be done by

- combining program analysis and model-checking techniques
- using OBDDs for concisely representing the reachable states of the programs

Predicate Abstraction and Refinement

```
1  int numUnits;
2  int level;
3
4  void getUnit() {
5   int canEnter=F;
6   if(numUnits==0) {
7    if(level>0) {
8     NewUnit();
9     numUnits=1;
10    canEnter=1;
11   }
12  } else canEnter=1;
13  if(canEnter==1) {
14   if(numUnits==0)
15    ERROR: ;
16   else
17    gotUnit();
18  }
19 }
```

Question

Is line 15 reachable?

Approach

We start by abstracting away all data from the program.

Predicate Abstraction and Refinement

```
1  int numUnits;
2  int level;
3
4  void getUnit() {
5   int canEnter=F;
6   if(numUnits==0) {
7    if(level>0) {
8     NewUnit();
9     numUnits=1;
10    canEnter=1;
11   }
12  } else canEnter=1;
13  if(canEnter==1) {
14   if(numUnits==0)
15    ERROR: ;
16   else
17    gotUnit();
18  }
19 }
```

Question

Is line 15 reachable?

Approach

We start by abstracting away all data from the program.

Predicate Abstraction and Refinement

```
1
2
3
4  void getUnit() {
5   ;
6   if(𝔘) {
7    if(𝔘) {
8     ;
9     ;
10    ;
11   }
12  } else ;
13  if(𝔘) {
14   if(𝔘)
15    ERROR: ;
16   else
17    ;
18  }
19 }
```

Question

Is line 15 reachable?

Approach

We start by abstracting away all data from the program.

Predicate Abstraction and Refinement

```
1
2
3
4   void getUnit() {
5    ;
6    if(𝔘) {
7     if(𝔘) {
8      ;
9      ;
10     ;
11    }
12   } else ;
13   if(𝔘) {
14    if(𝔘)
15     ERROR: ;
16    else
17     ;
18   }
19  }
```

Question

Is line 15 reachable?

Answer

Yes! The shortest trace is

$$5, 6, 12, 13, 14, 15$$

Question

Is this trace feasible in the original program?

Predicate Abstraction and Refinement

```
1  int numUnits;
2  int level;
3
4  void getUnit() {
5   int canEnter=0;
6   if(numUnits==0) {
7    if(level>0) {
8     NewUnit();
9     numUnits=1;
10    canEnter=1;
11   }
12  } else canEnter=1;
13  if(canEnter==1) {
14   if(numUnits==0)
15    ERROR: ;
16   else
17    gotUnit();
18  }
19 }
```

Simulation

Trace 5, 6, 12, 13, 14, 15 is not feasible, because `numUnits==0` cannot be false at line 6 and true at line 14.

Adding predicates

We then introduce a new boolean variable (predicate) `nU0` that is true when `numUnits==0` and false otherwise.

Predicate Abstraction and Refinement

```
1  int numUnits;
2  int level;
3
4  void getUnit() {
5   bool canEnter=0;
6   if(numUnits==0) {
7    if(level>0) {
8     NewUnit();
9     numUnits=1;
10    canEnter=1;
11   }
12  } else canEnter=1;
13  if(canEnter==1) {
14   if(numUnits==0)
15    ERROR: ;
16   else
17    gotUnit();
18  }
19 }
```

```
1  bool nU0;
2
3
4  void getUnit() {
5   ;
6   if(nU0) {
7    if(𝔘) {
8     ;
9     nU0=F;
10    ;
11   }
12  } else ;
13  if(𝔘){
14   if(nU0)
15    ERROR: ;
16   else
17    ;
18  }
19 }
```

Predicate Abstraction and Refinement

Question

Is line 15 reachable in this new refined (but still abstract) program?

Answer

Yes! There is a trace 5,6,7,12,13,14,15

Question

Is this trace feasible in the original program?

```
bool nU0;                          1
                                   2
                                   3
void getUnit() {                   4
 ;                                 5
 if(nU0) {                         6
  if(𝔘) {                          7
   ;                               8
   nU0=F;                          9
   ;                               10
  }                                11
 } else ;                          12
 if(𝔘) {                          13
  if(nU0)                          14
   ERROR: ;                        15
  else                             16
   ;                               17
 }                                 18
}                                  19
```

Predicate Abstraction and Refinement

Simulation

Trace 5,6,7,13,14,15 is not feasible, because the assigment at line 5 sets `canEnter` to 0 and hence `canEnter==1` at line 13 cannot possibly be true.

Adding predicates

We then introduce a new boolean variable (predicate) `cE` that is true when `canEnter==1` and false otherwise.

```
int numUnits;                          1
int level;                             2
                                       3
void getUnit() {                       4
 bool canEnter=0;                      5
 if(numUnits==0) {                     6
  if(level>0) {                        7
   NewUnit();                          8
   numUnits=1;                         9
   canEnter=1;                         10
  }                                    11
 } else canEnter=1;                    12
 if(canEnter==1) {                     13
  if(numUnits==0)                      14
   ERROR: ;                            15
  else                                 16
   gotUnit();                          17
 }                                     18
}                                      19
```

Predicate Abstraction and Refinement

Question

Is line 15 reachable in this new refined program?

Answer

No! Hence the `ERROR` statement at line 15 is not reachable in the original, concrete program.

```
bool nU0;                          1
                                   2
                                   3
void getUnit() {                   4
 bool cE=F;                        5
 if(nU0) {                         6
  if(𝔘) {                          7
   ;                               8
   nU0=F;                          9
   cE=T;                           10
  }                                11
 } else cE=T;                      12
 if(cE) {                          13
  if(nU0)                          14
   ERROR: ;                        15
   else                            16
   ;                               17
 }                                 18
}                                  19
```

Issues

- Decision Procedures eavily used both in the abstraction and in the refinement steps.
- **Refinement** is the **most critical step** as it often fails to discover new predicates when:
  - there are complex correlations between variables,
  - reasoning about complex data structures (e.g. pointers, arrays) is required.
- Most tools (e.g. BLAST, SLAM, SATABS) encounter difficulties: they are often inconclusive or even give **wrong** results!

