Symbolic Model Checking for Large Software Specifications

William Chan
Richard Anderson
Paul Beame
Steve Burns
Francesmary Modugno
David Jones (Boeing)
David Notkin
Jon D. Reese
William Warner (Boeing)

• How to increase confidence in correctness of safety-critical software?
• Existing techniques are limited to some degree
  – Inspection
  – Syntactic check
  – Simulation/testing
  – Theorem proving
• Symbolic model checking successful for industrial hardware
  – Effective also for software?
  – Many people’s conjecture: No
Temporal-Logic Model Checking [Clarke & Emerson 81]

- Some properties expressible in temporal logics
  - Error states not reached (invariant)
    - Ex: $AG \neg \text{Err} \iff$ Today’s focus
  - Eventually acknowledge for each request (liveness)
    - $AG (\text{Req} \rightarrow AF \text{Ack})$
  - Always possible to restart machine (possibility)
    - $AG \text{EF Restart}$
Two Approaches to Model Checking

• Explicit
  – Conventional state-space search: depth-first, breadth-first, etc.
  – Needs substantial manual abstraction and state reduction

• Symbolic
  – Can search huge state spaces (e.g. $10^{20}$)
  – Practical for many industrial hardware circuits
  – Provably bad for certain arithmetic operations.
  – Not believed to work well for software
Software Experts Said

• “The time and space complexity of the symbolic approach is affected...by the regularity of specification. Software requirements specifications lack this necessary regular structure...” [Heimdahl & Leveson 96]
And say...

- “[Symbolic model checking] works well for hardware designs with regular logical structures...However, it is less likely to achieve similar reductions in software specifications whose logical structures are less regular.” [Cheung & Kramer 99]
Model Checking Co-Inventor Says

• “…[symbolic model checkers] are often able to exploit the regularity…in many hardware designs. Because software typically lacks this regularity, [symbolic] model checking seems much less helpful for software verification.” [Emerson 97]
Contributions

• Case Studies: successfully analyzed state-machine specifications of
  – TCAS II (aircraft collision avoidance system) [FSE 96, TSE 98]
  – Electrical power distribution (EPD) system on Boeing 777 [ICSE 99, TSE 00]
• Optimizations: obtained orders-of-magnitude speedup [ISSTA 98, ICSE 99, TSE 00]
  – Developed intuitions about efficiency
  – Enabled difficult analyses
• Extension: handle complicated arithmetic
  – Combine with a constraint-satisfaction engine [CAV 97]
Invariant Checking as Set Manipulations

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

States that can reach an Error State

Error States

Backward breadth-first search
Explicit vs. Implicit (Symbolic Sets)

- All even numbers between 0 and 127
  - Explicit representation
    - 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 100, 102, 104, 106, 108, 110, 112, 114, 116, 118, 120, 122, 124, 126.
  - Implicit (symbolic) representation
    - $\neg x_0$ ($x_0$: least significant bit)

- Need efficient Boolean-function representation
Symbolic Model Checking [Burch et al. 90, Coudert et al. 89]

• Define Boolean state variables $X$
  – e.g., define $x_{n-1}$, $x_{n-2}$, …, $x_0$ for an $n$-bit integer.

• A state set becomes a Boolean function $S(X)$
  – e.g., $x_0$ for the set of $n$-bit even integers.

• Set operations ($\cap$, $\cup$) become Boolean operations ($\land$, $\lor$)

• Transition relation: $R(X,X)$.

• Compute predecessors also using Boolean operations
  – $\text{Pre } (S) = \exists X'. S(X') \land R(X,X')$
Binary Decision Diagrams (BDDs) [Bryant 86]

- DAGs, evaluated like binary decision trees.
- Efficiency depends on BDD size
  - Usually small; some large hardware circuits can be handled
  - Some well-known limitations
    - e.g., exponential size for $a > bc$
    - Few theoretical results known
    - Performance unpredictable

Odd Parity
Symbolic Model Checking Ineffective for Software?

