

Software Model Checking at AI-Lab

Jacopo Mantovani

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

SAP – Sophia Antipolis – November 9, 2005



1 Introduction

2 Symbolic Software Model Checking

- A few words about Model Checking
- The state of the art
- The EUREKA approach at AI-Lab

3 Bounded Model Checking of Software

- A few words about Bounded Model Checking
- The state of the art
- The EUREKA approach at AI-Lab

4 Conclusion

1 Introduction

2 Symbolic Software Model Checking

- A few words about Model Checking
- The state of the art
- The EUREKA approach at AI-Lab

3 Bounded Model Checking of Software

- A few words about Bounded Model Checking
- The state of the art
- The EUREKA approach at AI-Lab

4 Conclusion

- Model Checking has confirmed itself as an important application of logic in the field of **hardware** verification.
- Several industries adopted Model Checking as a complementary technique to testing (Intel, Motorola, etc.)
- In the last years growing effort has been spent in the application of Model Checking to **software** artifacts.

The State of the Art in Software Model Checking

Success stories:

- **Symbolic Model Checking using Abstraction and Refinement:**
 - The SLAM Toolkit (Ball et al. at Microsoft Research),
 - BLAST (Henzinger, Majumdar, et al. at Berkeley University),
 - ComFort/MAGIC (Clarke et al. at CMU)
 - ...
- **Bounded Model Checking:**
 - CBMC (Clarke, Kroening, et al. at CMU).

Industrial applications:

- Windows device drivers (SLAM, BLAST, ComFort),
- encryption algorithms like DES (CBMC),
- flights collision avoidance software like TCAS (CBMC).

1 Introduction

2 Symbolic Software Model Checking

- A few words about Model Checking
- The state of the art
- The EUREKA approach at AI-Lab

3 Bounded Model Checking of Software

- A few words about Bounded Model Checking
- The state of the art
- The EUREKA approach at AI-Lab

4 Conclusion

Model Checking - Kripke Structures

In Model Checking, a system is formalised as a Kripke Structure.

Definition (Kripke structure)

Let AP be a set of atomic propositions. A *Kripke Structure* M over AP is a 4-tuple

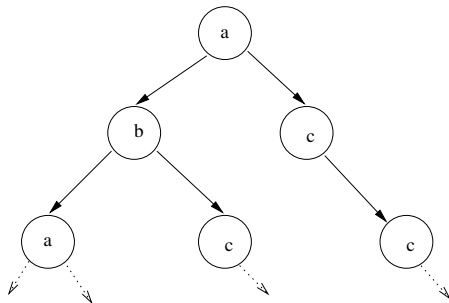
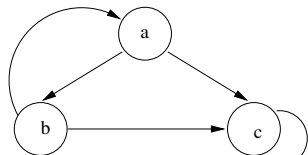
$$M = \langle \Sigma, \mathcal{I}, \rightsquigarrow, \mathcal{L} \rangle$$

where

- Σ is a **finite** set of states,
- $\mathcal{I} \subseteq \Sigma$ is the set of initial states,
- $\rightsquigarrow \subseteq \Sigma \times \Sigma$ is the transition relation,
- $\mathcal{L} : \Sigma \rightarrow \Sigma^{AP}$ is a function that labels each state with the set $AP' \subseteq AP$ of atomic propositions that are true in that state.

Model Checking - Kripke Structures

Roughly, a Kripke structure is nothing but a transition graph.
Its unwinding leads to a computation tree.



Model Checking - Specification of the properties

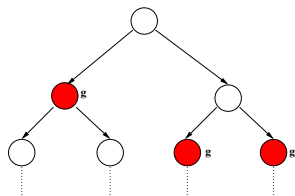
Desirable properties of a system are usually specified in some temporal logics like *LTL* and *CTL*.

Some examples:

Unavoidableness: **AF***g*

Reachability: **EF***g*

Invariance: **AG***g*



Model Checking - Specification of the properties

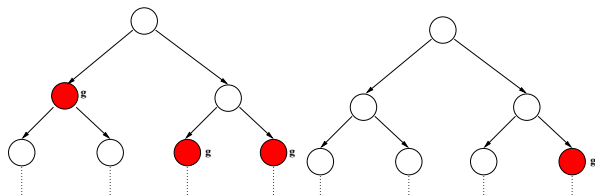
Desirable properties of a system are usually specified in some temporal logics like *LTL* and *CTL*.

Some examples:

Unavoidableness: **AF***g*

Reachability: **EF***g*

Invariance: **AG***g*



Model Checking - Specification of the properties

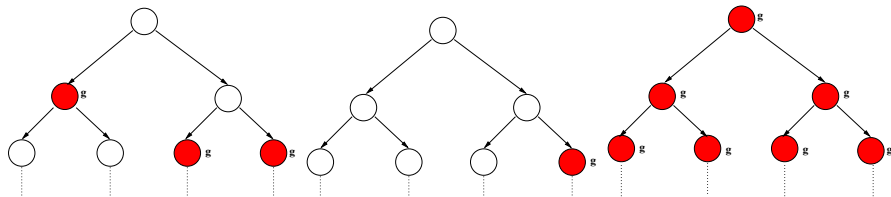
Desirable properties of a system are usually specified in some temporal logics like *LTL* and *CTL*.

