Model Checking Imperative Programs

Jacopo Mantovani

(joint work with A. Armando, M. Benerecetti, and C. Castellini)

Artificial Intelligence Laboratory
University of Genova, Italy

DSI Milano, May 19, 2005

Outline of the talk

1. Motivation
2. Verification - A brief overview
   - Explicit state space exploration
   - Abstract Interpretation
   - Model Checking
3. Software Model Checking
   - Issues in Software Model Checking
   - The state of the art
   - The EUREKA approach
   - Experimental Results

Motivation (1)

February 25, 1991:

- a Patriot Missile system that had been running for over 100 hours at Dhahran, Saudi Arabia, failed to intercept a SCUD missile.
- The SCUD hit an Army Barracks, killing 28 Americans.
- The problem is that time is stored to an accuracy of 1/10th of a second. A 24-bit register does not have enough precision to store 0.1, so a small fraction of each second is lost. The result is that the register used to keep track of time is off by 0.0001% of the amount of time that the system has been in operation.

Motivation (2)

October 30, 1994:

- A bug in the Intel Pentium:

$$x = 4195835, y = 3145727, z = x - (x/y) * y$$

Motivation (2)

October 30, 1994:

- A bug in the Intel Pentium:

$$x = 4195835, y = 3145727, z = x - (x/y) * y$$

- It came out that the chip gave as answer $z = 256$.
- Intel was forced to offer to replace all flawed Pentium processors.

Motivation (3)

June 4, 1996:

- The ESA rocket named Ariane 5 exploded just 40 seconds after lift-off.
- The destroyed rocket and its cargo were evaluated $500 million.
- Cause: a 64 bit floating point number relating to the horizontal velocity of the rocket wrt. platform was converted to a 16 bit signed integer.

Motivation (4)

For more interesting cases, you may have a look at:

- ACM Forum on Risks to the Public in Computers and Related Systems: `www.risks.org`
- Peter Neumann's homepage `www.csl.sri.com/users/neumann`

Last piece of data[1]:

- Cost of inadequate Software Testing Infrastructure to the U.S.A.: $59.5 Billions.
- Potential portion of cost reduced from feasible improvements: $22.2 Billions.

[1] Source: NIST Planning Report 02-3, June 2002

The context

Formal Verification is the act of proving or disproving the correctness of a (hardware or software) system with respect to a certain formal specification or property, using formal methods (logic, automata, etc.).

Formal Verification ensures consistency with its specification for all possible input patterns.

Formal Methods - Description of a System

For the aim of this talk, a system S can be represented by a triple:

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

where

- Σ is the set of states
- $\mathcal{I} \subseteq \Sigma$ is the set of initial states
- $\rightsquigarrow \subseteq \Sigma \times \Sigma$ is the transition relation

Formal Methods - Description of a System

For the aim of this talk, a system S can be represented by a triple:

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

where

- Σ is the set of states
- $\mathcal{I} \subseteq \Sigma$ is the set of initial states
- $\rightsquigarrow \subseteq \Sigma \times \Sigma$ is the transition relation

States

State: a "snapshot" of the system. Usually involves valuations of the system variables $\omega : V_S \rightarrow \mathcal{D}$.

Formal Methods - Reachability (1)

As example property, in this talk we only consider reachability.
The operator *post* on Σ is defined as

$$post(\Sigma_1) = \{\sigma' \in \Sigma \mid exists\ \sigma \in \Sigma_1 : \sigma \rightsquigarrow \sigma'\}$$

for all $\Sigma_1 \subseteq \Sigma$. The set of reachable states is defined as the closure of $\mathcal{I}$ under *post*:

$$post^*(\mathcal{I}) = \mathcal{I} \cup post(\mathcal{I}) \cup \ldots$$

The set $post^*(\mathcal{I})$ is also known as the *least fix point* of *post* that contains $\mathcal{I}$.

Formal Methods - Reachability (2)

A finite computation can lead to:

- *erroneous states* $E : E \subseteq \Sigma$;
- *safe states* $F : F \subseteq \Sigma$ and $F = \Sigma \setminus E$.