Outline

A new Model for Programs: Linear Programs

Linear Programs

Like Boolean Programs, Linear Programs have the usual control-flow constructs, **but**:

- Variables range over a numerical domain such as $\mathbb{Z}$ or $\mathbb{Q}$,
- Expressions in conditions and assignments are linear.

Model Checking Linear Programs

Model Checkers for Boolean programs can be lifted to work with Linear Programs by replacing OBBDs with packages capable to manipulate convex polyedra symbolically (e.g. Parma Polyedra Library).

- Pro: Many programs can be analysed directly
- Pro: Can successfully analyse programs involving complex correlation of data
- Cons: Termination not guaranteed (problem is undecidable)

Example of Linear Program

```
1  int numUnits;
2  int level;
3
4  void getUnit(){
5   int canEnter=0;
6   if(numUnits==0) {
7    if(level>0) {
8     NewUnit();
9     numUnits=1;
10    canEnter=1;
11   }
12  } else canEnter=1;
13  if(canEnter==1) {
14   if(numUnits==0)
15    assert(0);
16   else
17    gotUnit();
18  }
19 }
```

Linear Program

Same program as before does not need any abstraction: only one run of the model checker is needed!

- No abstraction
- No refinement

Example of Linear Program

```
1  int numUnits;
2  int level;
3
4  void getUnit(){
5   int canEnter=0;
6   if(numUnits==0) {
7    if(level>0) {
8     NewUnit();
9     numUnits=1;
10    canEnter=1;
11   }
12  } else canEnter=1;
13  if(canEnter==1) {
14   if(numUnits==0)
15    assert(0);
16   else
17    gotUnit();
18  }
19 }
```

Linear Program

Same program as before does not need any abstraction: only one run of the model checker is needed!

- No abstraction
- No refinement

Example of Linear Program

```
1  int numUnits;
2  int level;
3
4  void getUnit(){
5   int canEnter=0;
6   if(numUnits==0) {
7    if(level>0) {
8     NewUnit();
9     numUnits=1;
10    canEnter=1;
11   }
12  } else canEnter=1;
13  if(canEnter==1) {
14   if(numUnits==0)
15    assert(0);
16   else
17    gotUnit();
18  }
19 }
```

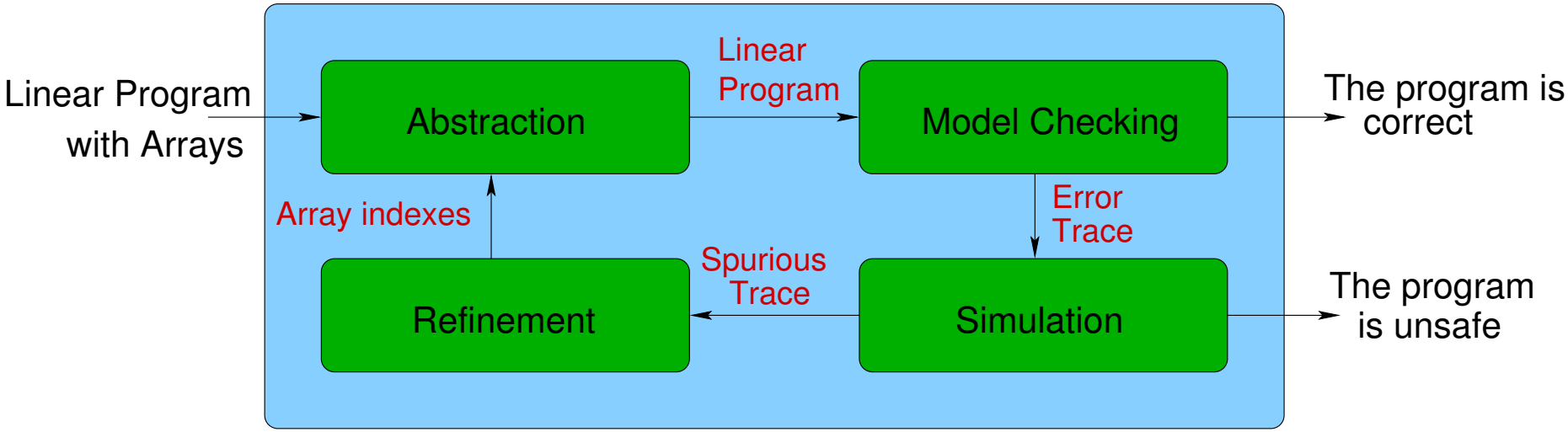
Linear Program

Same program as before does not need any abstraction: only one run of the model checker is needed!

- No abstraction
- No refinement

Array abstraction

Linear Programs pave the way to new kinds of abstraction techniques.