Some examples:

Unavoidableness: **AF***g*

Reachability: **EF***g*

Invariance: **AG***g*



Given a property ϕ , the goal of Model Checking is to verify whether ϕ holds in M .

$$\models_M \phi$$

- The problem is decidable
- For *LTL* it is P-SPACE complete;
- For *CTL* it is decidable in $\mathcal{O}(|M| \cdot |\phi|)$.

Issues in Software Model Checking

Pros:

- Model Checking is fully automated;
- No proof inspection needed (unlike theorem proving);
- Counterexamples are output in case an error is found.

But:

- Model Checking only applies to **finite** systems;
- Software is inherently infinite-state;
- Model Checking alone is not applicable anymore;
- Some **abstraction** is needed in order to reduce infiniteness of software to finiteness.

Given a program P and a property ϕ to check,

- 1 Build an abstract model $P^\#$ of P
- 2 Model check $P^\#$
- 3 Two possible outcomes:
 - If $\models_{P^\#} \phi$, then $\models_P \phi$, and return **TRUE**.
 - If $\not\models_{P^\#} \phi$ with a counterexample $C^\#$, then check $C^\#$:

What are abstract models like?

Given a program P and a property ϕ to check,

- 1 Build an abstract model $P^\#$ of P
- 2 Model check $P^\#$
- 3 Two possible outcomes:
 - If $\models_{P^\#} \phi$, then $\models_P \phi$, and return **TRUE**.
 - If $\not\models_{P^\#} \phi$ with a counterexample $C^\#$, then check $C^\#$:
 - If $C^\#$ is also a concrete counterexample, then $\not\models_P \phi$ and return **FALSE**.
 - If $C^\#$ is spurious, then refine $P^\#$ and go back to Step 2.

What are abstract models like?

Given a program P and a property ϕ to check,

- 1 Build an abstract model $P^\#$ of P
- 2 Model check $P^\#$
- 3 Two possible outcomes:
 - If $\models_{P^\#} \phi$, then $\models_P \phi$, and return **TRUE**.
 - If $\not\models_{P^\#} \phi$ with a counterexample $C^\#$, then check $C^\#$:
 - If $C^\#$ is also a concrete counterexample, then $\not\models_P \phi$ and return **FALSE**.
 - If $C^\#$ is spurious, then refine $P^\#$ and go back to Step 2.

What are abstract models like?

Given a program P and a property ϕ to check,

- 1 Build an abstract model $P^\#$ of P
- 2 Model check $P^\#$
- 3 Two possible outcomes:
 - If $\models_{P^\#} \phi$, then $\models_P \phi$, and return **TRUE**.
 - If $\not\models_{P^\#} \phi$ with a counterexample $C^\#$, then check $C^\#$:
 - If $C^\#$ is also a concrete counterexample, then $\not\models_P \phi$ and return **FALSE**.
 - If $C^\#$ is spurious, then refine $P^\#$ and go back to Step 2.

What are abstract models like?

Example of Boolean Abstraction and Refinement

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

Question

Assertion at label [14] can be reached?

Approach

We first completely abstract all data flow.

Example of Boolean Abstraction and Refinement

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

Question

Assertion at label [14] can be reached?

Approach

We first completely abstract all data flow.

Example of Boolean Abstraction and Refinement

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

Question

Assertion at label [14] can be reached?

Answer

Yes! The shortest trace is

4, 5, 12, 13, 14

Question

Is that trace feasible in the concrete program?

Example of Boolean Abstraction and Refinement

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

Simulation

Trace 4, 5, 12, 13, 14 is not feasible, because it generates an inconsistency:
 $\text{numUnits} \neq 0$ at line 5 and
 $\text{numUnits} = 0$ at line 13!

Adding predicates

We then introduce a new boolean variable $nU0$ that is true when $\text{numUnits} = 0$ and false otherwise.

Example of Boolean Abstraction and Refinement

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

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

Example of Boolean Abstraction and Refinement

Question

Is now statement at label 14 reachable in this new abstract program?

Answer

Yes! There is a trace
4,5,6,12,13,14

Question

Is that trace feasible in the concrete program?

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

Example of Boolean Abstraction and Refinement

Simulation

Again, trace 4, 5, 6, 12, 13, 14 is not feasible, because it generates an inconsistency: $\text{canEnter} = F$ at line 4 and $\text{canEnter} = T$ at line 12!

Adding predicates

We then introduce a new boolean variable cE that is true when $\text{canEnter} = T$ and false otherwise.

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

Example of Boolean Abstraction and Refinement

Reachable?

At this step label 14 is not reachable anymore in the abstract program, and so is in the concrete one.

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

- The detection of a spurious execution trace leads to a new iteration of the check-and-refine loop.
- Abstraction and Refinement are computationally **expensive**, as they make intensive use of theorem provers. (Worst case: exponential in the number of predicates.)
- The efficiency of the approach depends in a critical way on the number of spurious execution traces allowed by the abstract program.
- The closer is the abstraction to the original program the smaller is the number of spurious execution traces that may be necessary to analyse.

The big challenge is to find a **finer grained model**, in order to decrease the number of loops.

We propose **Linear Programs** as a more suitable abstraction for Software Model Checking [Armando et al., ICFEM 04].

Linear Programs

Imperative programs such that:

- variables range over a numerical domain \mathcal{D}
- expressions are of the form

$$c_0 + c_1x_1 + \dots + c_nx_n,$$

where c_0, \dots, c_n are numeric constants and x_1, \dots, x_n are program variables ranging over \mathcal{D} .

