Introduction to
Bounded Model Checking

with slides from P.Gastin, the INFO-F-412 lecture, and from the CBMC Website
Need for formal verifications methods

Critical systems

- Transport
- Energy
- Medicine
- Communication
- Finance
- Embedded systems
- ...

Disastrous software bugs

Mariner 1 probe, 1962
See http://en.wikipedia.org/wiki/Mariner_1

- Destroyed 293 seconds after launch
- Missing hyphen in the data or program? No!
- Overbar missing in the mathematical specification:
  $\ddot{R}_n$: $n$th smoothed value of the time derivative of a radius.
Without the smoothing function indicated by the bar, the program treated normal minor variations of velocity as if they were serious, causing spurious corrections that sent the rocket off course.
Disastrous software bugs

Ariane 5 flight 501, 1996

See http://en.wikipedia.org/wiki/Ariane_5_Flight_501

- Destroyed 37 seconds after launch (cost: 370 millions dollars).
- Data conversion from a 64-bit floating point to 16-bit signed integer value caused a hardware exception (arithmetic overflow).
- Efficiency considerations had led to the disabling of the software handler (in Ada code) for this error trap.
- The fault occurred in the inertial reference system of Ariane 5. The software from Ariane 4 was re-used for Ariane 5 without re-testing.
- On the basis of those calculations the main computer commanded the booster nozzles, and somewhat later the main engine nozzle also, to make a large correction for an attitude deviation that had not occurred.
- The error occurred in a realignment function which was not useful for Ariane 5.
Disastrous software bugs

Spirit Rover (Mars Exploration), 2004


- Ceased communicating on January 21.
- Flash memory management anomaly: too many files on the file system
- Resumed to working condition on February 6.
Disastrous software bugs

Other well-known bugs


Model Checking

- Purpose 1: automatically finding software or hardware bugs.
- Purpose 2: prove correctness of abstract models.
- Should be applied during design.
- Real systems can be analysed with abstractions.

Prix Turing 2007.
Model Checking: The Basic Algorithm

- **requirements**
  - formalize
  - specification

- **system**
  - model & formalize
  - model

---

- **model check**
  - (run out of resources)
  - counter-example

- **fix/debug**
  - simulate
    - get error locations
A Mutual Exclusion Algorithm

- problem setting: find an algorithm such that
  - a group of (two) concurrent processes share a common resource
  - no more than one process has access at the same time
  - access to the resource is modeled by a critical section

- a first simplistic example:
  assert two processes $P_0, P_1$ given as

```plaintext
1 # non-critical section
2 while (other process critical) :
   3      wait()
4 # critical section
5 # return to non-critical
```
focus mainly “models” that are Kripke structures $\mathcal{M} = \langle S, R, L \rangle$
(set of state $S$, transitions $R \subseteq S \times S$, labeling $L : S \rightarrow 2^{AP}$, $AP$
is finite set of atomic predicates, no default initial states)

model mutex algorithm

- 2 processes $P_0$ and $P_1$ as before
- generate $\mathcal{M} = \langle S, R, L \rangle$ by product construction
- write (global) states as $(s_0s_1) \in S$, i.e., $P_0$ in $s_0$ and $P_1$ in $s_1$
Safety: The protocol allows only one process to be in its critical section at any time.
Safety: The protocol allows only one process to be in its critical section at any time.

Liveness: Whenever any process wants to enter its critical section, it will eventually be permitted to do so.
Mutex: Specifying Properties

**Safety:** The protocol allows only one process to be in its critical section at any time.

**Liveness:** Whenever any process wants to enter its critical section, it will eventually be permitted to do so.

**Non-Blocking:**
A process can always request to enter its critical section.
Safety: The protocol allows only one process to be in its critical section at any time.

Liveness: Whenever any process wants to enter its critical section, it will eventually be permitted to do so.

Non-Blocking: A process can always request to enter its critical section.

No Strict Sequencing: Processes need not enter their critical section in a strict sequence.
can **simplify** our properties as $P_0$ and $P_1$ are “identical”:

**Safety:** The protocol allows only process to be in its critical section at any time.

**Liveness:** Whenever $P_0$ wants to enter its critical section, it will eventually be permitted to do so.

**Non-Blocking:**

$P_0$ can always request to enter its critical section.

**No Strict Sequencing:**

$P_0$ needs not enter its critical section in a strict sequence with $P_1$. 
Temporal Logics

How to formalize these requirements?

...such that we have a rigorous semantics?
...such that we can verify that they hold?
...such that a tool can help us checking it?