Linear Programs

A *Linear Program* is a program in a fragment of the C programming language with

- expressions in conditional statements and assignments involve *linear* combinations of numeric variables and the symbol $\mathfrak{U}$ (standing for *undefined*).
- the usual control-flow constructs (**if**, **else**, **while**, **assert**),
- an additional **assume** statement,
- procedural abstraction with call-by-value parameter passing and recursion,

```
void main() {
 int i, a¹;
 a¹=(1==1)?9:a¹;
 i=0;
 while(((i==1)?a¹:𝔘)!=9) {
  a¹=(i==1)?2*i:a¹;
  i=i+1;
 }
 if(i<=1)
  ERROR: ;
}
```

Linear Programs with Arrays

A *Linear Program with arrays* is a program in a fragment of the C programming language with

- expressions in conditional statements and assignments involve *linear* combinations of numeric variables, array elements, and the symbol $\mathfrak{U}$ (standing for *undefined*).
- the usual control-flow constructs (**if**, **else**, **while**, **assert**),
- an additional **assume** statement,
- procedural abstraction with call-by-value parameter passing and recursion,

```
void main() {
 int i, a[30];
 a[1]=9;
 i=0;
 while(a[i]!=9) {
  a[i]=2*i;
  i=i+1;
 }
 if(i<=1)
  ERROR: ;
}
```

Array Abstraction

An *array abstraction for a program P* is a function R that maps every array of P into a subset of $\mathrm{dom}(a)$, i.e. $R(a) \subseteq \mathrm{dom}(a)$ for all a in P.

An *array abstraction of a linear expression e w.r.t. R* is and expression $\widehat{e}$ obtained by replacing every expression of the form $a[e]$ with $\mathrm{abs}(a[e], [k_1, \ldots, k_n])$, where $[k_1, \ldots, k_n]$ is some permutation of $R(a)$ and $\mathrm{abs}(a[e], [k_1, \ldots, k_n])$ is defined by:

$$\mathrm{abs}(a[e], []) = \mathfrak{U}$$
$$\mathrm{abs}(a[e], [k_1, k_2, \ldots, k_n]) = (\widehat{e} == k_1 \; ? \; a^{k_1} \; : \; \mathrm{abs}(a[e], [k_2, \ldots, k_n]))$$

Example: If e is $\mathtt{a[i]} + 1$ and $R(a) = \{2, 5\}$, then $\widehat{e}$ is $((\mathtt{i} == 2) \; ? \; \mathtt{a}^2 : (\mathtt{i} == 5) \; ? \; \mathtt{a}^5 : \mathfrak{U}) + 1$.

Array Abstractions

An *array abstraction of P w.r.t. R* is obtained by replacing

- every expressions e occurring in P with $\widehat{e}$,
- every assignment of the form $a[i] = e;$ with the (parallel) assignment

$$a^{k_1}, \ldots, a^{k_n} = (\widehat{i} == k_1 \; ? \; \widehat{e} : a^{k_1}), \ldots, (\widehat{i} == k_n \; ? \; \widehat{e} : a^{k_n});$$

Example

Original Program

```
1  void main() {
2   int i, a[30];
3   a[1]=9;
4   i=0;
5   while(a[i]!=9) {
6    a[i]=2*i;
7    i=i+1;
8   }
9   if(i<=1)
10   ERROR: ;
11 }
```

Abstraction w.r.t. $R(a) = \{1\}$

```
void main() {                                  1
 int i, a^1;                                   2
 a^1=(1==1)?9:a^1;                             3
 i=0;                                          4
 while(((i==1)?a^1:𝔘)!=9) {                    5
  a^1=(i==1)?2*i:a^1;                          6
  i=i+1;                                       7
 }                                             8
 if(i<=1)                                      9
  ERROR: ;                                     10
}                                              11
```

CEGAR of Linear Programs with Arrays

procedure AR(P,R)

1. $\widehat{P} \leftarrow$ abstract(P, R);
2. *Trace* $\leftarrow$ model-check($\widehat{P}$);
3. **if** (*Trace* $=$ none) **then return** SAFE;
4. **if** ($R = R_P$) **then return** *Trace*;
5. *Formula* $\leftarrow$ encode(*Trace*, P);
6. *Result* $\leftarrow$ decide(*Formula*);
7. **if** *SAT?*(*Result*) **then return** *Trace*;

   /* *Result* contains a proof of the unsatisfiability of *Formula* */
8. $R' \leftarrow$ refine(*Trace*, *Result*, R);
9. **return** AR(P,R');

Example

Original Program

```
1  void main() {
2   int i, a[30];
3   a[1]=9;
4   i=0;
5   while(a[i]!=9) {
6    a[i]=2*i;
7    i=i+1;
8   }
9   if(i<=1)
10   ERROR: ;
11 }
```

Question

Is line 10 reachable?

Method

Abstract away every occurrence of array elements.

Example

Original Program

```
1  void main() {
2   int i, a[30];
3   a[1]=9;
4   i=0;
5   while(a[i]!=9) {
6    a[i]=2*i;
7    i=i+1;
8   }
9   if(i<=1)
10   ERROR: ;
11 }
```

Question

Is line 10 reachable?

Method

Abstract away every occurrence of array elements.

Example

Original Program

```
1  void main() {
2   int i, a[30];
3   a[1]=9;
4   i=0;
5   while(a[i]!=9) {
6    a[i]=2*i;
7    i=i+1;
8   }
9   if(i<=1)
10   ERROR: ;
11 }
```

Abstraction w.r.t. $R(a) = \emptyset$

```
void main() {
 int i;
 ;
 i = 0;
 while(𝔘!=9) {
  ;
  i = i+1;
 }
 if(i>1)
  ERROR: ;
}
```

Example

Abstraction w.r.t. $R(a) = \emptyset$

Question

Is line 10 reachable?

Answer

Yes! The trace is:

$3, 4, 5, 6, 7, 5, 6, 7, 5, 9, 10$

```
void main() {                    1
 int i;                          2
 ;                               3
 i = 0;                          4
 while(𝔘!=9) {                   5
  ;                              6
  i = i+1;                       7
 }                               8
 if(i>1)                         9
  ERROR: ;                       10
}                                11
```

Example

Question

Is line 10 reachable?