We then apply interprocedural data-flow analysis *à la* Reps-Horwiz-Sagiv using Decision Procedures for Linear Arithmetic.

Example of Linear Program

```
1  int numUnits;
2  int level;
3  void getUnit(){
4      int 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             assert(0);
15         }
16         else {
17             gotUnit();}
18     }
19 }
```

Linear Program

The same example analysed before (slightly modified) does not need any abstraction: only one run of the model checker is needed!

- No abstraction.
- No refinement.

Overview of the procedure

Example

```
int x, y;  
main () { (x' = x) ∧ (y' = y)  
  x=0;  
  y=0;  
  while (x<1){  
    x=x+1;  
    p(x, y);  
  }  
  ;  
}  
p(a, b){  
  b=b+a;  
}
```

Overview of the procedure

Example

```
int x, y;  
main () {            $(x' = x) \wedge (y' = y)$   
  x=0;            $(x' = 0) \wedge (y' = y)$   
  y=0;  
  while (x<1){  
    x=x+1;  
    p(x, y);  
  }  
  ;  
}  
p(a, b){  
  b=b+a;  
}
```

Overview of the procedure

Example

```
int x, y;  
main () {             $(x' = x) \wedge (y' = y)$   
  x=0;             $(x' = 0) \wedge (y' = y)$   
  y=0;             $(x' = 0) \wedge (y' = 0)$   
  while (x < 1) {  
    x=x+1;  
    p(x, y);  
  }  
  ;  
}  
p(a, b) {  
  b=b+a;  
}
```

Overview of the procedure

Example

```
int x, y;  
main () {  $(x' = x) \wedge (y' = y)$   
  x=0;  $(x' = 0) \wedge (y' = y)$   
  y=0;  $(x' = 0) \wedge (y' = 0)$   
  while (x < 1) {  $(x' = 0) \wedge (y' = 0)$   
    x=x+1;  
    p(x, y);  
  }  
;  
}  
p(a, b) {  
  b=b+a;  
}
```

Overview of the procedure

Example

```
int x, y;  
main () { (x' = x) ∧ (y' = y)  
  x=0; (x' = 0) ∧ (y' = y)  
  y=0; (x' = 0) ∧ (y' = 0)  
  while (x < 1) { (x' = 0) ∧ (y' = 0)  
    x=x+1; (x' = 1) ∧ (x = 0) ∧ (y' = 0)  
    p(x, y);  
  }  
;  
}  
p(a, b) {  
  b=b+a;  
}
```

Overview of the procedure

Example

```
int x, y;  
main () { (x' = x) ∧ (y' = y)  
  x=0; (x' = 0) ∧ (y' = y)  
  y=0; (x' = 0) ∧ (y' = 0)  
  while (x < 1) { (x' = 0) ∧ (y' = 0)  
    x=x+1; (x' = 1) ∧ (x = 0) ∧ (y' = 0)  
    p(x, y);  
  }  
;  
}  
p(a, b) { (a = 1) ∧ (a' = a) ∧ (b = 0) ∧ (b' = b)  
  b=b+a;  
}
```

Overview of the procedure

Example

```
int x, y;
main () {             $(x' = x) \wedge (y' = y)$ 
  x=0;             $(x' = 0) \wedge (y' = y)$ 
  y=0;             $(x' = 0) \wedge (y' = 0)$ 
  while (x < 1) {             $(x' = 0) \wedge (y' = 0)$ 
    x=x+1;             $(x' = 1) \wedge (x = 0) \wedge (y' = 0)$ 
    p(x, y);
  }
;
}
p(a, b) {
  b=b+a;             $(a = 1) \wedge (a' = a) \wedge (b = 0) \wedge (b' = b)$ 
}             $(a = 1) \wedge (a' = a) \wedge (b = 0) \wedge (b' = 1)$ 
```

Overview of the procedure

Example

```
int x, y;
main () { (x' = x) ∧ (y' = y)
  x=0; (x' = 0) ∧ (y' = y)
  y=0; ((x' = 0) ∧ (y' = 0)) ∨ ((x' = 1) ∧ (x = 1) ∧ (y' = 1) ∧ (y = 0))
  while (x < 1) { (x' = 0) ∧ (y' = 0)
    x=x+1; (x' = 1) ∧ (x = 0) ∧ (y' = 0)
    p(x, y);
  }
;
}
p(a, b) {
  b=b+a; (x = 1) ∧ (x' = 1) ∧ (y = 0) ∧ (y' = 1)
}
```

Overview of the procedure

Example

```
int x, y;
main () { (x' = x) ∧ (y' = y)
  x=0; (x' = 0) ∧ (y' = y)
  y=0; ((x' = 0) ∧ (y' = 0)) ∨ ((x' = 1) ∧ (x = 1) ∧ (y' = 1) ∧ (y = 0))
  while (x < 1) { (x' = 0) ∧ (y' = 0)
    x=x+1; (x' = 1) ∧ (x = 0) ∧ (y' = 0)
    p(x, y);
  } (x' = 1) ∧ (x = 1) ∧ (y' = 1) ∧ (y = 0)
;
}
p(a, b) {
  b=b+a; (x = 1) ∧ (x' = 1) ∧ (y = 0) ∧ (y' = 1)
}
```