▷ we need to take a look at temporal logics...
(C)omputational (T)ree (L)ogic was proposed 1981 by E.M. Clarke & E.A. Emmerson

**Definition:**

Computational tree logic has the following syntax in BNF:

\[
\varphi ::= T \mid p \mid (\neg \varphi) \mid (\varphi \lor \varphi) \mid (EX \varphi) \mid (AX \varphi) \mid E(\varphi U \varphi) \mid A(\varphi U \varphi)
\]

- **atoms**
- **Boolean connectives**
- **temporal operators**

- note the new path quantifiers:
  - there “(E)xists a path” and “for (A)ll paths”
- **binding:** \(\{\neg, AX, EX\} < \{EU, AU\} < \{\land, \lor\} < \{\rightarrow, \leftrightarrow\}\)
- formal **semantics** is based on **infinite trees/unfoldings**
CTL: Semantic Intuition

CTL semantics by example

- EXφ, AXφ
CTL: Semantic Intuition

CTL semantics by example

- EXφ, AXφ
- E(φUφ), A(φUφ)

\[ \varphi \]

[Tree Diagram]

\[ \text{CTL semantics by example} \]

\[ \text{• EX}_\varphi, \text{AX}_\varphi \]

\[ \text{• E}(\varphi U\phi), \text{A}(\varphi U\phi) \]
CTL: Semantic Intuition

CTL semantics by example

- $\text{EX} \varphi$, $\text{AX} \varphi$
- $\text{E}(\varphi U \phi)$, $\text{A}(\varphi U \phi)$

introduce well known quantifiers

- $\text{AF} \varphi \equiv \text{A}(\top U \varphi)$
- $\text{EG} \varphi \equiv \text{E}(\bot U \varphi)$
- analogously: $\text{EF} \varphi$ and $\text{AG} \varphi$

draw the “tree diagrams” for $\text{EF} \varphi$ and $\text{AG} \varphi$
CTL: Back to Mutex Example

Safety: $\varphi_{\text{safety}} \equiv AG\neg(c_0 \land c_1)$

Liveness: $\varphi_{\text{liveness}} \equiv AG(t_0 \rightarrow AF c_0)$

Non-Blocking: $\varphi_{\text{nblock}} \equiv AG(n_0 \rightarrow EX t_0)$

No Strict Sequencing: $\varphi_{\text{nss}} \equiv EF(c_0 \land E(c_0 U(\neg c_0 \land E(\neg c_1 U c_0))))$
Model Checking: Mutex

\[ \varphi_{\text{safety}} \equiv AG \neg (c_1 \land c_2) \]
Model Checking: Mutex

Safety

$$\varphi_{\text{safety}} \equiv AG\neg(c_1 \land c_2)$$

obviously:

$$c_0 c_1$$ is not reachable
Model Checking: Mutex

\begin{itemize}
\item[⇒] Safety
\[ \varphi_{\text{safety}} \equiv AG\neg(c_1 \land c_2) \]
\checkmark
\item[⇒] Liveness
\[ \varphi_{\text{liveness}} \equiv AG(t_1 \rightarrow AFc_1) \]
\end{itemize}

obviously:

- \(c_0 \circ c_1\) is not reachable
Model Checking: Mutex

Safety
\[ \varphi_{\text{safety}} \equiv AG\neg(c_1 \land c_2) \]

obviously:
\( c_0c_1 \) is not reachable

Liveness
\[ \varphi_{\text{liveness}} \equiv AG(t_1 \rightarrow AFc_1) \]

counterexample:
\[ t_0n_1 \rightarrow t_0t_1 \rightarrow t_0c_1 \rightarrow t_0n_1 \rightarrow \ldots \]
Temporal Logic Model Checking

How to algorithmically model check a (CTL) formula

... on large finite models (e.g., with $10^{10^2}$ states) ?
... on infinite models ?
... efficiently ?
... returning a counterexample if it does not hold ?
Bounded Model Checking
Bounded Reachability

Given a model $M = \langle S, R, L \rangle$ over $AP$

we define the following predicate on states

- $Reach(s, s')$ iff $R(s, s')$
Bounded Reachability

- given a model $\mathcal{M} = \langle S, R, L \rangle$ over $AP$

- we define the following predicate on states
  
  - $Reach(s, s')$ iff $R(s, s')$

- now $\mathcal{M}^k = \bigwedge_{i=0}^{k-1} Reach(s_i, s_{i+1})$

- if $s_0, s_1, s_2, \ldots, s_k \in \mathcal{M}^k$ then...
Bounded Reachability

given a model $\mathcal{M} = \langle S, R, L \rangle$ over $AP$

we define the following predicate on states

- $Reach(s, s')$ iff $R(s, s')$

now $[\mathcal{M}]^k = \bigwedge_{i=0}^{k-1} Reach(s_i, s_{i+1})$

if $s_0, s_1, s_2, \ldots, s_k \in [\mathcal{M}]^k$ then...

what do you now about the SAT problem and SAT-solvers?
(hint: look at your notes from the INFO-F302 lecture)
Bounded Model Checking Algorithm

\( M, \varphi, k = 0 \)

\( k++ \)

\( \varphi \) satisfiable in \([M]^k\) ?

SAT \( \rightarrow M \models \varphi \)

UNSAT \( \rightarrow M \not\models \varphi \)

test \( k < ct \)

\( ct \) is an apriori given computation threshold

n \( \rightarrow y \)

M, \varphi, k = 0
Preliminaries

- We aim at the analysis of programs given in a commodity programming language such as C, C++, or Java.

