Rapid Overview of Computer Aided Verification

Moves - April 20, 2007

JF Raskin
U.L.B.
• What do we do?
• Why do we do it?
• How do we do it?
We catch bugs.
An exception 06 has occurred at 0028:C11B3ADC in VxD DiskTSD03 + 00001660. This was called from 0028:C11B40C8 in VxD voltrack04 + 00000000. It may be possible to continue normally.

* Press any key to attempt to continue.
* Press CTRL+ALT+RESET to restart your computer. You will lose any unsaved information in all applications.

Press any key to continue
June 4, 1996

The European Ariane 5 rocket explodes 40 s into its maiden flight due to a software bug.
Boeing could not assemble and integrate the fly-by-wire system until it solved problems with the databus and the flight management software. Solving these problems took more than a year longer than Boeing anticipated. In April, 1995, the FAA certified the 777 as safe.

Total development cost: $3 billion
Software integration and validation cost: one third of total
How to master complexity in science?

Model

System

Abstract

Predict

Calculate

Applied Mathematics

Bridge Aircraft etc.

Build & test

Complexity Management in Engineering
Model Applied Mathematics System Calculate Predict Abstract Build & test
How to master complexity in science?

Model

System

Applied Mathematics

Abstract

Predict

Calculate

A Program

Software?

kbfilter.c
12,000 lines of code
A Program

kbfilter.c
12,000 lines of code
A Program

An execution

kbfilter.c
12,000 lines of code
A Program

An execution
No continuous maths

kbfilter.c
12,000 lines of code
We need the proper maths

Engineering
- Differential Equations
- Linear Algebra
- Probability Theory

Computer Science
- Mathematical Logic
- Discrete Structures
- Automata Theory
We need the proper maths

Engineering
- Differential Equations
- Linear Algebra
- Probability Theory

Computer Science
- Mathematical Logic
- Discrete Structures
- Automata Theory

Sometimes, we need both...
The impossible dream

Program Property Verifier
Yes / No

Conclusion: Verifier cannot exist!

Program X
Verifier
terminates
if Yes then loop forever;

[Turing 1936]

[15]
We work with models
Methodology

- **Models** capture the relevant aspects of the system formally;
- **Specifications** in logical formalisms express the important properties of the system;
- **Model-checking** algorithms verify that the model of the system respects the specification.
We need a variety of models and their verification algorithms

- Finite state automata (on infinite words);
- Timed (and hybrid) automata;
- Abstract models of program semantics.
Finite state automata
Reactive systems

- Their semantics can be modeled as infinite sequences of events;
- Sets of infinite sequences of events = infinite word languages;
- Reactive systems can be modeled with finite state automata over infinite words, Nondeterministic Büchi Word Automata (NBW);
- Verification problems reduce to automata-theoretic problems.
Set of components

We model them as a product of automata

Correctness: Mutual exclusion

Translate to LTL

Semantics = language = sets of words

G¬(CS₁ ∧ CS₂)
\[ G \vdash (CS_1 \land CS_2) \]
$(CS_1 \land CS_2) \quad \neg (CS_1 \land CS_2)$
every LTL formula can be translated into NBW

[Vardi, Wolper]
Is the language of empty?
We have reduced a **model-checking** problem to a **automata-theoretic** problem

Is the language of empty?
Global state space
Global state space

To test the system, we can run on paths;
To test the system, we can run on paths; if we want to verify the system, we have to run on all paths!
Global state space

To test the system, we can run on paths;

Practical problem: state explosion! The number of state of the global state space can be exponential in the number of components!

If we want to verify the system, we have to run on all paths!
State explosion problem!

Cannot build
Cannot explore \}

the full state space

⇒ Solutions to tackle the state explosion problem
Solutions to tackle the state explosion problem

1. Compact representation of the state space
   - Efficient data structure
     . to represent sets of global states
     . to efficiently compute operations on sets;

2. Avoid to consider the complete graph
   . partial order reduction

3. Work on a a smaller / more abstract state space
   . reduction modulo some equivalence: symmetry reduction
Example of efficient data structure: the ROBDD

Reachable-States\((I, T, \bar{x}, \bar{x}' )\)

1: \( R \leftarrow [0] \)
2: repeat
3: \( R' \leftarrow R \)
4: \( R \leftarrow I \lor (\exists \bar{x}. T \land R)[\bar{x}/\bar{x}'] \)
5: until \( R' = R \)
6: return \( R \)

State = string of bits \((x_1, x_2, \ldots)\)

Symbolic manipulation of sets of states.