Overview of the procedure

Example

```
int x, y;
main () { (x' = x) ∧ (y' = y)
  x=0; (x' = 0) ∧ (y' = y)
  y=0; ((x' = 0) ∧ (y' = 0)) ∨ ((x' = 1) ∧ (x = 1) ∧ (y' = 1) ∧ (y = 0))
  while (x < 1) { (x' = 0) ∧ (y' = 0)
    x=x+1; (x' = 1) ∧ (x = 0) ∧ (y' = 0)
    p(x, y);
  } (x' = 1) ∧ (x = 1) ∧ (y' = 1) ∧ (y = 0)
; (x' = 1) ∧ (x = 1) ∧ (y' = 1) ∧ (y = 0)
}
p(a, b) {
  b=b+a; (x = 1) ∧ (x' = 1) ∧ (y = 0) ∧ (y' = 1)
}
```

The EUREKA library of Benchmarks

- Aim: assess the scalability of the model checkers;
- Programs are parametric in a positive integer N (the higher N , the more difficult the program)

Program name	data	control	iteration	recursion
swap_seq.c(N)	✓			
swap_iter.c(N)	✓	✓	✓	
parity.c(N)	✓	✓		✓
delay_iter.c(N)	✓	✓	✓	
delay_recur.c(N)	✓	✓	✓	✓
sum.c(N)	✓	✓	✓	

The EUREKA library of Benchmarks

- Aim: assess the scalability of the model checkers;
- Programs are parametric in a positive integer N (the higher N , the more difficult the program)

Program name	data	control	iteration	recursion
<code>swap_seq.c(N)</code>	✓			
<code>swap_iter.c(N)</code>	✓	✓	✓	
<code>parity.c(N)</code>	✓	✓		✓
<code>delay_iter.c(N)</code>	✓	✓	✓	
<code>delay_recur.c(N)</code>	✓	✓	✓	✓
<code>sum.c(N)</code>	✓	✓	✓	

Some Experimental Results

Comparison with BLAST

Numbers in table: the hardest instance (N) of the programs that could be verified by the tools.

Program	Eureka	BLAST
swap_seq.c(N)	500	no
swap_iter.c(N)	100	1
parity.c(N)	50	no
delay_iter.c(N)	150	1000
delay_recur.c(N)	39	no
sum.c(N)	3	2

Some Experimental Results

Comparison with BLAST

Numbers in table: the hardest instance (N) of the programs that could be verified by the tools.

Program	Eureka	BLAST
swap_seq.c(N)	500	no
swap_iter.c(N)	100	1
parity.c(N)	50	no
delay_iter.c(N)	150	1000
delay_recur.c(N)	39	no
sum.c(N)	3	2

Beyond Linear Programs

- Our current work deals with the extension of our model checking procedure to support linear programs featuring:
 - a symbol for *undefined* values ($*$),
 - conditional expressions ($e_1 ? e_2 : e_3$).
- This extension is particularly important as it paves the way to the construction of model checking procedures for wider classes of imperative programs such as, e.g., **Linear Programs with Arrays**.
- We show an abstraction method from Linear Programs with Arrays to Linear Programs.

Abstracting Linear Programs with Arrays

Let a be an array, and let $R(a) = \{k_1, \dots, k_n\}$ be a set of array indexes.

Expressions \rightarrow Conditional Expressions with *undefined*

$$\begin{array}{c} a[e] \\ \downarrow \\ (e == k_1 ? a_{k_1} : (e == k_2 ? a_{k_2} : \dots (e == k_n ? a_{k_n} : *) \dots)) \end{array}$$

Assignments \rightarrow Parallel Assignments

$$\begin{array}{c} a[e_1] = e_2; \\ \downarrow \\ a_{k_1}, \dots, a_{k_n} = (e_1 == k_1 ? e_2 : a_{k_1}), \dots, (e_1 == k_n ? e_2 : a_{k_n}); \end{array}$$

where every a_{k_i} models the $(k_i)^{th}$ array element.

(If $n = 0$, the assignment above reduces to a skip (;) statement.)

Abstracting Linear Programs with Arrays

Let a be an array, and let $R(a) = \{k_1, \dots, k_n\}$ be a set of array indexes.

Expressions \rightarrow Conditional Expressions with *undefined*

$$\begin{array}{c} a[e] \\ \downarrow \\ (e == k_1 ? a_{k_1} : (e == k_2 ? a_{k_2} : \dots (e == k_n ? a_{k_n} : *) \dots)) \end{array}$$

Assignments \rightarrow Parallel Assignments

$$\begin{array}{c} a[e_1] = e_2; \\ \downarrow \\ a_{k_1}, \dots, a_{k_n} = (e_1 == k_1 ? e_2 : a_{k_1}), \dots, (e_1 == k_n ? e_2 : a_{k_n}); \end{array}$$

where every a_{k_i} models the $(k_i)^{th}$ array element.

(If $n = 0$, the assignment above reduces to a skip (;) statement.)

Abstracting Linear Programs with Arrays

Let a be an array, and let $R(a) = \{k_1, \dots, k_n\}$ be a set of array indexes.

