Logic in Automatic Verification

Javier Esparza

Software Reliability and Security Group
Institute for Formal Methods in Computer Science
University of Stuttgart

Many thanks to Abdelwaheb Ayari, David Basin, Armin Biere, Paul Gastin and Denis Oddoux
Automatic verification

The dream:

- feed a machine with a system and a specification
- push a button
- get ‘yes’ or ‘no, because . . .’

In this talk: three small examples of application of decision procedures for logics to this problem

- SAT / QBF
- Temporal logics
- Monadic second order logics
Verifying adders with boolean logic
Modelling circuits with QBL

Gates as boolean formulas

Stable states as satisfying truth assignments

\[ \text{not} \equiv \neg a \leftrightarrow b \]
\[ \text{and} \equiv (a \land b) \leftrightarrow c \]
\[ \text{or} \equiv (a \lor b) \leftrightarrow c \]
\[ \text{xor} \equiv ((\neg a \land b) \lor (a \land \neg b)) \leftrightarrow c \]
Combine gates with $\land$, $\exists$ (and renaming of variables)

$$R(x, y, q, r, s) = \exists w. R_1(x, y, w, q) \land R_2(y, w, r, s)$$
A full adder

\[ \text{full_adder}(a, b, s, \text{cin}, \text{cout}) \equiv \]
\[ \exists w_1, w_2, w_3. \text{xor}(a, b, w_1) \land \text{xor}(w_1, \text{cin}, \text{cout}) \land \text{and}(a, b, w_2) \land \]
\[ \text{and}(\text{cin}, w_1, w_3) \land \text{or}(w_3, w_2, \text{cout}) \]
An $n$-bit ripple-carry adder

\[
\begin{align*}
  c_2 & \quad c_1 & \quad \text{cin} \quad (= 0) \\
  a_2 & \quad a_1 & \quad a_0 \\
  + & \quad b_2 & \quad b_1 & \quad b_0 \\
  \text{cout} & \quad s_2 & \quad s_1 & \quad s_0
\end{align*}
\]

Wire together $n$ 1-bit adders where $i$th carry-out is $i+1$st carry-in, first carry is the carry-in and last is the carry-out.
We obtain the formula

\[
\text{adder}_n(a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1}, s_0, \ldots, s_{n-1}, \ cin, \ cout) \equiv \\
\exists c_0, \ldots, c_n. (c_0 \leftrightarrow \ cin) \land (c_n \leftrightarrow \ cout) \land \\
\bigwedge_{i=1}^{n-1} \text{full\_adder}(a_i, b_i, s_i, c_i, c_{i+1})
\]

Problem: too slow!!

Each \(c_i\) can only be computed after all of \(c_{i-1}, \ldots, c_0\) have been computed

Delay: \(2n + 2\) time units for \(n\)-bit numbers
A carry-look-ahead $n$-adder

Compute all of $c_{n-1}, \ldots, c_0$ (and cout) concurrently

First step: given $a_{n-1} \ldots a_0$ and $b_{n-1} \ldots b_0$, identify the $i \in [0, n-1]$ that are

- Generating: $c_{i+1} \equiv 1$ independently of $c_i$.
  These are the positions such that $1 = g_i \equiv \text{and}(a_i, b_i)$.

- Propagating: $c_{i+1} \equiv c_i$, i.e., $c_i$ is ‘propagated’ to $c_{i+1}$.
  These are the positions such that $1 = p_i \equiv \text{xor}(a_i, b_i)$

Observe: all $g_i, p_i$ can be computed simultaneously

Second step: compute the $c_i$’s by

$$c_i \equiv g_i \lor (p_i \land g_{i-1}) \lor (p_i \land p_{i-1} \land g_{i-2}) \lor \ldots \lor (p_i \land p_{i-1} \land \ldots \land g_0)$$

Logarithmic delay in $n$ using balanced $\lor$-trees and $\land$-trees

Delay depends on tree structure. For 64 bits: 23-56 units (instead of 130)
Description of the circuit

\[ \text{cin} \rightarrow \text{RootCell} \rightarrow \text{cout} \]

\[ \text{NodeCell} \]

\[ \begin{array}{c}
\text{LeafCell} \\
 a_0 b_0 s_0 \\
\text{LeafCell} \\
 a_1 b_1 s_1 \\
\text{LeafCell} \\
 a_2 b_2 s_2 \\
\text{LeafCell} \\
 a_3 b_3 s_3
\end{array} \]
Description of the circuit II