The system S executes correctly iff no unsafe state is reachable:

$$post^*(\mathcal{I}) \cap E = \emptyset,$$

or, analogously, if

$$post^*(\mathcal{I}) \subseteq F.$$

Moreover, given a specification $Q : Q \subseteq \Sigma$, the system is correct wrt Q if

$$post^*(\mathcal{I}) \subseteq Q.$$

Formal Methods - Reachability (3)

- Reachability is in general undecidable; [Esparza, 97]
- However, there are many interesting cases to consider for which reachability becomes decidable:
  - finite-state automata,
  - timed automata, [Alur and Dill, 94]
  - pushdown automata,
  - ...

Formal Methods - An Overview

Formal methods include:

- State space exploration
- Abstract Interpretation
- Automatic Theorem Proving
- Model Checking
- ...

Formal Methods - Explicit State space exploration (1)

- States are explicitly represented.
- For hardware circuits, states are valuations of boolean variables of the formula representing them into $\{0, 1\}$.
- All the states have to be traversed in order to guarantee *verification*.

Formal Methods - Explicit State space exploration (1)

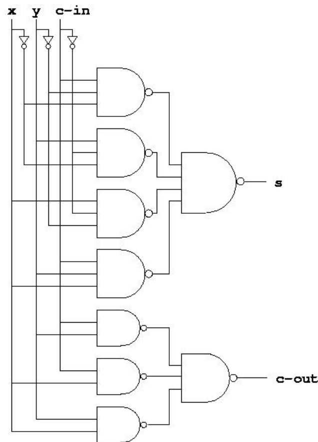
- States are explicitly represented.
- For hardware circuits, states are valuations of boolean variables of the formula representing them into $\{0, 1\}$.
- All the states have to be traversed in order to guarantee *verification*.

Main drawback

Combinatorial state space explosion!

Formal Methods - Explicit State space exploration (2)

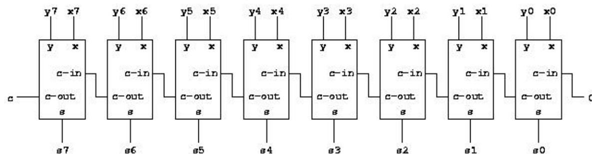
Example: full-adder.



Number of possible input states: 2^3.

Formal Methods - Explicit State space exploration (3)

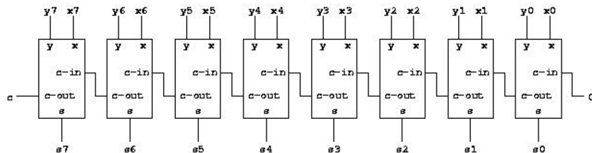
Example: 8-bit adder



Number of possible input states: $2^8 \times 2^8 = 2^{16} = 65536$!

Formal Methods - Explicit State space exploration (3)

Example: 8-bit adder



Number of possible input states: $2^8 \times 2^8 = 2^{16} = 65536$!

What if we want to verify a 32-bit adder?
Number of possible input states: $2^{64} \approx 1.8 \times 10^{19}$!!

Formal Methods - Abstract Interpretation (1)

- Introduced by Patrick and Rhadia Cousot in 1977.
- The system S is *abstracted* by a “simpler” one, $S^{\#}$.
- Then we apply verification techniques on $S^{\#}$.
- Aim of abstract interpretation is to make the computation of fix-points (in our case, $post^*$) easier.

Formal Methods - Abstract Interpretation (2)

Definition

Let $L, L^{\#}$ be complete lattices. Then,

$$\langle L, \sqsubseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle L^{\#}, \sqsubseteq^{\#} \rangle$$

is a Galois Connection iff

$$\forall x \in L, \forall y \in L^{\#} \; \alpha(x) \sqsubseteq^{\#} y \Leftrightarrow x \sqsubseteq \gamma(y),$$

where α and γ are monotonic functions that relate concrete and abstract domains.

Formal Methods - Abstract Interpretation (3)

A possible abstraction in the case of reachability:

- We can relate the "concrete" transition system S with a simpler one $S^{\#}$ (that preserves the original properties);
- We then compare concrete and abstract (sets of) states.
- Concrete and abstract states are related by a surjective function $h : \Sigma \rightarrow \Sigma^{\#}$ such that $h(\sigma) = \sigma^{\#}$ for all $\sigma \in \Sigma$.