Expressions \rightarrow Conditional Expressions with *undefined*

$$\begin{array}{c} a[e] \\ \downarrow \\ (e == k_1 ? a_{k_1} : (e == k_2 ? a_{k_2} : \dots (e == k_n ? a_{k_n} : *) \dots)) \end{array}$$

Assignments \rightarrow Parallel Assignments

$$\begin{array}{c} a[e_1] = e_2; \\ \downarrow \\ a_{k_1}, \dots, a_{k_n} = (e_1 == k_1 ? e_2 : a_{k_1}), \dots, (e_1 == k_n ? e_2 : a_{k_n}); \end{array}$$

where every a_{k_i} models the $(k_i)^{th}$ array element.

(If $n = 0$, the assignment above reduces to a skip (;) statement.)

Original Program

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

Question

Is line 10 reachable?

Method

We abstract every occurrence of array elements

Original Program

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

Question

Is line 10 reachable?

Method

We abstract every occurrence of array elements

Original Program

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

Abstraction wrt. $R(a) = \emptyset$

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

Question

Is line 10 reachable?

Answer

Yes! The trace is:

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

Abstraction wrt. $R(a) = \emptyset$

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

Question

Is line 10 reachable?

Answer

Yes! The trace is:

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

Abstraction wrt. $R(a) = \emptyset$

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

Trace Simulation

- The simulation step builds a formula ϕ in the combined theory of linear arithmetics + arrays s.t.
UNSAT(ϕ) \leftrightarrow the trace is feasible.
- The decision procedure returns that *SAT*(ϕ).
- By inspecting the model it can be determined that this is due to array index 1.

Refinement

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

Original Program

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

Trace Simulation

- The simulation step builds a formula ϕ in the combined theory of linear arithmetics + arrays s.t.
UNSAT(ϕ) \leftrightarrow the trace is feasible.
- The decision procedure returns that *SAT*(ϕ).
- By inspecting the model it can be determined that this is due to array index 1.

Refinement

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

Original Program

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

Abstraction wrt. $R(a) = \{1\}$

```
1  int i , a1;
2  void main(){
3      a1=(1==1)?1:*;
4      i=0;
5      while ((( (i==1)?a1:*)!=1)&& i < 30){
6          a1=(i==1)?2*i : a1;
7          i = i+1;
8      }
9      if (i > 1)
10     ERROR: ;
11 }
```

Original Program

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

Example

Abstraction wrt. $R(a) = \{1\}$

```
1  int i, a1;
2  void main(){
3      a1=(1==1)?1:*;
4      i=0;
5      while (((i==1)?a1:*)!=1)&& i < 30){
6          a1=(i==1)?2*i : a1;
7          i = i+1;
8      }
9      if(i > 1)
10     ERROR: ;
11 }
```

Question

Is line 10 reachable in the refined program?

Final result

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

Example

Abstraction wrt. $R(a) = \{1\}$

```
1  int i, a1;
2  void main(){
3      a1=(1==1)?1:*;
4      i=0;
5      while (((i==1)?a1:*)!=1)&& i < 30){
6          a1=(i==1)?2*i : a1;
7          i = i+1;
8      }
9      if (i > 1)
10     ERROR: ;
11 }
```

Question

Is line 10 reachable in the refined program?

Final result

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

Experimental Results

- The programs below make use of arrays and arithmetics.
- They have been used as regression tests for EUREKA.

Program	EUREKA	BLAST	CBMC
array_init.c	safe	error	safe
array_init_assign.c	safe	unsafe	safe
complex_guard.c	safe	unsafe	safe
simple_swap.c	safe	safe	safe
sequential_swap_call.c	safe	error	safe
simple_array_inversion.c	safe	unsafe	safe
bubblesort_inner_loop.c	unsafe	error	unsafe
bubblesort.c	unsafe	safe	unsafe
array_max.c	safe	error	safe
wrong_loop.c	unsafe	error	mout
loop_on_input.c	unsafe	unsafe	mout
simple_control_on_input.c	unsafe	unsafe	mout

Experimental Results

- The programs below make use of arrays and arithmetics.
- They have been used as regression tests for EUREKA.

Program	EUREKA	BLAST	CBMC
array_init.c	safe	error	safe
array_init_assign.c	safe	unsafe	safe
complex_guard.c	safe	unsafe	safe
simple_swap.c	safe	safe	safe
sequential_swap_call.c	safe	error	safe
simple_array_inversion.c	safe	unsafe	safe
bubblesort_inner_loop.c	unsafe	error	unsafe
bubblesort.c	unsafe	safe	unsafe
array_max.c	safe	error	safe
wrong_loop.c	unsafe	error	mout
loop_on_input.c	unsafe	unsafe	mout
simple_control_on_input.c	unsafe	unsafe	mout

Experimental Results

- The programs below make use of arrays and arithmetics.
- They have been used as regression tests for EUREKA.

Program	EUREKA	BLAST	CBMC
array_init.c	safe	error	safe
array_init_assign.c	safe	unsafe	safe
complex_guard.c	safe	unsafe	safe
simple_swap.c	safe	safe	safe
sequential_swap_call.c	safe	error	safe
simple_array_inversion.c	safe	unsafe	safe
bubblesort_inner_loop.c	unsafe	error	unsafe
bubblesort.c	unsafe	safe	unsafe
array_max.c	safe	error	safe
wrong_loop.c	unsafe	error	mout
loop_on_input.c	unsafe	unsafe	mout
simple_control_on_input.c	unsafe	unsafe	mout

Symbolic Model Checking: Summary.