Direct operations on the data structure
Example of efficient **data structure**: the ROBDD

[Bryant, McMillan]

Reachable-States\((I, T, \bar{x}, \bar{x'}\)\)

\begin{align*}
1: & \quad R \leftarrow \begin{bmatrix} 0 \end{bmatrix} \\
2: & \quad \text{repeat} \\
3: & \quad R' \leftarrow R \\
4: & \quad R \leftarrow I \lor (\exists \bar{x}. T \land R)[\bar{x}/\bar{x'}] \\
5: & \quad \text{until } R' = R \\
6: & \quad \text{return } R
\end{align*}

state = string of bits \((x_1, x_2, \ldots)\)

**Symbolic** manipulation of sets of states.

**Direct operations** on the data structure.

See also works for infinite sets
[Boigelot et al, Raskin et al]
Example to avoid to analyse the complete data structure

- partial order reduction [Godefroid, Wolper, Valmari, etc]

The State Explosion Problem
Allowing all possible orderings is a potential cause of the state explosion problem.
To resolve this, the specification does not distinguish between these sequences, it is beneficial to consider only one with states.

Principle: the analysis of a subset of paths is sufficient.
The other paths being equivalent for the property to check.
Work on a smaller state space
  . reduction modulo some equivalence: symmetry reduction
  . abstraction

[Emerson et al]

Identify equivalent states, and work on a reduce state space

See also,
- Counting abstractions [Delzanno, Geeraerts, Ganty, Raskin, Van Begin]
- Symmetry reduction in B [Leuschel, Massart]
An small example
Train model

Train

far

near

app!

exit!

$\epsilon$

past
Gate model

Gate

up -> open

raise?  lower?

down -> closed

raise?  lower?
Controller model

Controller model diagram with states:
- idle
- app?
- lower!
- exit?
- raise!

Diagram shows transitions between states.
$G (\text{ past } \rightarrow \text{ closed })$
G ( \text{past} \rightarrow \text{closed} )
$G ( \text{past} \rightarrow \text{closed} )$
$G \left( \text{past } \rightarrow \text{ closed} \right)$
G ( past → closed )
G ( past → closed )
G ( past → closed )
G ( past → closed )

Falsified!

48
Timed automata
Crucial real-time information of the behavior is missing!
Models = set of timed traces

A timed trace is an infinite sequence of the form

\[ s_0 \rightarrow (a_0, t_0) \rightarrow s_1 \rightarrow (a_1, t_1) \rightarrow s_2 \rightarrow (a_2, t_2) \rightarrow \ldots \rightarrow s_n \rightarrow (a_n, t_n) \rightarrow \ldots \]

where:
- each \( s_i \) is a subset of the set of propositions \( P \);
- each \( a_i \) is an element of \( \Sigma \), the set of events;
- each \( t_i \) is a positive real number, and we verify:
  (1) for each \( i \geq 0 : t_i \leq t_{i+1} \) (monotonicity) and
  (2) for any positive real \( r \), there exists a position \( i \geq 0 \) such that \( t_i \geq r \) (non-zenoness).
Timed Automata

[Alur, Dill, Henzinger, Larsen, Raskin, etc]

• Timed Automata = Finite State Machines + Clocks;

• Clocks = \textit{continuous} variables that count time;

• Operations on clocks = resetting and comparison to constants.
Here is a diagram titled "TA for the train." The diagram shows states and transitions with the following labels:

- **far**
  - Transition with label **x:=0**
  - Edge labeled **app!**

- **past**
  - Transition with label **exit!**

- **near**
  - Invariant: **x ≤ 30**
  - Edge labeled **x ≥ 20**

- **Guard**
  - Transition labeled **exit**

- **Invariants**
  - Transition labeled **app**

The diagram also includes a clock labeled **x: clock**.
G ( past → closed )

x:=0
app!

x\leq 30

x:=0
app!