Formal Methods - Abstract Interpretation (4)

Definition (Abstraction)

Let $\Sigma_1 \subseteq \Sigma$. Then, the abstraction α is a function $\alpha : 2^{\Sigma} \longrightarrow 2^{\Sigma^{\#}}$ such that

$$\alpha(\Sigma_1) = \{h(\sigma) \mid \sigma \in \Sigma_1\}.$$

Formal Methods - Abstract Interpretation (4)

Definition (Abstraction)

Let $\Sigma_1 \subseteq \Sigma$. Then, the abstraction α is a function $\alpha : 2^{\Sigma} \rightarrow 2^{\Sigma^{\#}}$ such that

$$\alpha(\Sigma_1) = \{h(\sigma) \mid \sigma \in \Sigma_1\}.$$

Definition (Concretization)

The concretization γ is a function $\gamma : 2^{\Sigma^{\#}} \rightarrow 2^{\Sigma}$ such that, for all $\Sigma_1^{\#} \subseteq \Sigma^{\#}$,

$$\gamma(\Sigma_1^{\#}) = \{\sigma \in \Sigma \mid h(\sigma) \in \Sigma_1^{\#}\}.$$

Formal Methods - Abstract Interpretation (5)

Definition (Galois connection)

The functions α and γ form a *Galois connection*:

$$\langle 2^{\Sigma}, \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle 2^{\Sigma^{\#}}, \subseteq \rangle.$$

Some Properties

- $\forall \Sigma_1 \in \Sigma,\ \Sigma_1 \subseteq \gamma(\alpha(\Sigma_1))$
- $\forall \Sigma_1^{\#} \in \Sigma^{\#},\ \Sigma_1^{\#} \subseteq \alpha(\gamma(\Sigma_1^{\#}))$
- For $post^{\#}(\Sigma_1^{\#}) = \alpha(post(\gamma(\Sigma_1^{\#})))$,

$$lfp(post) \subseteq \gamma(lfp(post^{\#})).$$

Formal Methods - Abstract Interpretation (6)

Drawback

$$lfp(post) \subseteq \gamma(lfp(post^{\#})).$$

- if the abstract program satisfies the properties, so does the concrete one.
- if it does not, no conclusive answer!

Formal Methods - Model Checking

In Model Checking, a system S is formalized 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
- $\mathcal{L} : \Sigma \to \Sigma^{AP}$ is a function that labels each state with the set $AP' \subseteq AP$ of atomic propositions that are true in that state.

Model Checking - Specification of the properties

Desirable properties of a system are usually specified in some temporal logics like *LTL*, *CTL*, CTL^*.
Examples:

- *LTL*:
  - Mutex: $\neg \mathbf{F}(c_1 \wedge c_2)$
  - Safety: $\mathbf{G}\phi$
  - Liveness: $\mathbf{GF}\mathit{active}$
- *CTL*:
  - Possibility: $\mathbf{EF}\phi$ (Reachability)
  - Invariance: $\mathbf{AG}\phi$
  - Unavoidableness: $\mathbf{AF}\phi$
- CTL^*: a combination of *LTL* and *CTL*..

Model Checking - The Problem

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

Model Checking - Pros & Cons.

Advantages

- Fully automatic ("push-button")
- No proofs needed!
- A counterexample is always given in case an error is found.

Drawbacks:

- Construction of a good model is hard;
- Size of the model may explode;
- Model is hard to represent in a compact way (BDDs heaviliy depend on variable ordering, and may become exponential).

Model Checking - BDDs

Applications

- 1992, Clarke: IEEE Futurebus+ cache coherence protocol (SMV),
- 1996, Bell Labs: AT&T High-Level Data Link Controller,
- 1996, Rahimi and Lear: PowerPC 620,
- 2000, Tacchella et al.: part of contol logic of Intel Pentium IV,
- 2001, Stanford FLASH multiprocessor cache coherence protocol,
- etc..

What if systems are not finite? Model Checking becomes undecidable.

Issues in Software Model Checking

- Software systems are inherently infinite state.
- The "usual" model checking formalization (Finite Kripke structures) does not fit anymore.
- The problem becomes undecidable.
- Even more "complexity" is added if the programs are concurrent.

Issues in Software Model Checking

- Software systems are inherently infinite state.
- The “usual” model checking formalization (Finite Kripke structures) does not fit anymore.
- The problem becomes undecidable.
- Even more “complexity” is added if the programs are concurrent.

Main actors:

- The SLAM Toolkit (Ball et al., Microsoft Research),
- BLAST (Henzinger et al., Berkeley University),
- MAGIC (Clarke et al., CMU),
- Java PathFinder (Visser et al, NASA).