**LeafCell** circuit

**⊕** circuit

**NodeCell** circuit

**RootCell** circuit
Verification of the carry-look-ahead $n$-adder

Check if

\[ \text{adder}_n(a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1}, s_0, \ldots, s_{n-1}, \text{cin}, \text{cout}) \]

\[ \iff \]

\[ \text{cla}_n(a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1}, s_0, \ldots, s_{n-1}, \text{cin}, \text{cout}) \]

Use SAT solvers or QBF solvers

Results of the SAT 2002 competition on a variant of this problem:

- Task was to compare 2, 4, 8, \ldots, 256 bits adders (8 problems)
- From 26 variables and 70 3CNF clauses to 4584 variables and 13226 clauses
- Fastest solver (Zchaff) checked all 8 problems in 14 seconds

Rule-of-thumb: circuits with some hundreds of gates are routinely solved
Verifying mutual exclusion algorithms with propositional temporal logics
The mutual exclusion problem

The setting:

Two computers connected to a database (e.g., of plane bookings)
Can communicate with each other through shared variables (i.e., variables that both can read and write)
Both computers run a program having a critical section, from which the program can update the database records

The problem: design the program run by the computers so that

At any time, at most one computer can be in the critical section
If a computer wishes to enter the critical section, it eventually will
These properties still hold if any of the computers breaks down in the non-critical section

Observe: not an input/output system!
A solution due to Peterson

\[
\text{var} \quad \text{flag}[0], \text{flag}[1] : \{\text{true}, \text{false}\} \text{ init false;}
\]
\[
\text{var} \quad \text{turn} : \{0,1\};
\]

\[
\text{while true do}
\]
\[
\begin{align*}
\text{s}_0 & \quad \text{non-critical section} \\
\text{s}_1 & \quad \text{flag}[0] := \text{true;} \\
\text{s}_2 & \quad \text{turn := 1;} \\
\text{s}_3 & \quad \text{while (flag[1] and turn=1) skip ;} \\
\text{s}_4 & \quad \text{critical section} \\
\text{s}_5 & \quad \text{flag}[0] := \text{false;} \\
\text{od}
\end{align*}
\]

\[
\text{while true do}
\]
\[
\begin{align*}
\text{r}_0 & \quad \text{non-critical section} \\
\text{r}_1 & \quad \text{flag}[1] := \text{true;} \\
\text{r}_2 & \quad \text{turn := 0;} \\
\text{r}_3 & \quad \text{while (flag[0] and turn=0) skip ;} \\
\text{r}_4 & \quad \text{critical section} \\
\text{r}_5 & \quad \text{flag}[1] := \text{false;} \\
\text{od}
\end{align*}
\]
Linear-time temporal logic (LTL)

Built on top of a set $AP$ of atomic propositions

**World**: valuation of the atomic propositions over $\{\text{true, false}\}$

Formulas of LTL interpreted over runs: infinite sequences of worlds

Notation:
- $\text{run } \rho = \rho_0 \rho_1 \rho_2 \cdots$
- $\text{suffix } \rho|_i = \rho_i \rho_{i+1} \rho_{i+2} \cdots$
<table>
<thead>
<tr>
<th>Type</th>
<th>Formula</th>
<th>$\rho \models \varphi$ iff ...</th>
<th>Intuition</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic</td>
<td>$p$</td>
<td>$p$ is true at $\rho_0$</td>
<td>$p$ holds now</td>
</tr>
<tr>
<td>boolean</td>
<td>$\neg \varphi$</td>
<td>$\rho \not\models \varphi$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\varphi \lor \psi$</td>
<td>$\rho \models \varphi$ or $\rho \models \psi$</td>
<td></td>
</tr>
<tr>
<td>temporal</td>
<td>$X \varphi$</td>
<td>$\rho</td>
<td>_1 \models \varphi$</td>
</tr>
<tr>
<td></td>
<td>$F \varphi$</td>
<td>$\rho</td>
<td>_i \models \varphi$ for some $i \in \mathbb{N}$</td>
</tr>
<tr>
<td></td>
<td>$G \varphi$</td>
<td>$\rho</td>
<td>_i \models \varphi$ for all $i \in \mathbb{N}$</td>
</tr>
<tr>
<td></td>
<td>$\varphi \mathcal{U} \psi$</td>
<td>there is $i \in \mathbb{N}$ such that $\rho</td>
<td>_i \models \psi$ and $\rho</td>
</tr>
</tbody>
</table>
Application to the mutex algorithm