Answer

Yes! The trace is:

$3, 4, 5, 6, 7, 5, 6, 7, 5, 9, 10$

Abstraction w.r.t. $R(a) = \emptyset$

```
void main() {                    1
 int i;                          2
 ;                               3
 i = 0;                          4
 while(𝔘!=9) {                   5
  ;                              6
  i = i+1;                       7
 }                               8
 if(i>1)                         9
  ERROR: ;                       10
}                                11
```

Trace Simulation

Line	Original Statement	Renamed Statement	Trace Formula
3	`a[1] = 9;`	a_1`[1] = 9;`	$a_1 = \text{store}(a_0, 1, 9)$
4	`i = 0;`	i_1 `= 0;`	$i_1 = 0$
5	**while**`(a[i] != 9);`	**assume**`(`a_1`[`i_1`] != 9);`	$\text{select}(a_1, i_1) \neq 9$
6	`a[i] = 2 * i;`	a_2`[`i_1`] = 2 * `i_1`;`	$a_2 = \text{store}(a_1, i_1, 2 * i_1)$
7	`i = i + 1;`	i_2 `= `i_1` + 1;`	$i_2 = i_1 + 1$
5	**while**`(a[i] != 9);`	**assume**`(`a_2`[`i_2`] != 9);`	$\text{select}(a_2, i_2) \neq 9$
6	`a[i] = 2 * i;`	a_3`[`i_2`] = 2 * `i_2`;`	$a_3 = \text{store}(a_2, i_2, 2 * i_2)$
7	`i = i + 1;`	i_3 `= `i_2` + 1;`	$i_3 = i_2 + 1$
5	**while**`(a[i] != 9);`	**assume**`(!(`a_3`[`i_3`] != 9));`	$\neg(\text{select}(a_3, i_3) \neq 9)$
9	**if**`(i <= 1);`	**assume**`(`i_3` <= 1);`	$\neg(i_3 \leq 1)$

Theorem

Let P be a program, τ a sequence of statements of P, and $\Phi(\tau, P_i)$ the set of trace formulae associated with τ, then

$$\tau \in \text{traces}(P) \quad \textit{iff} \quad \Phi(\tau, P) \textit{ is } \mathcal{T}\textit{-satisfiable.}$$

Refinement

Goal: Find $R' \supseteq R$ such that $\Phi(\tau, \text{abstract}(P, R'))$ is $\mathcal{T}$-unsatisfiable.
Idea: Turn the proof of $\mathcal{T}$-unsatisfiability of $\Phi(\tau, P)$ into a proof of the $\mathcal{T}$-unsatisfiability of $\Phi(\tau, \text{abstract}(P, R'))$.

(R' must be determined in the process!)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{Q(i_2,a) \vdash \text{select}(a_2,i_2) \neq 9 \qquad \vdash i_2 = i_1 + 1}{Q(i_1+1,a) \vdash \text{select}(a_2, i_1+1) \neq 9} \qquad \vdash a_2 = \text{store}(a_1, i_1, 2 * i_1)}{Q(i_1+1,a) \vdash \text{select}(\text{store}(a_1,i_1,2*i_1), i_1+1) \neq 9} \; (1)}{Q(i_1+1,a) \vdash (i_1+1 = i_1 \; ? \; 2*i_1 : \text{select}(a_1, i_1+1)) \neq 9}}{Q(i_1+1,a) \vdash \text{select}(a_1, i_1+1) \neq 9 \qquad \vdash i_1 = 0}}{Q(1,a) \vdash \text{select}(a_1,1) \neq 9 \qquad \vdash a_1 = \text{store}(a_0,1,9)}}{Q(1,a) \vdash \text{select}(\text{store}(a_0,1,9),1) \neq 9}}{Q(1,a) \vdash (1 = 1 \; ? \; 9 : \text{select}(a_0,1)) \neq 9}}{Q(1,a) \vdash \bot}$$

where

$$\forall a, i, j, e.\ \text{select}(\text{store}(a,i,e),j) = (j = i \; ? \; e : \text{select}(a,j)) \qquad (1)$$

is the axiom of the theory of arrays.

Refinement

Goal: Find $R' \supseteq R$ such that $\Phi(\tau, \text{abstract}(P, R'))$ is $\mathcal{T}$-unsatisfiable.
Idea: Turn the proof of $\mathcal{T}$-unsatisfiability of $\Phi(\tau, P)$ into a proof of the $\mathcal{T}$-unsatisfiability of $\Phi(\tau, \text{abstract}(P, R'))$.

(R' must be determined in the process!)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{Q(i_2,a) \vdash \text{select}(a_2,i_2) \neq 9 \qquad \vdash i_2 = i_1 + 1}{Q(i_1+1,a) \vdash \text{select}(a_2, i_1+1) \neq 9} \qquad \vdash a_2 = \text{store}(a_1, i_1, 2*i_1)}{Q(i_1+1,a) \vdash \text{select}(\text{store}(a_1,i_1,2*i_1), i_1+1) \neq 9}\,(1)}{Q(i_1+1,a) \vdash (i_1+1 = i_1 \;?\; 2*i_1 : \text{select}(a_1, i_1+1)) \neq 9}}{Q(i_1+1,a) \vdash \text{select}(a_1, i_1+1) \neq 9 \qquad \vdash i_1 = 0}}{Q(1,a) \vdash \text{select}(a_1,1) \neq 9 \qquad \vdash a_1 = \text{store}(a_0,1,9)}}{Q(1,a) \vdash \text{select}(\text{store}(a_0,1,9),1) \neq 9}}{Q(1,a) \vdash (1 = 1 \;?\; 9 : \text{select}(a_0,1)) \neq 9}}{Q(1,a) \vdash \bot}$$

How: **Step 1.** Add to the antecedent of each leaf sequent $\vdash \varphi$ a formula $Q(e,a)$ for each term $\text{select}(a_k, e)$ occurring in φ.
Informally, $Q(e,a)$ is a placeholder for $\bigvee_{k \in R'(a)} e = k$, i.e. $e \in R'(a)$.