Boolean Abstraction and Refinement

Given a program P and a property ϕ to check,

1. Build an abstract model $P^{\#}$ of P
2. Model check $P^{\#}$

Boolean Abstraction and Refinement

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 $C^{\#}$:

Boolean Abstraction and Refinement

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.

Boolean Abstraction and Refinement

- First introduced by Ball and Rajamani (Microsoft Research),
- C language as representative for the imperative paradigm,
- Combines Model Checking and Path Simulation techniques,
- Very successful on control-intensive programs like device drivers,
- Tested on Windows NT device drivers (many bugs found!).

Abstraction

Predicate abstraction

Given

- a concrete program P,
- a set of predicates $\mathcal{K}$,

let *abs* be the abstraction function such that

$$abs(P, \mathcal{K}) \rightarrow B,$$

where B is a Boolean Program.

Abstraction

Predicate abstraction

Given

- a concrete program P,
- a set of predicates $\mathcal{K}$,

let *abs* be the abstraction function such that

$$abs(P, \mathcal{K}) \rightarrow B,$$

where B is a Boolean Program.

Boolean Programs

Imperative programs whose variables are of type boolean only.

Abstraction

From concrete to Boolean Programs

The abstraction function is such that

- it preserves the control-flow graph of the concrete program. Therefore, Boolean Programs support:
  - iteration constructs (`while`),
  - conditional constructs (`if`,`assert`),
  - procedure calls (also with recursion) with call-by-value parameter passing;
- each boolean variable b_i is in a one-to-one relation with a concrete predicate $k_i \in \mathcal{K}$.

Boolean Programs are equivalent in power to pushdown automata (model checking is decidable).

Example

```
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 abstract the program with respect to $\mathcal{K} = \emptyset$.

Example

```
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
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 }
```

Example

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

Answer

Yes! The shortest trace is

$$4, 5, 12, 13, 14$$

Question

Is that trace feasible in the concrete program?

Example

```
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:
$\mathtt{numUnits} \neq 0$ at line 5 and
$\mathtt{numUnits} = 0$ at line 13!

Adding predicates

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

Example

```
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

```
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 }
```

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

Example

```
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

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

Adding predicates

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

Example

```
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    } cE:=T;
12    if(cE){
13      if(nU0){
14        ;
15      }
16      else {
17        ;}
18    }
19  }
```

Reachable?

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

Issues

- Abstraction and Refinement are 'expensive', as they make intensive use of theorem provers.
- Boolean abstraction is very coarse, and requires many Abstract-Check-Refine loops in order to output a solution.
- The whole Abstract/Check/Refine loop may not terminate, even by choosing an abstraction (like Boolean Programs) for which the model checking procedure is terminating.

The big challenge is to find a less coarse model, in order to decrease the number of loops.

Linear Programs - The EUREKA approach

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

Linear Programs

Imperative programs such that:

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

$$c_0 + c_1 x_1 + \ldots + c_n x_n,$$

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

Example

```
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 analyzed before (slightly modified) does not need any abstraction: only one run of the model checker is needed!

Overview of the procedure

Similarly to the SLAM's approach (based on the Interprocedural Dataflow Analysis procedure of Reps, Horwitz, Sagiv [POPL, 1995]), the procedure computes reachability of vertices of the control-flow graph, using:

- Path Edges to represent the reachability status of vertices
- Summary Edges to record the input/output behaviour of procedures.

Overview of the procedure

Similarly to the SLAM's approach (based on the Interprocedural Dataflow Analysis procedure of Reps, Horwitz, Sagiv [POPL, 1995]), the procedure computes reachability of vertices of the control-flow graph, using:

- Path Edges to represent the reachability status of vertices
- Summary Edges to record the input/output behaviour of procedures.

No need to compute twice the effects of procedure calls for the same input!

Overview of the procedure

Similarly to the SLAM's approach (based on the Interprocedural Dataflow Analysis procedure of Reps, Horwitz, Sagiv [POPL, 1995]), the procedure computes reachability of vertices of the control-flow graph, using:

- Path Edges to represent the reachability status of vertices
- Summary Edges to record the input/output behaviour of procedures.

No need to compute twice the effects of procedure calls for the same input!

The procedure works by incrementally storing in a worklist the next statement to be analyzed, by computing Path Edges and Summary Edges accordingly, and by removing the processed statement from the worklist.

Overview of the procedure

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

$(x' = x) \wedge (y' = y)$

Overview of the procedure

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

$(x' = x) \wedge (y' = y)$

$(x' = 0) \wedge (y' = y)$

Overview of the procedure

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

Annotations:

- main(){ : $(x' = x) \wedge (y' = y)$
- x=0; : $(x' = 0) \wedge (y' = y)$
- y=0; : $(x' = 0) \wedge (y' = 0)$

Overview of the procedure

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

$(x' = x) \wedge (y' = y)$

$(x' = 0) \wedge (y' = y)$

$(x' = 0) \wedge (y' = 0)$

$(x' = 0) \wedge (y' = 0)$

Overview of the procedure

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

$(x' = x) \wedge (y' = y)$

$(x' = 0) \wedge (y' = y)$

$(x' = 0) \wedge (y' = 0)$

$(x' = 0) \wedge (y' = 0)$

$(x' = 1) \wedge (x = 0) \wedge (y' = 0)$

Overview of the procedure

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

- `main(){`: $(x' = x) \land (y' = y)$
- `x=0;`: $(x' = 0) \land (y' = y)$
- `y=0;`: $(x' = 0) \land (y' = 0)$
- `while(x<1){`: $(x' = 0) \land (y' = 0)$
- `x=x+1;`: $(x' = 1) \land (x = 0) \land (y' = 0)$
- `p(a,b){`: $(a = 1) \land (a' = a) \land (b = 0) \land (b' = b)$

Overview of the procedure

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

$(x' = x) \wedge (y' = y)$

$(x' = 0) \wedge (y' = y)$

$(x' = 0) \wedge (y' = 0)$

$(x' = 0) \wedge (y' = 0)$

$(x' = 1) \wedge (x = 0) \wedge (y' = 0)$

$(a = 1) \wedge (a' = a) \wedge (b = 0) \wedge (b' = b)$

$(a = 1) \wedge (a' = a) \wedge (b = 0) \wedge (b' = 1)$

Overview of the procedure

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

$(x' = x) \wedge (y' = y)$

$(x' = 0) \wedge (y' = y)$

$((x' = 0) \wedge (y' = 0)) \vee ((x' = 1) \wedge (x = 1) \wedge (y' = 1) \wedge (y = 0))$

$(x' = 0) \wedge (y' = 0)$

$(x' = 1) \wedge (x = 0) \wedge (y' = 0)$

$(x = 1) \wedge (x' = 1) \wedge (y = 0) \wedge (y' = 1)$

Overview of the procedure

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

$(x' = x) \wedge (y' = y)$

$(x' = 0) \wedge (y' = y)$

$((x' = 0) \wedge (y' = 0)) \vee ((x' = 1) \wedge (x = 1) \wedge (y' = 1) \wedge (y = 0))$

$(x' = 0) \wedge (y' = 0)$

$(x' = 1) \wedge (x = 0) \wedge (y' = 0)$

$(x' = 1) \wedge (x = 1) \wedge (y' = 1) \wedge (y = 0)$

$(x = 1) \wedge (x' = 1) \wedge (y = 0) \wedge (y' = 1)$

Overview of the procedure

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

$(x' = x) \wedge (y' = y)$

$(x' = 0) \wedge (y' = y)$

$((x' = 0) \wedge (y' = 0)) \vee ((x' = 1) \wedge (x = 1) \wedge (y' = 1) \wedge (y = 0))$

$(x' = 0) \wedge (y' = 0)$

$(x' = 1) \wedge (x = 0) \wedge (y' = 0)$

$(x' = 1) \wedge (x = 1) \wedge (y' = 1) \wedge (y = 0)$

$(x' = 1) \wedge (x = 1) \wedge (y' = 1) \wedge (y = 0)$

$(x = 1) \wedge (x' = 1) \wedge (y = 0) \wedge (y' = 1)$

Overview of the procedure

We represent Path Edges and Summary Edges by means of Abstract Disjunctive Linear Constraints (ADLCs for short).

ADLCs

An Abstract Disjunctive Linear Constraint is an expression of the form

$$\lambda \mathbf{x} \lambda \mathbf{x}'. \bigvee_i \bigwedge_j c_{ij}.$$

Overview of the procedure

We represent Path Edges and Summary Edges by means of Abstract Disjunctive Linear Constraints (ADLCs for short).

ADLCs

An Abstract Disjunctive Linear Constraint is an expression of the form

$$\lambda \mathbf{x} \lambda \mathbf{x}'. \bigvee_i \bigwedge_j c_{ij}.$$

Main operators over ADLCs:

- $Join(P, \tau)$: computes the ADLC resulting from the application of the transition relation τ to an ADLC P,
- $\sqsubseteq$: entailment,
- $\exists \mathbf{x}$: quantifier elimination,
- $\sqcap$, $\sqcup$, $\sim$: and, or, not.

Another Example