Atomic propositions: flag[0] = \texttt{true}, at_s_4, \ldots

The program satisfies a property if all its runs (executions) satisfy it

The mutual exclusion property:

$$G(\neg at_{s_4} \lor \neg at_{r_4})$$

If computer 0 wants to enter the critical section, it eventually will:

$$G(\text{flag}[0] = \texttt{true} \Rightarrow F at_{s_4})$$

But this property does not take breakdowns out of the non-critical section into account \ldots
Dealing with breakdowns

Introduce propositions $\text{l}ast_0$, $\text{l}ast_1$ saying which computer did the last step

No breakdowns for computer 0:

$$G \ F \ last_0$$

No breakdowns for computer 0 but possibly in the non-critical section:

$$G \ F \ last_0 \lor F \ G \ at\_s_0$$

The final property to be checked:

$$(G \ F \ last_0 \lor F \ G \ at\_s_0) \land (G \ F \ last_1 \lor F \ G \ at\_r_0)$$

$$\implies$$

$$G(\text{flag}[0] = \text{true} \Rightarrow F \ at\_r_4) \land G(\text{flag}[1] = \text{true} \Rightarrow F \ at\_s_4)$$
Automatic verification

The **model-checking** problem: whether all runs of the algorithm satisfy a given LTL formula

Can be algorithmically solved in three steps (Vardi, Wolper 85):

- Construct a **Büchi automaton** for the negation of the formula (decision procedure for satisfiability)
- Construct the product of this automaton and the state space of the system
- Check emptiness of the product

**Linear complexity** in the number of states of the program, **exponential complexity** in the size of the formula

Formula verified in less than one second with Holzmann’s SPIN checker (http://spinroot.com/)
Automaton for the formula

LTL2BA by Gastin and Oddoux (www.liafa.jussieu.fr/oddoux/ltl2ba/)

Quite sophisticated: formula $\rightarrow$ alternating Büchi $\rightarrow$ generalized Büchi $\rightarrow$ Büchi, with simplification heuristics at each step

The automaton for the negation of the formula has 36 states
Verifying parameterized adders
with monadic second order logics
WS1S : weak MSO logic of one successor

First order variables $p, q, \ldots$ interpreted over $\mathbb{N}$

Second-order variables $X, Y, \ldots$ interpreted over finite subsets of $\mathbb{N}$

$\phi ::= p = s(q) \mid p \in X \mid \neg \phi \mid \phi \lor \phi \mid \exists p. \phi \mid \exists X. \phi$
## Definitions (Sample)

<table>
<thead>
<tr>
<th>Definition</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\phi_1 \land \phi_2$</td>
<td>$\equiv \neg(\neg\phi_1 \lor \neg\phi_2)$</td>
</tr>
<tr>
<td>$\forall p. \phi$</td>
<td>$\equiv \neg\exists p. \neg\phi$</td>
</tr>
<tr>
<td>$X(p)$</td>
<td>$\equiv p \in X$</td>
</tr>
<tr>
<td>$X(0)$</td>
<td>$\equiv \exists p. (\forall q. p \neq s(q)) \land X(p)$</td>
</tr>
<tr>
<td>$X(p + n)$</td>
<td>$\equiv \exists p_1, \ldots, p_n. p_1 = s(p) \land \ldots \land p_n = s(p_{n-1}) \land X(p_n)$</td>
</tr>
<tr>
<td>$x = y$</td>
<td>$\equiv \forall X. X(x) \leftrightarrow X(y)$</td>
</tr>
<tr>
<td>$x \leq y$</td>
<td>$\equiv \forall X. (X(y) \land \forall z, w. (X(z) \land s(w) = z \rightarrow X(w)) \rightarrow X(x))$</td>
</tr>
<tr>
<td>$x &lt; y$</td>
<td>$\equiv x \leq y \land \neg(x = y)$</td>
</tr>
</tbody>
</table>
WS1S as a logic of binary strings

Second-order variables interpreted as strings over \{0, 1\}

First-order variables interpreted as positions in the string

‘X(p) holds iff string X has a 1 at position p’

Formula \(\phi\) with free variables determines a language \(L(\phi)\)

\[
1101 \in L(X(1) \land X(3)) \quad 1011 \notin L(X(1) \land X(3))
\]