ε

x \geq 20

y:=0
app?

z:=0
raise?

z:=0
lower?

z \leq 9

z:=0
raise?

z \leq 9

z:=0
lower?

z \leq 9

x=y=z=0, time=0
G ( past $\rightarrow$ closed )

$y := 0$

$x := 0$

$x \leq 30$

$x \geq 20$

$z := 0$

$z \leq 9$

$y \leq 1$

$y \leq 1$

$z := 0$

$z \leq 9$

$x = y = z = 1, \text{ time } = 1$
G ( past → closed )

x=0, y=0, z=1, time=1
\( G \) (past \( \rightarrow \) closed)

\[ x := 0 \]
\[ x \leq 30 \]
\[ x \geq 20 \]
\[ \epsilon \]

\[ y := 0 \]
\[ y \leq 1 \]

\[ z := 0 \]
\[ z \leq 9 \]

\[ x = y = 0.5, z = 1.5, \text{ time} = 1.5 \]
G ( past → closed )

x=y=0.5, z=0, time=1.5
G ( \text{past} \rightarrow \text{closed} )

x=\text{y}=9.5, \text{z}=9, \text{time}=10.5
G (past → closed)

x = y = 34.5, z = 9, time = 35.5
TA are infinite state systems!
TA are infinite state systems!

We need a symbolic approach for the verification!
Regions
Regions

\[
\begin{array}{c|c|c|c|c|c}
\hline
\text{y} & 3 & 2 & 1 & 0 & \text{x} \\
\hline
0 & & & & & \\
1 & & & & & \\
2 & & & & & \\
3 & & & & & \\
\hline
\end{array}
\]

Diagram showing the regions with a shaded triangle between x=1 and y=2.
Time passing
Clock resetting
Program verification
What a program *really* is...

Example ( ) {
  1: do{
      lock();
      old = new;
      q = q->next;
      if (q != NULL){
        q->data = new;
        unlock();
        new ++;
      }
    } while(new != old);
  unlock ();
  return;}

What a program *really* is...

**State**

```c
Example ( ) {
  1:  do{
      lock();
      old = new;
      q = q->next;
    2:    if (q != NULL){
      3:     q->data = new;
      unlock();
      new ++;
    5:  } while(new != old);
    5:  unlock ();
    return;
```
What a program really is...

State Transition

Example ( ) {
  1: do{
      lock();
      old = new;
      q = q->next;
  2:    if (q != NULL){
      3:     q->data = new;
      unlock();
            new ++;
      }
  4: } while(new != old);
  5: unlock ();
    return;}

$pc \mapsto 3$
$lock \mapsto \bullet$
$old \mapsto 5$
$new \mapsto 5$
$q \mapsto 0x133a$
What a program really is...

State Transition

Example () {
1: do{
   lock();
   old = new;
   q = q->next;
2:   if (q != NULL){
3:    q->data = new;
    unlock();
     new ++;
   }
4: } while(new != old);
5: unlock();
    return;
}
What a program really is...

Example () {
  1: do{
      lock();
      old = new;
      q = q->next;
  2:    if (q != NULL){
      3:     q->data = new;
          unlock();
            new ++;
      4: } ... }
  5: } while(new != old);
  6: unlock ();
    return;
The Safety Verification Problem
The Safety Verification Problem
The Safety Verification Problem

Is there a path from an initial to an error state?
The Safety Verification Problem

Is there a path from an initial to an error state?
The Safety Verification Problem

Is there a path from an initial to an error state?

Problem: Infinite state graph
The Safety Verification Problem

Is there a path from an initial to an error state?