```
1  int a[10];
2  main(){
3     int b[2];
4     int i;
5     i=0;
6     while(i<10){
7        a[i]=i+1;
8        i=i+1;
9     }
10    if(a[8]==0){
11    ERROR: ;
12    } else {
13       ;
14    }
15 }
```

Blast Output

```
Ack! The gremlins again
Ack! The gremlins again!
Fatal error:
("No new preds found!..
```

EUREKA Output

```
The input program is safe.
```

Another Example

```
1  int a[10];
2  main(){
3     int b[2];
4     int i;
5     i=0;
6     while(i<10){
7        a[i]=i+1;
8        i=i+1;
9     }
10    if(a[8]==0){
11    ERROR: ;
12    } else {
13       ;
14    }
15 }
```

Blast Output

```
Ack! The gremlins again
Ack! The gremlins again!
Fatal error:
("No new preds found!..
```

EUREKA Output

```
The input program is safe.
```

?

There is an array in the program! It is not a Linear Program!

Linear Programs with Arrays

- Our recent work deals with the abstraction of Linear Programs with Arrays to Linear Programs. How?
- Instead of having the predicates $\mathcal{K}$ as reference, we have a family of sets of array indexes $\{R(a)\}$.

Linear Programs with Arrays

- Our recent work deals with the abstraction of Linear Programs with Arrays to Linear Programs. How?
- Instead of having the predicates $\mathcal{K}$ as reference, we have a family of sets of array indexes $\{R(a)\}$.
- Every expression of the form $a[e]$ is replaced by

$$(e == k_1 \, ? \, a_{k_1} : (e == k_2 \, ? \, a_{k_2} : \ldots (e == k_n \, ? \, a_{k_n} : \mathfrak{u}) \ldots))$$

for each array a, with $R(a) = \{k_1, \ldots, k_n\}$.

Linear Programs with Arrays

- Our recent work deals with the abstraction of Linear Programs with Arrays to Linear Programs. How?
- Instead of having the predicates $\mathcal{K}$ as reference, we have a family of sets of array indexes $\{R(a)\}$.
- Every expression of the form $a[e]$ is replaced by

$$(e == k_1\ ?\ a_{k_1} : (e == k_2\ ?\ a_{k_2} : \ldots (e == k_n\ ?\ a_{k_n} : \mathfrak{u}) \ldots))$$

for each array a, with $R(a) = \{k_1, \ldots, k_n\}$.
- Moreover, every assignment $a[e_1] = e_2;$ is replaced by

$$a_{k_1}, \ldots, a_{k_n} = (e_1 == k_1\ ?\ e_2 : a_{k_1}), \ldots, (e_1 == k_n\ ?\ e_2 : a_{k_n});$$

where $a_{k_1}, \ldots, a_{k_n}$ are new variables of numeric type. (If $n = 0$, the assignment above reduces to a skip ($;$) statement.).

Linear Programs with Arrays

Original Program

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

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

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

Linear Programs with Arrays

Original Program

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

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

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

Experimental results

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

Conclusions

Future research directions:

- Extend the approach to pointers
- Use of widening operator
- Use of more efficient data-structures
- Find sufficient conditions for termination

The EUREKA project

www.ai.dist.unige.it/eureka

Conclusions

Questions?

A. Armando and C. Castellini and J. Mantovani,
Software Model Checking Using Linear Constraints,
Proc. of ICFEM, Seattle, 2004,
LNCS 3308, Springer.

Rajeev Alur and David L. Dill,
A theory of timed automata,
Theoretical Computer Science, vol. 126, nr. 2,
Elsevier Science Publishers Ltd., 1994.

J. Esparza,
Decidability of Model-checking for Infinite-state Concurrent Systems,
Acta Informatica, vol. 34, 1997.