Overview of Model Checking

William Chan
wchan@cs.washington.edu

Department of Computer Science & Engineering
University of Washington
Outline

- Basics of model checking and temporal logic
- The *symbolic* variant
- Applications to specifications of reactive software
- Some lessons
Some properties expressible in temporal logics:

- Error states not reached (invariant).
- Eventually ack for each request (liveness).
- Can always restart the machine.
Computation Tree Logic (CTL)

- The usual Boolean operators: $\land$, $\lor$, $\neg$, etc., plus:
  
  $A$: for all paths, $E$: for some path,
  
  $G$: globally on the path, $F$: in a future state on the path, and some more.

- Examples:
  
  Error states not reached $AG\neg Err$
  
  Eventually ack for each request $AG(req \rightarrow AFack)$
  
  Can always restart the machine $AGEFrestart$

Many other temporal logics exist.

- Decade-long debate: expressiveness and complexity.
Why Temporal Logics?

What's wrong with partial correctness and termination?

- Not suitable for reactive systems.
- e.g., cannot express liveness and fairness.

The introduction of temporal logic is an award-winning idea. (Pnueli)

Why model checking?

- “Easy” for finite-state machines.
- Fancy graph traversals, linear in # states & transitions.
- You already know how to evaluate $\text{AG} \neg \text{Err}$. 
State Explosion

# states grow exponentially with # components.

Attacks:

- Abstraction and composition (Cospan)
- Symmetry reduction ($\text{Mur}_\varphi$)
- Partial-order reduction (Spin)
- **Symbolic search** (SMV, VIS):
  - Represent a set of states symbolically without enumerating the states individually.
Invariant Checking as Set Manipulations

Compute $Y_{i+1} = Pre(Y_i) \cup Y_i$. Check if $Y_n \cap Init = \emptyset$.

Backward breadth-first search

Fixed point

States that may reach an error state

Error states
Symbolic Search  [Burch et al. 90, Coudert et al. 89]

- Define Boolean state variables $X$.
  - e.g., define $x_{n-1}$, $x_{n-2}$, \ldots, $x_0$ for an $n$-bit integer.
- A set of states: a Boolean function $S(X)$.
  - e.g., $\neg x_0$ for the set of $n$-bit even integers.
- Set operations ($\cup$, $\cap$) becomes Boolean operations ($\lor$, $\land$).
- Transition relation: $R(X, X')$.
- Compute predecessors also using Boolean operations:
  \[ Pre(S) = \lambda X . \exists X' . S(X') \land R(X, X'). \]
Binary Decision Diagrams (BDDs) [Bryant 86]

BDD for odd parity

- Generalization of binary decision trees to DAGs.

- Restrictions:
  - Reduced: isomorphic subgraphs merged.
  - Ordered: every path conforms to a common variable order.

- Properties:
  - Canonical.
  - Operations poly-time in BDD size.
BDDs are Wild

BDD size not directly related to numbers of states or variables.

✓ Usually small. Some large state spaces ($10^{20}$) can be handled.
✓ Reduce the amount of manual abstraction needed.
✗ Sensitive to implementation details like variable order.
✗ Some well-known limitations (e.g., exponential size for $x > yz$).
✗ Few theoretical results known for general control systems. Performance can be unpredictable.
**Why Might BDDs Not Work Well for Software?**

Common view:

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td></td>
</tr>
<tr>
<td>Simple</td>
<td>Complex</td>
</tr>
<tr>
<td>States</td>
<td></td>
</tr>
<tr>
<td>Finite</td>
<td>Infinite</td>
</tr>
<tr>
<td>Concurrency</td>
<td></td>
</tr>
<tr>
<td>Synchronous</td>
<td>Asynchronous (aka Interleaving)</td>
</tr>
<tr>
<td>(aka Simultaneous)</td>
<td></td>
</tr>
<tr>
<td>Strategy</td>
<td></td>
</tr>
<tr>
<td>Use BDDs</td>
<td>Abstract and search explicitly</td>
</tr>
</tbody>
</table>

This may be true for software like multi-threaded programs, but . . . .
Consider Many Safety-Critical Software Specs