Problem: Infinite state graph

Solution: Set of states = logical formula
Idea 1: Predicate Abstraction
Idea 1: Predicate Abstraction

- **Predicates** on program state:
Idea 1: Predicate Abstraction

- **Predicates** on program state: 
  
  *lock*
Idea 1: Predicate Abstraction

- **Predicates** on program state:
  - `lock`
  - `old = new`
Idea 1: Predicate Abstraction

- **Predicates** on program state:
  - `lock`
  - `old = new`

- States satisfying **same** predicates
Idea 1: Predicate Abstraction

- **Predicates** on program state: 
  \( lock \)
  \( old = new \)

- States satisfying **same** predicates are **equivalent**
Idea 1: Predicate Abstraction

- **Predicates** on program state: $lock$
  $old = new$

- States satisfying **same** predicates are **equivalent**
  - Merged into one abstract state
Idea 1: Predicate Abstraction

• **Predicates** on program state: 
  - `lock`
  - `old = new`

• States satisfying **same** predicates are **equivalent**
  - Merged into one abstract state

• #abstract states is **finite**
Idea 1: Predicate Abstraction

- **Predicates** on program state: 
  
  \[
  \text{lock} \\
  \text{old} = \text{new}
  \]

- States satisfying **same** predicates are **equivalent**
  - Merged into one **abstract state**

- **#abstract states is finite**
Abstract States and Transitions

State

\[ \text{new} \to \text{unlock(); new++; } \]

3: unlock();
new++;
4: ...
Abstract States and Transitions

State

3: unlock();
    new++;
4: } ...

pc  \mapsto  3
lock \mapsto \bullet
old  \mapsto  5
new  \mapsto  5
q    \mapsto  0x133a

pc  \mapsto  4
lock \mapsto \circ
old  \mapsto  5
new  \mapsto  6
q    \mapsto  0x133a
Abstract States and Transitions

State

3: unlock();
    new++;
4:} ...

lock
old=new
Abstract States and Transitions

```
3: unlock();
   new++;
4: }
```

State transitions:
- 3: unlock(); new++
- 4: }

Transition rules:
- pc
- lock
- old
- new
- q

Initial state:
- pc
- lock
- old
- new
- q

Final state:
- pc
- lock
- old
- new
- q

Transition:
- lock
- old=new
- ¬lock
- ¬old=new
Theorem Prover

Abstract States and Transitions

State

pc \rightarrow 3
lock \rightarrow q
old \rightarrow 5
new \rightarrow 5
q \rightarrow 0x133a

3: unlock();
new++;
4: }

¬lock
old=new

pc \rightarrow 4
lock \rightarrow q
old \rightarrow 5
new \rightarrow 6
q \rightarrow 0x133a

Theorem Prover

lock
old=new

¬lock
¬old=new
Abstraction

State

3: unlock();
   new++;
4: }

Existential Lifting

Theorem Prover

lock
old=new
¬lock
¬old=new
Abstraction

State

3: unlock();
    new++;
4: }

pc
lock
old
new
q
⇒ 3
⇒ 5
⇒ 5
⇒ 0x133a

pc
lock
old
new
q
⇒ 4
⇒ 6
⇒ 0x133a

Existential Lifting

Theorem Prover

¬lock
¬old=new
Abstraction

State

3: unlock();
new++;}

4: }

pc
→ 3
lock
→ ●
old
→ 5
new
→ 5
q
→ 0x133a

pc
→ 4
lock
→ ○
old
→ 5
new
→ 6
q
→ 0x133a

lock
old=new

: lock
: old=new
Abstraction

State

3: unlock();
   new++;
4: } ...

pc  ↦ 3
lock  ↦ …
old  ↦ 5
new  ↦ 5
q  ↦ 0x133a

lock
old=new

pc  ↦ 4
lock  ↦ …
old  ↦ 5
new  ↦ 6
q  ↦ 0x133a
Analyze Abstraction
Analyze Abstraction
Analyze Abstraction

Analyze finite graph

Over Approximate:
Analyze Abstraction

Analyze finite graph

Over Approximate:
Safe $\Rightarrow$ System Safe
Analyze Abstraction

Analyze finite graph

**Over** Approximate:
Safe $\Rightarrow$ System Safe

No **false negatives**
Analyze Abstraction

Analyze finite graph

Over Approximate: Safe $\Rightarrow$ System Safe

No false negatives

Problem
Analyze Abstraction

Analyze finite graph

Over Approximate:
Safe $\Rightarrow$ System Safe

No false negatives

Problem
Spurious counterexamples
Analyze Abstraction

Analyze finite graph

Over Approximate:
Safe $\Rightarrow$ System Safe

No false negatives

Problem
Spurious counterexamples
Idea 2: Counterex.-Guided Refinement

Solution
Use spurious counterexamples to refine abstraction!
Idea 2: Counterex.-Guided Refinement