- As the first step, we transform the program into a control flow graph (CFG).

Diagram:

1. C/C++ Source
2. Parse
3. Parse tree
4. CFG

Example: SHS

```c
if ( (0 <= t) && (t <= 79) )
    switch ( t / 20 )
    {
        case 0:
            TEMP2 = ( (B AND C) OR (~B AND D) );
            TEMP3 = ( K_1 );
            break;

        case 1:
            TEMP2 = ( (B XOR C XOR D) );
            TEMP3 = ( K_2 );
            break;

        case 2:
            TEMP2 = ( (B AND C) OR (B AND D) OR (C AND D) );
            TEMP3 = ( K_3 );
            break;

        case 3:
            TEMP2 = ( B XOR C XOR D );
            TEMP3 = ( K_4 );
            break;

        default:
            assert(0);
    }
```
Example: SHS

```c
if ( (0 <= t) && (t <= 79) )
switch ( t / 20 )
{
    case 0:
        TEMP2 = ( (B AND C) OR (~B AND D) );
        TEMP3 = ( K_1 );
        break;
    case 1:
        TEMP2 = ( (B XOR C XOR D) );
        TEMP3 = ( K_2 );
        break;
    case 2:
        TEMP2 = ( (B AND C) OR (B AND D) OR (C AND D) );
        TEMP3 = ( K_3 );
        break;
    case 3:
        TEMP2 = ( B XOR C XOR D );
        TEMP3 = ( K_4 );
        break;
    default:
        assert(0);
}
```
Goal: check properties of the form $\text{AG}p$, say assertions.

Idea: follow paths through the CFG to an assertion, and build a formula that corresponds to the path.
Example

if
  0 ≤ t ≤ 79
switch
  case-0
    t/20 ≠ 0
  case-1
    t/20 ≠ 1
  case-2
    t/20 ≠ 2
  case-3
    t/20 ≠ 3
  default

Example

if

0 ≤ t ≤ 79

switch

case-0

t/20 ≠ 0

case-1

t/20 ≠ 1

case-2

t/20 ≠ 2

case-3

t/20 ≠ 3

default
Example

if

0 ≤ t ≤ 79

switch

case-0

t/20 ≠ 0

case-1

t/20 ≠ 1

case-2

t/20 ≠ 2

case-3

t/20 ≠ 3

default

0 ≤ t ≤ 79
∧ t/20 ≠ 0
∧ t/20 = 1
∧ TEMP2 = B ⊕ C ⊕ D
∧ TEMP3 = K_2
Example

We pass

\[ 0 \leq t \leq 79 \]
\[ \land \ t/20 \neq 0 \]
\[ \land \ t/20 = 1 \]
\[ \land \ TEMP2 = B \oplus C \oplus D \]
\[ \land \ TEMP3 = K_2 \]

to a decision procedure, and obtain a satisfying assignment, say:

\[ t \mapsto 21, \ B \mapsto 0, \ C \mapsto 0, \ D \mapsto 0, \ K_2 \mapsto 10, \]
\[ TEMP2 \mapsto 0, \ TEMP3 \mapsto 10 \]

✓ It provides the values of any inputs on the path.
Let’s Look at Another Path

if

0 ≤ t ≤ 79

switch

case-0

t/20 ≠ 0

case-1

t/20 ≠ 1

case-2

t/20 ≠ 2

case-3

t/20 ≠ 3

default

That is UNSAT, so the assertion is unreachable.

Let’s Look at Another Path

```
if
  0 ≤ t ≤ 79
switch
  case-0
    t/20 ≠ 0
  case-1
    t/20 ≠ 1
  case-2
    t/20 ≠ 2
  case-3
    t/20 ≠ 3
default
```

That is UNSAT, so the assertion is unreachable.

Let's Look at Another Path

if

0 ≤ t ≤ 79

switch

case-0

t/20 ≠ 0

case-1

t/20 ≠ 1

case-2

t/20 ≠ 2

case-3

t/20 ≠ 3

default

0 ≤ t ≤ 79
∧ t/20 ≠ 0
∧ t/20 ≠ 1
∧ t/20 ≠ 2
∧ t/20 ≠ 3

That is UNSAT, so the assertion is unreachable.
Let’s Look at Another Path

if

0 ≤ t ≤ 79

switch

case-0

  t/20 ≠ 0

case-1

  t/20 ≠ 1

case-2

  t/20 ≠ 2

case-3

  t/20 ≠ 3

default

That is UNSAT, so the assertion is unreachable.
for (i = 0; i < 16; i++)
    x[i+32] = state[i] ^ block[i];

/* Encrypt block (18 rounds). */
t = 0;
for (i = 0; i < 18; i++) {
    for (j = 0; j < 48; j++)
        t = x[j] ^ PI_SUBST[t];
    t = (t + i) & 0xff;
}