- We have provided a formal semantics and a symbolic model checking procedure for the class of Linear Programs with the *undefined* symbol and with conditional expressions.
- We have proposed an **alternative abstraction and refinement schema** for a subset of the C programming language that employs linear arithmetics and arrays.
- Experimental results obtained with our prototype model checker EUREKA indicate that our procedure correctly analyses programs on which other **state-of-the-art tools** either **fail** or **provide incorrect answers**.

1 Introduction

2 Symbolic Software Model Checking

- A few words about Model Checking
- The state of the art
- The EUREKA approach at AI-Lab

3 Bounded Model Checking of Software

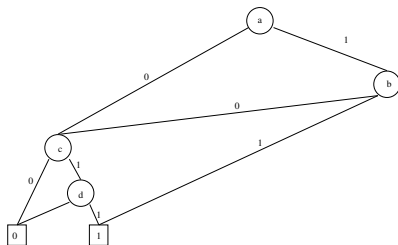
- A few words about Bounded Model Checking
- The state of the art
- The EUREKA approach at AI-Lab

4 Conclusion

Issues in Model Checking

In Model Checking:

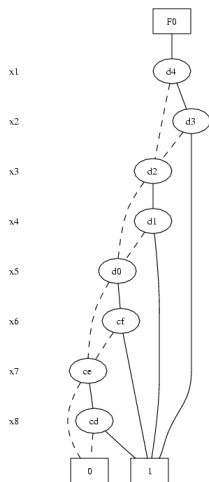
- States (and sets of states) are usually represented with boolean formulas.
- Formulas are in turn encoded using BDDs. Ex.: $(a \wedge b) \vee (b \wedge d)$



Model is hard to represent in a compact way because size of BDDs heavily depends on variable ordering, and may become exponential.

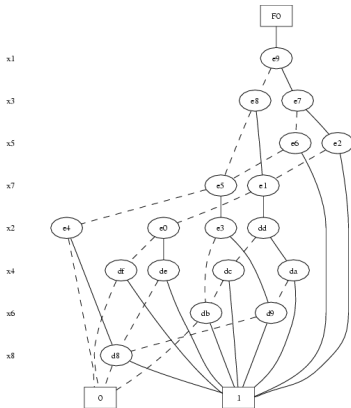
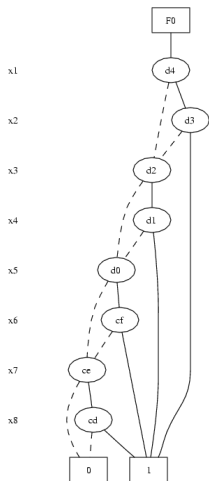
Issues in Model Checking

Example BDD size explosion



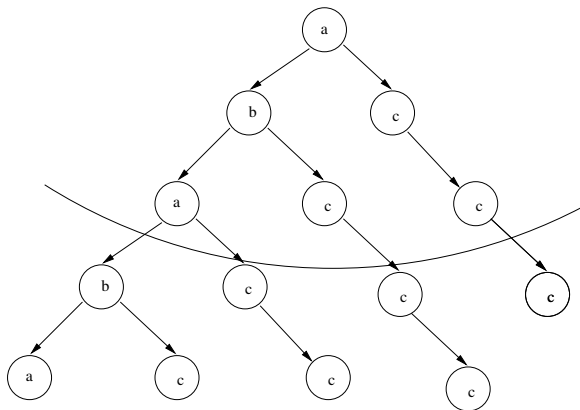
Issues in Model Checking

Example BDD size explosion



What is Bounded Model Checking?

- In Model Checking, a Kripke Structure is capable to describe **infinite sequences** of transitions (paths).
- If we **bound the length of paths** it is possible to encode a Kripke Structure and its desired properties into a **propositional formula**.



What is Bounded Model Checking? (2)

- The formula describes all the behaviours of the system and the properties: every model of the formula (if any) corresponds to a path leading to a violation of a property.
- We call **SAT problem** the problem of determining whether a boolean formula is satisfiable or not.
- Given
 - a set of propositional variables $\{x_1, x_2, \dots, x_m\}$, and
 - a propositional formula $\Phi(x_1, x_2, \dots, x_m)$is there an assignment $\{v_1, v_2, \dots, v_m\}$ with $v_i \in \{true, false\}$ such that $\Phi(v_1, v_2, \dots, v_m)$ is *true* ?
- Example: $(a \wedge b) \vee (c \wedge d)$

$A_1 : a = true, b = true, c = false, d = false$

$A_2 : a = false, b = false, c = true, d = true$

$A_3 : a = true, b = false, c = true, d = true$

\vdots

What is Bounded Model Checking? (2)

- The formula describes all the behaviours of the system and the properties: every model of the formula (if any) corresponds to a path leading to a violation of a property.
- We call **SAT problem** the problem of determining whether a boolean formula is satisfiable or not.
- Given
 - a set of propositional variables $\{x_1, x_2, \dots, x_m\}$, and
 - a propositional formula $\Phi(x_1, x_2, \dots, x_m)$is there an assignment $\{v_1, v_2, \dots, v_m\}$ with $v_i \in \{true, false\}$ such that $\Phi(v_1, v_2, \dots, v_m)$ is *true* ?
- Example: $(a \wedge b) \vee (c \wedge d)$

