Formal Verification by Model Checking

Natasha Sharygina
Carnegie Mellon University

Outline

Lecture 1: Overview of Model Checking
Lecture 2: Complexity Reduction Techniques
Lecture 3: Software Verification
Lecture 4: State/Event-based software model checking
Lecture 5: Component Substitutability
Lecture 6: Model Checking Practicum (Student Reports on the Lab exercises)

Today's Lecture

– Model checking basics (review from the last class)
– Problems with model checking
– Combating state space explosion

Temporal Logic Model Checking

[Clarke, Emerson 81][Quelle, Sifakis 82]

• Systems are modeled by finite state machines
• Properties are written in propositional temporal logic
• Verification procedure is an exhaustive search of the state space of the design
• Diagnostic counterexamples

Temporal Logic Model Checking

Model of Computation

Unwind State Graph to obtain Infinite Tree.

A trace is an infinite sequence of states.
What is “M”?

\[ M = (W, I, R, L, \Gamma) \]

- \( W \) - set of states
- \( I \subseteq W \) - set of initial states
- \( R \subseteq W \times W \) - set of arcs
- \( L \) - set of atomic propositions
- \( \Gamma : W \rightarrow 2^L \) - mapping from states to colors

Computation Tree Logics

**Examples:**

- **Safety** (mutual exclusion): no two processes can be at a critical section at the same time
- **Liveness** (absence of starvation): every request will be eventually granted

In a linear temporal logic (LTL), operators are provided for describing system behavior along a single computation path.

In a branching-time logic (CTL), the temporal operators quantify over the paths that are possible from a given state.

State Space Explosion

**Problem:** Size of the state graph can be exponential in size of the program (both in the number of the program variables and the number of program components)

\[ M = M_1 \parallel \ldots \parallel M_n \]

If each \( M_i \) has just 2 local states, potentially \( 2^n \) global states

**Research Directions:** State space reduction

Principal Approaches to State Space Reduction

**Compositional reasoning** (reasoning about parts of the system)

**Abstraction** (elimination of details irrelevant to verification of a property)

**Symbolic Verification** (BDDs represent state transition diagrams more efficiently)

**Partial Order Reduction** (reduction of number of states that must be enumerated)

**Domain specific reductions** (syntactic program transformations)

**Other** (symmetry, cone of influence reduction, ….)

Divide and Conquer

**Compositional reasoning** reduces reasoning about entire system to reasoning about individual parts

- Decompose the model: \( M = M_1 \parallel M_2 \)
- Partition global properties into local properties: \( P = P_1 \land P_2 \)
- Show that \( M_1 \models P_1 \) and \( M_2 \models P_2 \)

**Limitation:** It does not work in practice for properties that are constrained by behavior of the environment

Example

Process M1
\[
\begin{align*}
\text{var x: Boolean; } \\
\text{init x = 0; } \\
\text{x := y;}
\end{align*}
\]

Property P1: Always(x);

Process M2
\[
\begin{align*}
\text{var y: Boolean; } \\
\text{init y = 0; } \\
\text{y := 1;}
\end{align*}
\]

Property P2: Always(y);

M2 \models P2 can be checked but M1 \models P1 can not be checked since y is not constrained
Assume-Guarantee Compositional Reasoning

In the assume-guarantee paradigm: each component guarantees properties based on assumptions about other components via proof rules.

\[
M_1 \parallel P_2 \models P_1 \\
M_2 \parallel P_1 \models P_2 \\
M_1 \parallel M_2 \models P_1 \parallel P_2^* 
\]

*Circularity during verification of different blocks is broken by induction over time

[AbadiLamport95][AlurHenzinger96][HenzingerQadeerRajmani99][Kurschارد] [McMillan98][Stark85]

---

Scalability of Multi-Process Model Checking

Limitation: Properties do not give details about the verifiable processes

Approach: Add abstraction constraints

Abstraction constraints (cf. Henzinger 98) are added to the property specifications:

\[
P_1^{\text{abs}}, \ldots, P_n^{\text{abs}} \Rightarrow A 
\]

Abstraction constraints specification:

- Temporal logic formulae constraining the external variables of the verifiable processes

---

Abstractions in Model Checking

Data Abstraction (abstraction of details irrelevant to verification of a property)

Approach: \( M_{\text{abs}} \Rightarrow P \Rightarrow M_{\text{conc}} \Rightarrow P \)

To Prove: soundness and completeness

Systematic Construction of Abstractions (Predicate Abstraction) [SaidiGraf 97]

- Define an abstraction function as a predicate over concrete data
- Specify decision procedures to compute a set of abstraction predicates
- Demonstrate the soundness and completeness of the abstraction

Refinement-based abstraction [MSR SLAM Project][Clarke et. al. '00], [Saidi 05][Visser et. al 00]

---

Conservative Abstraction

- Every trace of M is a trace of A
  - A over-approximates what M can do (Preserves safety properties): \( A \models \phi \Rightarrow M \models \phi \)
  - Some traces of A may not be traces of M
  - May yield spurious counterexamples - (a, e)
  - Eliminated via abstraction refinement
  - Splitting some clusters in smaller ones
  - Refinement can be automated

Original Abstraction

Conservative Abstraction
Abstract Transitions:

• Existential abstraction
  conservative

– Typically done in a conservative manner

\[ \bar{I}(\bar{s}) : \iff \exists s : I(s) \]

\[ \bar{R}(\bar{s}, \bar{s}') : \iff \exists s, s' : R(s, s') \]

\[ \land h(s) = \bar{s} \land h(s') = \bar{s}' \]

⇒ Preserves safety properties

Under- vs. Overapproximation

• How to abstract the transitions?
  – Depends on the property we want to show
  – Typically done in a conservative manner

\[ \bar{I}(\bar{s}) : \iff \exists s : I(s) \]

\[ \bar{R}(\bar{s}, \bar{s}') : \iff \exists s, s' : R(s, s') \]

\[ \land h(s) = \bar{s} \land h(s') = \bar{s}' \]

⇒ Preserves safety properties

Predicate Abstraction

[Graf/Saidi 97]

• Idea: Only keep track of predicates on data
  \( p_1(s), \ldots, p_n(s) \)

• Abstraction function:
  \( h(s) = (p_1(s), p_2(s), \ldots, p_n(s)) \)

Concrete States:

Predicates:

\( p_1(s) = (s \cdot x > s \cdot y) \)

\( p_2(s) = (s \cdot y = 0) \)

Abstract transitions?
Predicate Abstraction for Software

- Let’s take existential abstraction seriously
- Basic idea: with n predicates, there are $2^n \times 2^n$ possible abstract transitions
- Let’s just check them!

Predicate Abstraction for Software

- A precise existential abstraction can be way too slow
  - Use an over-approximation instead
    - Fast to compute
    - But has additional transitions
  - E.g.: SLAM (FastAbs) - Microsoft
    - Predicate partitioning
Example for Predicate Abstraction

```c
int main() { 
    int i; 
    i=0; 
    while(!even(i)) 
        i++; 
} 
```

```c
void main() { 
    bool p1, p2; 
    p1=TRUE; 
    p2=TRUE; 
    while(p2) 
        { 
        p1=p1?FALSE:nondet(); 
        p2=!p2; 
        } 
} 
```

C program  Predicates  Boolean program

<table>
<thead>
<tr>
<th>Predicates</th>
<th>Boolean program</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>p1</code> and <code>p2</code></td>
<td><code>p1</code> and <code>p2</code></td>
</tr>
</tbody>
</table>

Fast Syntactic Abstraction

- Problem: Computing precise existential abstraction is exponential
- Even moderate-sized software needs >500 predicates
  - Won’t scale
- Syntactic abstraction is polynomial
  - Will scale!
- But: May result in additional spurious behavior and more non-determinism

Predicate Abstraction for Software

- How do we get the predicates?
  - Automatic abstraction refinement!
- [Clarke et al. '00]
- [Kurshan et al. '93]
- [Ball, Rajamani '00]

Abstraction Refinement Loop

<table>
<thead>
<tr>
<th>Abstraction refinement</th>
<th>Verification</th>
<th>Initial Abstraction</th>
<th>Model Checker</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulator</td>
<td>Property holds</td>
<td>No error or bug found</td>
<td></td>
</tr>
<tr>
<td>Spurious counterexample</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SLAM

- Microsoft blames most Windows crashes on third party device drivers
- The Windows device driver API is quite complicated
- Low level C code
- SLAM: Tool to automatically check device drivers for certain errors
- To be shipped with Device Driver Development Kit
- Full detail (and all the slides) available at
**Problem with Existing Tools**

- Existing tools (BLAST, SLAM, MAGIC) use a Theorem Prover like Simplify
- Theorem prover works on real or natural numbers, but C uses bit-vectors → false positives
- Most theorem provers support only few operators (+, -, <, ≤), no bitwise operators
- Idea: Use SAT solver to do bit-vector!

**Abstraction with SAT**

- Successfully used for abstraction of C programs (Clarke, Kroening, Sharygina, Yorav’03 – SAT-based predicate abstraction)
- There is now a version of SLAM that has it
  - Found previously unknown Windows bug
- Create a SAT instance which relates initial value of predicates, basic block, and the values of predicates after the execution of basic block
- SAT also used for simulation and refinement

**Abstraction Refinement Loop**

**Model Checkers for Boolean Programs**

- Explicit State
  - Zing
  - SPIN
- Symbolic
  - Moped
  - Bebop
  - SMV

**Refinement**

- Need to distinguish two sources of spurious behavior
  1. Too few predicates
  2. Laziness during abstraction
- SLAM:
  - First tries to find new predicates (NEWTON) using weakest preconditions
  - If this fails, second case is assumed. Transitions are refined (CONSTRAIN)
- Refine transitions using UNSAT cores (Clarke, Kroening, Sharygina, Yorav’05)
## Experimental Results

<table>
<thead>
<tr>
<th>Model checking result</th>
<th>SLAM/Zapato</th>
<th>SLAM/SAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property passes</td>
<td>243</td>
<td>264</td>
</tr>
<tr>
<td>Time threshold exceeded</td>
<td>39</td>
<td>17</td>
</tr>
<tr>
<td>Property violations found</td>
<td>17</td>
<td>19</td>
</tr>
<tr>
<td>Cases of abstraction-refinement failure</td>
<td>9</td>
<td>8</td>
</tr>
</tbody>
</table>

- Comparison of SLAM with Integer-based theorem prover against SAT-based SLAM
- 308 device drivers
- Timeout: 1200s