<table>
<thead>
<tr>
<th></th>
<th>Hardware</th>
<th>Spec</th>
<th>Multi-threaded Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>States</td>
<td>Finite</td>
<td>Finite (except numbers)</td>
<td>Possibly infinite</td>
</tr>
<tr>
<td>Data</td>
<td>Simple</td>
<td>Simple (except numbers)</td>
<td>Often complex</td>
</tr>
<tr>
<td>Concurrency</td>
<td>Synchronous</td>
<td>Synchronous</td>
<td>Asynchronous</td>
</tr>
</tbody>
</table>

Perhaps BDDs would work for such specs?
The Iterative Process

Spec (probably with abstraction) → Boolean encoding (in HDL) → Model Checker → Yes, or counterexample

Analyst → Property
TCAS II

- Traffic Alert and Collision Avoidance System
  - Warns pilots of traffic. (Does not control aircraft.)
  - Issues vertical resolution advisories (RAs)
    e.g., Climb, Descend, Increase-Climb, Do Not Descend > 500 ft/min.
  - Required on most commercial aircraft in USA.
  - One of the most complex systems on commercial aircraft.
- 400-page specification reverse-engineered from pseudo-code.
- Written in RSML [Leveson et al. 94], based on statecharts.
- Complexity in guarding conditions, not hierarchy or synchronization.
Analysis of TCAS II  [FSE 96, TSE 98]

• Around 200 Boolean variables, $10^{60}$ states.
• Used model checker SMV.  [McMillan 93]

• Domain-independent properties:
  – Transition consistency:
    $\text{AG} \neg(x \land c_1 \land c_2)$

• Domain-dependent properties:
  – Descent inhibition:
    $\text{AG}(\text{Alt} < 1000 \rightarrow \neg \text{Descend})$
  – Output agreement:
    $\text{AG} \neg(\text{GoalRate} \geq 0 \land \text{Descend})$
**EPD System**

Electrical Power Distribution system used on Boeing 777.

- Distribute power from power sources to power busses via circuit breakers.
- Tolerate failures in power sources and circuit breakers.
- Prototype specification for research purposes.
- Exercised extensively in simulation.
Failure Handling

- **Power Sources:** $LGen \quad RGen \quad \cdots$
- **Circuit Breakers:**
- **Power Busses:** $LMain \quad RMain \quad \cdots$

(Note: The diagram shows a comparison between two systems, with one marked as faulty.)
Analysis of EPD System

Joint work with David Jones and William Warner of Boeing. [ICSE 99]

- 90 Boolean variables, $10^{27}$ states.
- Fault tolerance
  - $\text{AG}(\text{NoFailures} \rightarrow (\text{LMain} \land \text{RMain} \land \text{LBackup} \land \text{RBackup}))$.
  - $\text{AG}(\text{AtMostOneFailure} \rightarrow (\text{LMain} \land \text{RMain}))$.
  - $\text{AG}(\text{AtMostTwoFailures} \rightarrow (\text{LBackup} \lor \text{RBackup}))$.
- Found modeling errors and logical flaws.

Not as complex as TCAS II, but initial analysis failed.
Issues/Lessons

• BDDs can't handle complicated arithmetic.
  – Abstract
  – Bound and discretize
    • Not sound, but it's ok.
  – Combine with a constraint solver.

• Domain expertise is essential.
  – For domain-specific properties
  – For abstraction
    • But, again, doesn't need to be sound and complete.
Issues/Lessons (cont'd)

- Can help understand interactions among components.
- Forward vs. backward search
  - Lots of open questions.
  - For us, backward can be much better than forward.
- Synchronization affects efficiency.
- Can exploit high-level knowledge to do optimizations.
  - Can be much more efficient than using model checker as a black box.
SMC vs. Theorem Proving

Similarity: *Pre* is essentially the dual of *WP*.

Some key differences:

<table>
<thead>
<tr>
<th>SMC</th>
<th>Theorem Proving</th>
</tr>
</thead>
<tbody>
<tr>
<td>finite-state</td>
<td>no assumption</td>
</tr>
<tr>
<td>can be automated</td>
<td>need user guidance</td>
</tr>
<tr>
<td>efficient representations</td>
<td>readable representations</td>
</tr>
<tr>
<td>counterexamples (if false)</td>
<td>inspiring proofs (if true)</td>
</tr>
</tbody>
</table>

- MC is more useful because most systems are buggy!
- In MC, you gain confidence in correctness thru experiments.
- Much current work on *infinite-state* SMC.