$A_1 : a = true, b = true, c = false, d = false$

$A_2 : a = false, b = false, c = true, d = true$

$A_3 : a = true, b = false, c = true, d = true$

\vdots

\vdots

What is Bounded Model Checking? (2)

- The formula describes all the behaviours of the system and the properties: every model of the formula (if any) corresponds to a path leading to a violation of a property.
- We call **SAT problem** the problem of determining whether a boolean formula is satisfiable or not.
- Given
 - a set of propositional variables $\{x_1, x_2, \dots, x_m\}$, and
 - a propositional formula $\Phi(x_1, x_2, \dots, x_m)$is there an assignment $\{v_1, v_2, \dots, v_m\}$ with $v_i \in \{true, false\}$ such that $\Phi(v_1, v_2, \dots, v_m)$ is *true* ?
- Example: $(a \wedge b) \vee (c \wedge d)$

$A_1 : a = true, b = true, c = false, d = false$

$A_2 : a = false, b = false, c = true, d = true$

$A_3 : a = true, b = false, c = true, d = true$

\vdots

\vdots

The State of the Art in Bounded Model Checking of Software

- The state of the art is represented by a tool called **CBMC** (C Bounded Model Checker), developed at Carnegie Mellon University.
- It is capable to verify realistic C programs thanks to efficient SAT solvers.

Bounded Model Checking of Sequential Software

The procedure used in CBMC consists of 4 steps:

- 1 **Preprocess**: the program is transformed into an equivalent one that uses only `while` and `if`.
- 2 **Unwind loops** (n times):

$$\begin{aligned}\text{while}(e) \text{ instr} &\longrightarrow \text{if}(e) \{ \text{instr}; \text{while}(e) \text{ instr} \} \\ \text{while}(e) \text{ instr} &\longrightarrow \text{assert}(!e); \end{aligned}$$

- 3 **Build** a quantifier-free formula Φ such that any model of Φ corresponds to an execution path ending with the violation of an `assert` statement.
- 4 **Check** whether Φ is satisfiable. If any unwinding assertion is violated, then increase n and go to 2 otherwise return result of the analysis.

Let p the transformed program and g a formula representing the initial states.

Then $\Phi := (\mathcal{C}(p, g) \wedge \neg \mathcal{P}(p, g))$, where

- $\mathcal{C}(p, g)$ represents the set of execution paths of p starting in an initial state and
- $\mathcal{P}(p, g)$ represents the set of states for which the assert statements succeed.

Thus, any model of Φ corresponds to an execution path that starts in an initial state and violates an assert statement.

Building Φ : example

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

$$\begin{aligned} \mathcal{C} &= \{ \text{true} \supset a_1 = \text{write}(a_0, 0, 1) \wedge (\text{false} \supset (a_1 = a_0)), \\ &\quad (\text{true} \supset a_2 = \text{write}(a_1, 1, 1) \wedge (\text{false} \supset (a_2 = a_1)), \\ &\quad (\text{true} \supset (i_1 = 2)) \wedge (\text{false} \supset (i_1 = i_0)), \\ &\quad ((i_1 < n) \supset a_3 = \text{write}(a_2, i_1, \text{read}(a_2, i_1 - 1))) \wedge (\neg(i_1 < n) \supset (a_3 = a_2)), \\ &\quad ((i_1 < n) \supset i_2 = (i_1 + 1)) \wedge (\neg(i_1 < n) \supset (i_2 = i_1)) \} \\ \mathcal{P} &= \{ ((i_1 < n) \supset \neg(i_2 < n)) \} \end{aligned}$$

Solving Φ with a SAT solver

- Basic data types (e.g. `int`, `float`) modelled as fixed size bit-vectors.
- Arithmetic operations are thus converted into equivalent combinations of bit-vectors operations.
- Arrays with m elements modelled as m variables.
- The resulting formula is a boolean combination of bit-vector equations that can be readily compiled into an equivalent **boolean formula**.
- The formula is then solved using a SAT solver.

Solving Φ with a SAT solver (2)

Pros:

- Faithful modelling of the data and operations upon them.

Cons:

- Size of the encoding increases with the size of the arrays used in the program.

Statement	SAT encoding
$a[e_1] = e_2$	$\bigwedge_{i=0}^{dim(a)-1} a_k[i] = ((i = e_1) ? e_2 : a_{k-1}[i])$
$x = a[e]$	$\bigwedge_{i=0}^{dim(a)-1} (i = e) \supset x_j = a_k[i]$

A different encoding: the theory of arrays.

Arrays are nothing but functions that bind indexes to elements:

$$a : \mathbb{N} \rightarrow D.$$

Functional symbols *select* e *store* model access to and storage of elements. They are defined as follows:

Axioms

$$\begin{aligned} \forall a : \text{Array}, \forall i : \text{Index}, \forall e : \text{Element}, \\ \forall a, \forall i, \forall e : \text{select}(\text{store}(a, i, e), i) &= e, \\ \forall a, \forall i, \forall j, \forall e : i \neq j \supset \text{select}(\text{store}(a, i, e), j) &= \text{select}(a, j). \end{aligned}$$

A different encoding: the theory of arrays.

What if we model arrays as.. arrays?