<table>
<thead>
<tr>
<th></th>
<th>Hardware</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>Simple</td>
<td>Complex</td>
</tr>
<tr>
<td>States</td>
<td>Finite</td>
<td>Infinite</td>
</tr>
<tr>
<td>Concurrency</td>
<td>Synchronous</td>
<td>Asynchronous</td>
</tr>
<tr>
<td>Strategy</td>
<td>Symbolic search</td>
<td>Abstraction and explicit search</td>
</tr>
</tbody>
</table>

This common view may be true for software like multi-threaded programs, but…
Consider Safety-Critical Software

- Most costly bugs in specification
- Use analyzable formal specification
  - State-machine specifications
  - Intuitive to domain experts like aircraft engineers
  - Statecharts [Harel 87], RSML [Leveson et al. 94], SCR [Parnas et al.], etc.
Model-Check the Spec!

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

- Symbolic model checking good for such specs?
- Develop more intuitions about efficiency? Optimize analyses?
- How to handle arithmetic?
Case Study 1: TCAS II

• Traffic Alert and Collision Avoidance System
  – Reduce mid-air collisions
    • Warns pilots of traffic
    • Issues resolution advisories
  – Required on most commercial aircraft
  – “One of the most complex systems on commercial aircraft.”

• 400-page specification reverse-engineered from pseudo-code

• Written in RSML by Leveson et al., based on statecharts
Case Study 2: EPD System

- Electrical Power Distribution system used on Boeing 777
- Distribute power from sources to buses via circuit breakers
  - Tolerate failures in power sources and circuit breakers
- Prototype specification in statecharts
- Analysis joint with Jones and Warner of Boeing
Model Check the Specifications

Spec → (with simple abstraction) → Boolean encoding → Behavioral Property → Model Checker

Model Checker → TRUE

Model Checker → FALSE (with counterexample)
Translation to SMV

VAR
A: \{0,1\};
x: boolean;
y: boolean;

ASSIGN
init (A):= 0;
next (A):= case
A=0 & x & c : 1;
1 : A;
esac;
...

Analyses and Results

• Used and modified SMV [McMillan 93]

<table>
<thead>
<tr>
<th></th>
<th>TCAS II</th>
<th>EPD System</th>
</tr>
</thead>
<tbody>
<tr>
<td>State space</td>
<td>230 bits, $10^{60}$ states</td>
<td>90 bits, $10^{27}$ states</td>
</tr>
<tr>
<td>Prior verification</td>
<td>inspection, static analysis</td>
<td>simulation</td>
</tr>
<tr>
<td>Problems we found</td>
<td>inconsistent outputs, safety violations, etc.</td>
<td>violations of fault tolerance</td>
</tr>
</tbody>
</table>

• Optimizations crucial for successful model checking
Some Formulas Checked

• TCAS II
  – Descent inhibition
    • AG (Alt < 1000 → ¬Descend)
  – Output agreement
    • AG ¬(GoalRate ≥ 0 ∧ Descend)

• EPD system
  – AG (NoFailures → (LMain ∧ RMain ∧ LBackup ∧ RBackup))
  – AG (AtMostOneFailure → (LMain ∧ RMain))
  – AG (AtMostTwoFailures → (LBackup ∨ RBackup))
A Counterexample Found

- A single failure can cause a bus to lose power
  1. Power-up sequence; normal operation
  2. A circuit breaker fails
  3. Other circuit breakers reconfigured to maintain power
  4. User changes some inputs
  5. The first circuit breaker recovers
  6. User turns off a generator
  7. A bus loses power

This error does not exist in onboard system
• **Synchrony hypothesis**
  – No new inputs within macrostep
  – Macrostep encoded as a sequence of transitions
  – Statecharts, Esterel [Berry & Gonthier 92], Lustre [Halbwachs et al. 92], etc.
Synchronization in Statecharts

- Event-driven
- Label: trigger[guard]/action
Forward vs. Backward Search

• Generally unclear which is better
• Forward search
  – Often good for low-level hardware.
  – But always bad for us; large BDDs
• Focus on backward search
A Disadvantage of Backward Search

- Visiting unreachable states

Reachable States
Use Known Invariants for Pruning

- Need known invariants that are
  - small as BDDs and
  - effective in reducing BDD size
Optimization 1: Mutual Exclusion of Transitions

- Many “concurrent” transitions are sequential
  - Determine using static analysis
- Use this to prune backward search
Overall Effects on TCAS II

Without pruning

With pruning

P1, P2, P3, P4, P5, P6

Min.