Refinement

Goal: Find $R' \supseteq R$ such that $\Phi(\tau, \text{abstract}(P, R'))$ is $\mathcal{T}$-unsatisfiable.
Idea: Turn the proof of $\mathcal{T}$-unsatisfiability of $\Phi(\tau, P)$ into a proof of the $\mathcal{T}$-unsatisfiability of $\Phi(\tau, \text{abstract}(P, R'))$.

(R' must be determined in the process!)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{Q(i_2,a) \vdash \text{select}(a_2,i_2) \neq 9 \qquad \vdash i_2 = i_1 + 1}{Q(i_1+1,a) \vdash \text{select}(a_2, i_1+1) \neq 9} \qquad \vdash a_2 = \text{store}(a_1, i_1, 2 * i_1)}{Q(i_1+1,a) \vdash \text{select}(\text{store}(a_1,i_1,2*i_1), i_1+1) \neq 9} \; (1)}{Q(i_1+1,a) \vdash (i_1 + 1 = i_1 \; ? \; 2 * i_1 : \text{select}(a_1, i_1+1)) \neq 9}}{Q(i_1+1,a) \vdash \text{select}(a_1, i_1+1) \neq 9 \qquad \vdash i_1 = 0}}{Q(1,a) \vdash \text{select}(a_1,1) \neq 9 \qquad \vdash a_1 = \text{store}(a_0,1,9)}}{Q(1,a) \vdash \text{select}(\text{store}(a_0,1,9),1) \neq 9}}{Q(1,a) \vdash (1 = 1 \; ? \; 9 : \text{select}(a_0,1)) \neq 9}}{Q(1,a) \vdash \bot}$$

How: **Step 2.** "Replay" the proof.

Refinement

Goal: Find $R' \supseteq R$ such that $\Phi(\tau, \text{abstract}(P, R'))$ is $\mathcal{T}$-unsatisfiable.
Idea: Turn the proof of $\mathcal{T}$-unsatisfiability of $\Phi(\tau, P)$ into a proof of the $\mathcal{T}$-unsatisfiability of $\Phi(\tau, \text{abstract}(P, R'))$.

(R' must be determined in the process!)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{Q(i_2,a) \vdash \text{select}(a_2,i_2) \neq 9 \qquad \vdash i_2 = i_1 + 1}{Q(i_1+1,a) \vdash \text{select}(a_2,i_1+1) \neq 9} \qquad \vdash a_2 = \text{store}(a_1,i_1,2*i_1)}{Q(i_1+1,a) \vdash \text{select}(\text{store}(a_1,i_1,2*i_1),i_1+1) \neq 9}\,(1)}{Q(i_1+1,a) \vdash (i_1+1 = i_1\ ?\ 2*i_1 : \text{select}(a_1,i_1+1)) \neq 9}}{Q(i_1+1,a) \vdash \text{select}(a_1,i_1+1) \neq 9 \qquad \vdash i_1 = 0}}{Q(1,a) \vdash \text{select}(a_1,1) \neq 9 \qquad \vdash a_1 = \text{store}(a_0,1,9)}}{Q(1,a) \vdash \text{select}(\text{store}(a_0,1,9),1) \neq 9}}{Q(1,a) \vdash (1 = 1\ ?\ 9 : \text{select}(a_0,1)) \neq 9}}{Q(1,a) \vdash \bot}$$

How: **Step 2.** "Replay" the proof.

Refinement

Goal: Find $R' \supseteq R$ such that $\Phi(\tau, \text{abstract}(P, R'))$ is $\mathcal{T}$-unsatisfiable.
Idea: Turn the proof of $\mathcal{T}$-unsatisfiability of $\Phi(\tau, P)$ into a proof of the $\mathcal{T}$-unsatisfiability of $\Phi(\tau, \text{abstract}(P, R'))$.

(R' must be determined in the process!)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{Q(i_2,a) \vdash \text{select}(a_2,i_2) \neq 9 \qquad \vdash i_2 = i_1 + 1}{Q(i_1+1,a) \vdash \text{select}(a_2, i_1+1) \neq 9} \qquad \vdash a_2 = \text{store}(a_1, i_1, 2 * i_1)}{Q(i_1+1,a) \vdash \text{select}(\text{store}(a_1,i_1,2*i_1), i_1+1) \neq 9} \; (1)}{Q(i_1+1,a) \vdash (i_1+1 = i_1 \; ? \; 2*i_1 : \text{select}(a_1, i_1+1)) \neq 9}}{Q(i_1+1,a) \vdash \text{select}(a_1, i_1+1) \neq 9 \qquad \vdash i_1 = 0}}{Q(1,a) \vdash \text{select}(a_1, 1) \neq 9 \qquad \vdash a_1 = \text{store}(a_0, 1, 9)}}{Q(1,a) \vdash \text{select}(\text{store}(a_0,1,9), 1) \neq 9}}{Q(1,a) \vdash (1 = 1 \; ? \; 9 : \text{select}(a_0, 1)) \neq 9}}{Q(1,a) \vdash \bot}$$

How: **Step 2.** "Replay" the proof.

Refinement

Goal: Find $R' \supseteq R$ such that $\Phi(\tau, \text{abstract}(P, R'))$ is $\mathcal{T}$-unsatisfiable.
Idea: Turn the proof of $\mathcal{T}$-unsatisfiability of $\Phi(\tau, P)$ into a proof of the $\mathcal{T}$-unsatisfiability of $\Phi(\tau, \text{abstract}(P, R'))$.