Statement	SMT encoding	SAT encoding
$a[e_1]=e_2$	$a_k = \text{store}(a_{k-1}, e_1, e_2)$	$\bigwedge_{i=0}^{\text{dim}(a)-1} a_k[i] = ((i = e_1) ? e_2 : a_{k-1}[i])$
$x=a[e]$	$x_j = \text{select}(a_k, e)$	$\bigwedge_{i=0}^{\text{dim}(a)-1} (i = e) \supset x_j = a_k[i]$

- SAT encoding grows with the size of the arrays;
- Our approach is independent from the size of the arrays.

A different encoding: the theory of arrays.

What if we model arrays as.. arrays?

Statement	SMT encoding	SAT encoding
$a[e_1]=e_2$	$a_k = \text{store}(a_{k-1}, e_1, e_2)$	$\bigwedge_{i=0}^{\text{dim}(a)-1} a_k[i] = ((i = e_1) ? e_2 : a_{k-1}[i])$
$x=a[e]$	$x_j = \text{select}(a_k, e)$	$\bigwedge_{i=0}^{\text{dim}(a)-1} (i = e) \supset x_j = a_k[i]$

- SAT encoding grows with the size of the arrays;
- Our approach is independent from the size of the arrays.

A different encoding: the theory of arrays.

What if we model arrays as.. arrays?

Statement	SMT encoding	SAT encoding
$a[e_1] = e_2$	$a_k = \text{store}(a_{k-1}, e_1, e_2)$	$\bigwedge_{i=0}^{\text{dim}(a)-1} a_k[i] = ((i = e_1) ? e_2 : a_{k-1}[i])$
$x = a[e]$	$x_j = \text{select}(a_k, e)$	$\bigwedge_{i=0}^{\text{dim}(a)-1} (i = e) \supset x_j = a_k[i]$

- SAT encoding grows with the size of the arrays;
- Our approach is independent from the size of the arrays.

A different encoding: SMT.

SMT

Given a decidable theory \mathcal{T} and a quantifier-free formula ϕ in the same language as \mathcal{T} , by *satisfiability modulo theory (SMT)* we mean the problem of determining whether $\mathcal{T} \cup \{\phi\}$ is satisfiable.

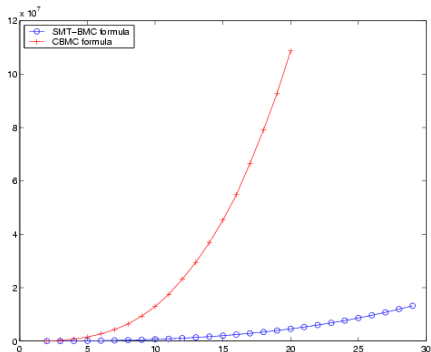
SMT solver

By *SMT solver for \mathcal{T}* we mean a program capable to solve the SMT problem for \mathcal{T} .

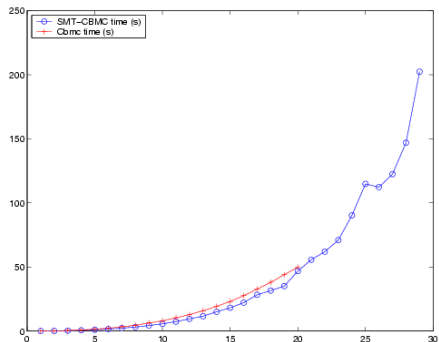
Decidable theories of interest:

Linear Arithmetics, lists, arrays, records, and bit-vectors.

Experimental Results: Selection Sort

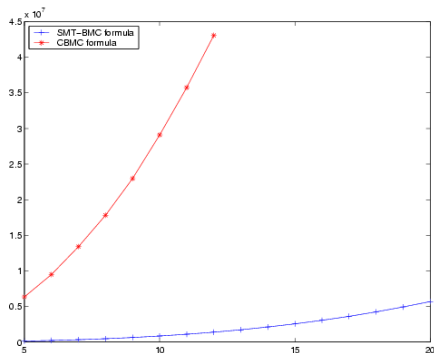


Size of the formulae

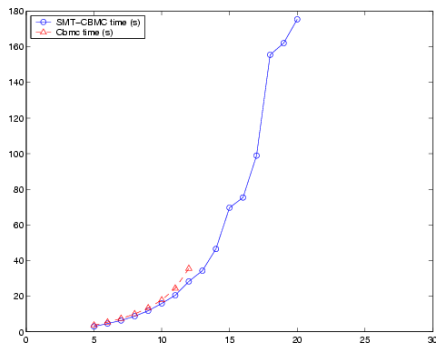


CPU time

Experimental Results: Bellman-Ford



Size of the formulae



CPU time

Bounded Model Checking: Summary

- State-of-the-art techniques adopt SAT procedures in Bounded Model Checking;
- We have shown that the SMT encoding leads to more compact representations.
- Experimental results show that our prototype implementation compares favourably and, for some classes of programs, outperforms the state-of-the-art applications.

- 1 Introduction
- 2 Symbolic Software Model Checking
 - A few words about Model Checking
 - The state of the art
 - The EUREKA approach at AI-Lab
- 3 Bounded Model Checking of Software
 - A few words about Bounded Model Checking
 - The state of the art
 - The EUREKA approach at AI-Lab
- 4 Conclusion

- We have introduced (combination of) decision procedures into Software Model Checking (both Bounded and Symbolic).
- In Software Verification, SMT Decision Procedures can be a complementary approach to SAT based procedures.
- Computer experiments show that our approach often outperforms the state of the art.

Thank you!

<http://www.ai.dist.unige.it/eureka>