>> 1 hour

0 1 2 3 4 5 6 7 8 9 10
Initial EPD Analyses Failed

- Even though it has fewer states than TCAS II

<table>
<thead>
<tr>
<th></th>
<th>TCAS II</th>
<th>EPD System</th>
</tr>
</thead>
<tbody>
<tr>
<td>State space</td>
<td>230 bits, $10^{60}$ states</td>
<td>90 bits, $10^{27}$ states</td>
</tr>
</tbody>
</table>

- Main difference in synchronization
Oblivious Synchronization (used in TCAS II)

- y signals completion of machine A
  - Macrostep length: 2
  - x → y → stable
Non-Oblivious Synchronization (used in EPD)

- $y$ signals state change in machine A
- Macrostep length: 1 or 2
  - $x \rightarrow y \rightarrow$ stable
  - $x \rightarrow$ stable
Oblivious Synchronization: General Case

- Event sequence always identical
  - Thus, every macrostep has the same length
Backward Search for Oblivious Synchronization

• Yields small BDDs
Non-Oblivious Synchronization: General Case

• Macrosteps may have different lengths.

```
| x | y, z | w | z |
```

```
| x | w | z |
```

```
| x |
```
Backward Search for Non-Oblivious Synchronization

- Larger BDDs
Optimization 2: Restoring Regularity in State Sets [ICSE 99]

• Automatic semantics-preserving transformation:

• Add stuttering states.
• Preserve most properties, e.g., invariants and eventualities. [Lamport 83, Browne et al. 89]
New Backward Search

✓ Make every macrostop equal in length. Smaller BDDs.
✗ Increase # states and # state variables.
✗ Increase # iterations to reach fixed points.
Other Optimizations [ISSTA 98]

• Partition transition relation in various ways.
  — Use multiple BDDs for transition relation.
• Abstract automatically be dependency analysis.
  — Remove part of system that can’t affect result.
• Improve counter-example search.
  — Avoid work in forward search.
Arithmetic

– TCAS II spec contains nontrivial arithmetic.
  • Variables assumed discrete and bounded.
  • Encoded as bits.
  • Ok for linear constraints (e.g., $a+b > c$).
  • But nonlinear constraints abstracted away.

– *BMD [Bryant & Chen] and HDD [Clarke & Zhao 95] do not help.
  • Good for $ab$
  • But not $ab > c$. 
Why Nonlinear Constraints Hard?

Recall: \( \text{Pre}(S) = X \cdot S(X) \cdot R(X,X) \).

- Assume \( X \) has \( n \) numeric variables.
- Project \( 2n \)-dimensional regions on \( n \)-dimensional space.
- Hard if \( S \) and \( R \) nonlinear.

May not need to solve the general problem.

Assume:

- All numeric variables are inputs,
- and unconstrained at beginning of macrostep
Different Approach

• Represent each constraint as a BDD variable.
  – Overly conservative if infeasible constraints not detected.
• Detect infeasible paths using constraint-satisfaction engine (black box).
• Prune infeasible paths from BDD (“filtering”).
Some Lessons Learned

• Focus on restricted models that people care about.
• Exploit high-level knowledge to improve analysis.
  – Synchronization, environmental assumptions, etc.
  – In addition to low-level BDD tricks.
• Combine static analysis and symbolic model checking.
• Help understand system behaviors.
  – In addition to verification/falsification.
How General are the Techniques?

- Optimizations specific to events, macrosteps, and the synchrony hypothesis:
  - Maybe applicable to synchronous programming languages.
- Combining forward static analysis and backward symbolic search:
  - Seems promising.
- Constraint-satisfaction approach:
  - Applicable if environment not constrained.