Solution
Use spurious counterexamples to refine abstraction
Idea 2: Counterex.-Guided Refinement

Solution
Use spurious counterexamples to refine abstraction
Idea 2: Counterex.-Guided Refinement

Solution
Use spurious counterexamples to refine abstraction
Idea 2: Counterex.-Guided Refinement

Solution

Use spurious counterexamples to refine abstraction

Imprecision due to merge
Idea 2: Counterex.-Guided Refinement

Solution
Use spurious counterexamples to refine abstraction

1. Add predicates to distinguish states across cut
2. Build refined abstraction
Iterative Abstraction-Refinement

Solution

Use spurious counterexamples to refine abstraction

[Kurshan et al 93] [Clarke et al 00]
[Ball-Rajamani 01]
Iterative Abstraction-Refinement

Solution
Use spurious counterexamples to refine abstraction

[Kurshan et al 93] [Clarke et al 00]
[Ball-Rajamani 01]
Iterative Abstraction-Refinement

Solution
Use spurious counterexamples to refine abstraction

[Kurshan et al 93] [Clarke et al 00]
[Ball-Rajamani 01]
Iterative Abstraction-Refinement

Solution
Use spurious counterexamples to refine abstraction

1. Add predicates to distinguish

[Kurshan et al 93] [Clarke et al 00]
[Ball-Rajamani 01]
Iterative Abstraction-Refinement

Solution
Use spurious counterexamples to refine abstraction

1. Add predicates to distinguish states across cut

[Kurshan et al 93] [Clarke et al 00]
[Ball-Rajamani 01]
Iterative Abstraction-Refinement

Solution

Use spurious counterexamples to refine abstraction

1. Add predicates to distinguish states across cut
2. Build refined abstraction

[Kurshan et al 93] [Clarke et al 00]
[Ball-Rajamani 01]
Iterative Abstraction-Refinement

Solution
Use spurious counterexamples to refine abstraction

1. Add predicates to distinguish states across cut
2. Build refined abstraction - eliminates counterexample

[Kurshan et al 93] [Clarke et al 00]
[Ball-Rajamani 01]
Iterative Abstraction-Refinement

Solution
Use spurious **counterexamples** to **refine** abstraction

1. Add predicates to distinguish states across **cut**
2. Build **refined** abstraction
   - eliminates counterexample
3. **Repeat** search

[Kurshan et al 93] [Clarke et al 00]
[Ball-Rajamani 01]
Iterative Abstraction-Refinement

Solution
Use spurious counterexamples to refine abstraction

1. Add predicates to distinguish states across cut
2. Build refined abstraction - eliminates counterexample
3. Repeat search Till real counterexample

[Kurshan et al 93] [Clarke et al 00] [Ball-Rajamani 01]
Iterative Abstraction-Refinement

Solution

Use spurious countereXamples to refine abstraction

1. Add predicates to distinguish states across cut
2. Build refined abstraction - eliminates countereXample
3. Repeat search
   Till real countereXample or system proved safe

[Kurshan et al 93] [Clarke et al 00]
[Ball-Rajamani 01]
Iterative Abstraction-Refinement

Recent work by Cousot, Ganty and Raskin: beyond CEGAR!

[Kurshan et al 93] [Clarke et al 00] [Ball-Rajamani 01]
Iterative Abstraction-Refinement

Recent work by Cousot, Ganty and Raskin: beyond CEGAR!

[Kurshan et al 93] [Clarke et al 00]
[Ball-Rajamani 01]
Iterative Abstraction-Refinement

Recent work by Cousot, Ganty and Raskin: beyond CEGAR!

[Kurshan et al 93] [Clarke et al 00] [Ball-Rajamani 01]
Research questions

• Efficient verification algorithms: fight the state explosion problem;

• Adequate models: we need infinite state models and efficient exploration algorithms;

• We need robust models: timed automata are fragile;

• We need better abstraction refinements algorithms;

• Beyond verification: how to synthesize correct designs from specifications? ➔ Synthesis