(R' must be determined in the process!)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{Q(i_2,a) \vdash \text{select}(a_2,i_2) \neq 9 \qquad \vdash i_2 = i_1 + 1}{Q(i_1+1,a) \vdash \text{select}(a_2, i_1+1) \neq 9} \qquad \vdash a_2 = \text{store}(a_1, i_1, 2 * i_1)}{Q(i_1+1,a) \vdash \text{select}(\text{store}(a_1,i_1,2*i_1), i_1+1) \neq 9} \; (1)}{Q(i_1+1,a) \vdash (i_1+1 = i_1 \; ? \; 2*i_1 : \text{select}(a_1, i_1+1)) \neq 9}}{Q(i_1+1,a) \vdash \text{select}(a_1, i_1+1) \neq 9 \qquad \vdash i_1 = 0}}{Q(1,a) \vdash \text{select}(a_1,1) \neq 9 \qquad \vdash a_1 = \text{store}(a_0,1,9)}}{Q(1,a) \vdash \text{select}(\text{store}(a_0,1,9),1) \neq 9}}{Q(1,a) \vdash (1 = 1 \; ? \; 9 : \text{select}(a_0,1)) \neq 9}}{Q(1,a) \vdash \bot}$$

How: **Step 2.** "Replay" the proof.

Refinement

Goal: Find $R' \supseteq R$ such that $\Phi(\tau, \text{abstract}(P, R'))$ is $\mathcal{T}$-unsatisfiable.
Idea: Turn the proof of $\mathcal{T}$-unsatisfiability of $\Phi(\tau, P)$ into a proof of the $\mathcal{T}$-unsatisfiability of $\Phi(\tau, \text{abstract}(P, R'))$.

(R' must be determined in the process!)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{Q(i_2,a) \vdash \text{select}(a_2,i_2) \neq 9 \qquad \vdash i_2 = i_1 + 1}{Q(i_1+1,a) \vdash \text{select}(a_2,i_1+1) \neq 9} \qquad \vdash a_2 = \text{store}(a_1,i_1,2*i_1)}{Q(i_1+1,a) \vdash \text{select}(\text{store}(a_1,i_1,2*i_1),i_1+1) \neq 9}\,(1)}{Q(i_1+1,a) \vdash (i_1+1 = i_1\ ?\ 2*i_1 : \text{select}(a_1,i_1+1)) \neq 9}}{Q(i_1+1,a) \vdash \text{select}(a_1,i_1+1) \neq 9 \qquad \vdash i_1 = 0}}{Q(1,a) \vdash \text{select}(a_1,1) \neq 9 \qquad \vdash a_1 = \text{store}(a_0,1,9)}}{Q(1,a) \vdash \text{select}(\text{store}(a_0,1,9),1) \neq 9}}{Q(1,a) \vdash (1 = 1\ ?\ 9 : \text{select}(a_0,1)) \neq 9}}{Q(1,a) \vdash \bot}$$

How: **Step 2.** "Replay" the proof.

Refinement

Goal: Find $R' \supseteq R$ such that $\Phi(\tau, \text{abstract}(P, R'))$ is $\mathcal{T}$-unsatisfiable.
Idea: Turn the proof of $\mathcal{T}$-unsatisfiability of $\Phi(\tau, P)$ into a proof of the $\mathcal{T}$-unsatisfiability of $\Phi(\tau, \text{abstract}(P, R'))$.

(R' must be determined in the process!)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{Q(i_2,a) \vdash \text{select}(a_2,i_2) \neq 9 \qquad \vdash i_2 = i_1 + 1}{Q(i_1+1,a) \vdash \text{select}(a_2,i_1+1) \neq 9} \qquad \vdash a_2 = \text{store}(a_1,i_1,2*i_1)}{Q(i_1+1,a) \vdash \text{select}(\text{store}(a_1,i_1,2*i_1),i_1+1) \neq 9}\,(1)}{Q(i_1+1,a) \vdash (i_1+1 = i_1\ ?\ 2*i_1 : \text{select}(a_1,i_1+1)) \neq 9}}{Q(i_1+1,a) \vdash \text{select}(a_1,i_1+1) \neq 9 \qquad \vdash i_1 = 0}}{Q(1,a) \vdash \text{select}(a_1,1) \neq 9 \qquad \vdash a_1 = \text{store}(a_0,1,9)}}{Q(1,a) \vdash \text{select}(\text{store}(a_0,1,9),1) \neq 9}}{Q(1,a) \vdash (1 = 1\ ?\ 9 : \text{select}(a_0,1)) \neq 9}}{Q(1,a) \vdash \bot}$$

How: **Step 2.** "Replay" the proof.

Refinement

Goal: Find $R' \supseteq R$ such that $\Phi(\tau, \text{abstract}(P, R'))$ is $\mathcal{T}$-unsatisfiable.
Idea: Turn the proof of $\mathcal{T}$-unsatisfiability of $\Phi(\tau, P)$ into a proof of the $\mathcal{T}$-unsatisfiability of $\Phi(\tau, \text{abstract}(P, R'))$.

(R' must be determined in the process!)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{Q(i_2,a) \vdash \text{select}(a_2,i_2) \neq 9 \qquad \vdash i_2 = i_1 + 1}{Q(i_1+1,a) \vdash \text{select}(a_2,i_1+1) \neq 9} \qquad \vdash a_2 = \text{store}(a_1,i_1,2*i_1)}{Q(i_1+1,a) \vdash \text{select}(\text{store}(a_1,i_1,2*i_1),i_1+1) \neq 9}\,(1)}{Q(i_1+1,a) \vdash (i_1+1 = i_1\ ?\ 2*i_1 : \text{select}(a_1,i_1+1)) \neq 9}}{Q(i_1+1,a) \vdash \text{select}(a_1,i_1+1) \neq 9 \qquad \vdash i_1 = 0}}{Q(1,a) \vdash \text{select}(a_1,1) \neq 9 \qquad \vdash a_1 = \text{store}(a_0,1,9)}}{Q(1,a) \vdash \text{select}(\text{store}(a_0,1,9),1) \neq 9}}{Q(1,a) \vdash (1 = 1\ ?\ 9 : \text{select}(a_0,1)) \neq 9}}{Q(1,a) \vdash \bot}$$

How: **Step 2.** "Replay" the proof.

Refinement

Goal: Find $R' \supseteq R$ such that $\Phi(\tau, \text{abstract}(P, R'))$ is $\mathcal{T}$-unsatisfiable.
Idea: Turn the proof of $\mathcal{T}$-unsatisfiability of $\Phi(\tau, P)$ into a proof of the $\mathcal{T}$-unsatisfiability of $\Phi(\tau, \text{abstract}(P, R'))$.

(R' must be determined in the process!)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{Q(i_2,a) \vdash \text{select}(a_2,i_2) \neq 9 \qquad \vdash i_2 = i_1 + 1}{Q(i_1+1,a) \vdash \text{select}(a_2,i_1+1) \neq 9} \qquad \vdash a_2 = \text{store}(a_1,i_1,2*i_1)}{Q(i_1+1,a) \vdash \text{select}(\text{store}(a_1,i_1,2*i_1),i_1+1) \neq 9}\,(1)}{Q(i_1+1,a) \vdash (i_1+1 = i_1\ ?\ 2*i_1 : \text{select}(a_1,i_1+1)) \neq 9}}{Q(i_1+1,a) \vdash \text{select}(a_1,i_1+1) \neq 9} \qquad \vdash i_1 = 0}{Q(1,a) \vdash \text{select}(a_1,1) \neq 9} \qquad \vdash a_1 = \text{store}(a_0,1,9)}{Q(1,a) \vdash \text{select}(\text{store}(a_0,1,9),1) \neq 9}}{Q(1,a) \vdash (1 = 1\ ?\ 9 : \text{select}(a_0,1)) \neq 9}}{Q(1,a) \vdash \bot}$$

How: **Step 2.** "Replay" the proof.

Refinement

Goal: Find $R' \supseteq R$ such that $\Phi(\tau, \text{abstract}(P, R'))$ is $\mathcal{T}$-unsatisfiable.
Idea: Turn the proof of $\mathcal{T}$-unsatisfiability of $\Phi(\tau, P)$ into a proof of the $\mathcal{T}$-unsatisfiability of $\Phi(\tau, \text{abstract}(P, R'))$.

(R' must be determined in the process!)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{Q(i_2,a) \vdash \text{select}(a_2,i_2) \neq 9 \qquad \vdash i_2 = i_1 + 1}{Q(i_1+1,a) \vdash \text{select}(a_2, i_1+1) \neq 9} \qquad \vdash a_2 = \text{store}(a_1, i_1, 2 * i_1)}{Q(i_1+1,a) \vdash \text{select}(\text{store}(a_1,i_1,2*i_1), i_1+1) \neq 9} \; (1)}{Q(i_1+1,a) \vdash (i_1+1 = i_1 \; ? \; 2*i_1 : \text{select}(a_1, i_1+1)) \neq 9}}{Q(i_1+1,a) \vdash \text{select}(a_1, i_1+1) \neq 9 \qquad \vdash i_1 = 0}}{Q(1,a) \vdash \text{select}(a_1, 1) \neq 9 \qquad \vdash a_1 = \text{store}(a_0, 1, 9)}}{Q(1,a) \vdash \text{select}(\text{store}(a_0,1,9), 1) \neq 9}}{Q(1,a) \vdash (1 = 1 \; ? \; 9 : \text{select}(a_0, 1)) \neq 9}}{Q(1,a) \vdash \bot}$$

How: **Step 3.** Set $R'(a) = R(a) \cup \{k_1, \ldots, k_n\}$, where $Q(a, k_1), \ldots, Q(a, k_n) \vdash \bot$ is the root sequent.

Example (continued)

Original Program

```
void main() {                    1
 int i, a[30];                   2
 a[1]=9;                         3
 i=0;                            4
 while(a[i]!=9) {                5
  a[i]=2*i;                      6
  i=i+1;                         7
 }                               8
 if(i<=1)                        9
  ERROR: ;                       10
}                                11
```

Refinement

We add the index 1 to $R(a)$.

Example (continued)

Original Program

```
1  void main() {
2   int i, a[30];
3   a[1]=9;
4   i=0;
5   while(a[i]!=9) {
6    a[i]=2*i;
7    i=i+1;
8   }
9   if(i<=1)
10   ERROR: ;
11 }
```

Abstraction w.r.t. $R(a) = \{1\}$

```
void main() {                                  1
 int i, a^1;                                   2
 a^1=(1==1)?9:a^1;                             3
 i=0;                                          4
 while(((i==1)?a^1:𝔘)!=9) {                    5
  a^1=(i==1)?2*i:a^1;                          6
  i=i+1;                                       7
 }                                             8
 if(i<=1)                                      9
  ERROR: ;                                     10
}                                              11
```

Example (continued)

Abstraction w.r.t. $R(a) = \{1\}$

Question

Is line 10 reachable in the refined program?

Final result

No! The refined abstract program is safe, and so is the original one.

```
void main() {                                  1
 int i, a^1;                                   2
 a^1=(1==1)?9:a^1;                             3
 i=0;                                          4
 while(((i==1)?a^1:𝔘)!=9) {                    5
  a^1=(i==1)?2*i:a^1;                          6
  i=i+1;                                       7
 }                                             8
 if(i<=1)                                      9
  ERROR: ;                                     10
}                                              11
```

Example (continued)

Abstraction w.r.t. $R(a) = \{1\}$

Question

Is line 10 reachable in the refined program?

Final result

No! The refined abstract program is safe, and so is the original one.

```
void main() {                                  1
 int i, a^1;                                   2
 a^1=(1==1)?9:a^1;                             3
 i=0;                                          4
 while(((i==1)?a^1:𝔘)!=9) {                    5
  a^1=(i==1)?2*i:a^1;                          6
  i=i+1;                                       7
 }                                             8
 if(i<=1)                                      9
  ERROR: ;                                     10
}                                              11
```

The Eureka Software Model Checker

- Implements the above ideas
- Model checker for Linear Programs: based on Parma Polyedra Library
- Abstraction-Refinement Loop: based on custom version of CVC-Lite

Experimental results - Safe Instances

Benchmark	EUREKA			BLAST		SATABS		
	N	total time	refined/total array elems	N	total time	N	ref. time	total time
STRING COPY	1000*	127.89	$1/2N$	Incorrect		10	105.98	144.69
STRING COMPARE	1000*	28.62	$2/2N$	Incorrect		9	210.619	296.20
GRAY CODE	60	62.75	$16/28$	Incorrect			Inconclusive	
PARTITION	40	108.23	$1/N$	Incorrect			Inconclusive	
BUBBLE SORT	9	77.67	N/N	Incorrect		2	24.39	30.42
INSERTION SORT	16	60.86	N/N	Incorrect		2	51.43	74.74
SELECTION SORT	9	64.20	N/N	Incorrect		2	75.53	115.86
TCAS	–	129.64	$1/4$	Incorrect		–	11.29	32.42
FIBONACCI	1000*	6.91	$0/0$	24	5.68	1000*	1.24	2.21
BRESENHAM	1000*	16.84	$0/0$	Error		1000*	71.15	83.16
SWAP	1000*	2.40	$0/0$	24	5.33	64	8.25	109.44

Experimental results - Unsafe Instances

Benchmark	EUREKA			BLAST		SATABS		
	N	total time	refined/total array elems	N	total time	N	ref. time	total time
STRING COPY	1000*	73.70	$0/2N$	100*	167.98	21	724.40	819.77
STRING COMPARE	1000*	15.07	$0/2N$	100*	435.26	12	292.19	348.19
GRAY CODE	1000*	1.70	12/28	1000*	7.62		Inconclusive	
PARTITION	1000*	61.42	$0/N$	10*	214.31	21	127.91	186.63
BUBBLE SORT	50*	26.19	$0/N$	15*	395.38	7	327.33	460.76
INSERTION SORT	100*	190.74	$0/N$	20*	316.00	5	131.58	270.64
SELECTION SORT	500*	41.15	$0/N$	20*	506.81	7	101.97	291.20
TCAS	–	2.96	1/4	–	3.58	–	1.16	4.86
FIBONACCI	1000*	7.01	0/0	25	5.29	20*	444.31	533.99
BRESENHAM	1000*	0.74	0/0		Error	2	263.09	287.70
SWAP	1000*	5.73	0/0	1000*	16.79	1000*	4.74	136.43

Outline

Summing up

- *Bounded Model Checking of Software*
  - Encoding into a SMT instead of SAT leads to considerable savings
  - This allows us to analyse programs more deeply (w.r.t. the bound) than SAT-based tools whose encoding depend on the size of data structures like arrays.
- *CEGAR with Predicate Abstraction*
  - Good on programs that manipulate simple data structures
  - Improvements are being put forward
- *CEGAR with Array Abstraction*
  - Linear Programs are a new, useful target of abstraction that natively supports analysis of complex correlations between variables.
  - Tight interplay between model checker and decision procedure via proof transformation.
  - Superior to state-of-the-art SW model checkers on many programs of interest.

Challanges and Opportunities

- *Bounded Model Checking of Software*
  - New challange testbed for SMT-solvers (SMT-LIB, SMT-COMP)
  - First results are encouraging but competition with SAT-based approaches is fierce.
- *CEGAR with Predicate Abstraction*
  - Extracting interpolants from proofs is very useful to get better refinements, cf. work by McMillan on Interpolating Theorem Prover [McM05]
- *CEGAR with Array Abstraction*
  - Devise new abstraction/refinement mechanisms to support model checking of programs that manipulate other data structures (e.g. dynamic data structures)

Acknowledgements

The work presened would not have been possible without the countribution of the following people:

- *Jacopo Mantovani*: Array Abstraction, design and development of Eureka, Bounded Model Checking of SW
- *Massimo Benerecetti*: Array Abstraction, design of Eureka
- *Lorenzo Platania*: Bounded Model Checking of SW, design and development of SMT-CBMC
- *Dario Carotenuto* and *Pasquale Spica*: development of Eureka

References


Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania.
Bounded model checking of software using SMT solvers instead of SAT solvers.
In Antti Valmari, editor, *SPIN*, volume 3925 of *Lecture Notes in Computer Science*, pages 146–162. Springer, 2006.


Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu.
Symbolic model checking without BDDs.
In Rance Cleaveland, editor, *Proceedings of TACAS (Tools and Algorithms for Construction and Analysis of Systems)*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.


Daniel Kroening, Edmund Clarke, and Karen Yorav.
Behavioral consistency of C and Verilog programs using bounded model checking.
In *Proceedings of DAC03*, pages 368–371. ACM Press, 2003.


McMillan.
An interpolating theorem prover.
TCS: Theoretical Computer Science, 345, 2005.


Sriram Rajamani.
SLAM: Software model checking from theory to practice.
Invited Talk at APSSEM II Workshop, 2004.