\(n\) free variables in \(\phi\) determine language over \(\{0, 1\}^n\)

\[
\forall p. \ p < 4 \rightarrow (X(p) \leftrightarrow \neg Y(p))
\]

\[
\begin{array}{cccc}
X & 0 & 1 & 1 & 0 \\
Y & 1 & 0 & 0 & 1 \\
\end{array}
\in L(\phi)
\quad \text{and} \quad
\begin{array}{cccc}
X & 0 & 1 & 1 \\
Y & 0 & 0 & 0 \\
\end{array}
\notin L(\phi)
\]
Examples

\[ \exists p, q. p \neq q \land X(p) \land X(q) \]

– \( X \) is a string with a 1 in at least 2 positions, e.g., 010100

\[ \exists p. \ (\forall q. p \neq s(q)) \land X(p) \]

– \( X \) is a string whose initial letter is 1

\[ \forall p. X(p) \leftrightarrow Y(s(p)) \]

– \( Y \) is \( X \) ‘right-shifted’ 1 position, e.g.,

\[
\begin{array}{ccc}
0 & 1 & 1 \\
0 & 1 & 1 \\
\end{array}
\]
Well-known results

Satisfiability of WS1S is decidable in non-elementary time
(each quantifier alternation adds one exponential)

The language $L(\phi)$ is regular

A finite automaton accepting $L(\phi)$ can be computed directly from $\phi$
Modelling the family of **ALL** ripple-carry adders

Recall the formula for a ripple carry $n$-adder

$$\text{adder}_n(a_0, \ldots, a_{n-1}, b_0, \ldots, b_{n-1}, s_0, \ldots, s_{n-1}, \text{cin}, \text{cout}) \equiv$$

$$\exists c_0, \ldots, c_n. (c_0 \leftrightarrow \text{cin}) \land (c_n \leftrightarrow \text{cout}) \land$$

$$\bigwedge_{i=0}^{n-1} \text{full_adder}(a_i, b_i, s_i, c_i, c_{i+1})$$

We construct the WS1S formula

$$\text{adder}(n, A, B, S, \text{cin}, \text{cout}) \equiv$$

$$\exists C. (C(0) \leftrightarrow \text{cin}) \land (C(n) \leftrightarrow \text{cout}) \land$$

$$\forall p. p < n \rightarrow \text{full_adder}(A(p), B(p), S(p), C(p), C(p + 1)) \land$$

$$\forall p. p \geq n \rightarrow (\neg A(p) \land \neg B(p) \land \neg S(p))$$
A model of **adder**\((A, B, S, cin, cout)\)

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
A & 1 & 1 & 1 & 0 & 0 & 0 \\
B & 1 & 0 & 0 & 1 & 0 & 0 \\
S & 0 & 0 & 0 & 0 & 1 & 0 \\
\end{array}
\]

\(n = 4\)  
\(cin = 0\)  
\(cout = 0\)

The set of models of **adder** is ‘the union’ of all the sets of models of **adder**\(_n\)
WS2S : weak MSO logic of two successors

Seen as a logic over binary trees

Second-order variables interpreted as trees over \{0, 1\}

First-order variables interpreted as positions (nodes) in the tree

Example:

\[
X(\epsilon) \land (\forall p. X(s_0(p)) \leftrightarrow X(s_1(p))) \land \forall p. \neg Y(s_0(p)) \lor \neg Y(s_1(p))
\]

‘\(X\) contains the root node \(\epsilon\), and

a node \(p\) is in \(X\) iff its brother is also in \(X\), and

for any node \(p\), \(Y\) contains at most one of \(p\)’s successors’
Models

A model of a formula with $n$ free variables is a ‘superposition’ of trees over $B$, i.e., a tree whose nodes are labelled with elements of $\{0, 1\}^n$.

The tree

```
0
  1
  0
  1 1
  1 0
  1 1 0 0
0 1 0 1
```

is a model of

\[ X(\epsilon) \land (\forall p. X(s_0(p)) \leftrightarrow X(s_1(p))) \land \forall p. \neg Y(s_0(p)) \lor \neg Y(s_1(p)) \]
Modelling the family of ALL carry-look-ahead adders
The family can be modelled by the formula

\[ \text{cla}(A, B, S, \text{cin}, \text{cout}) \equiv \exists T, E_1, E_2, F_1, F_2 \]
\[ \text{RootCell}(F_1, F_2, E_1, E_2, \text{cin}, \text{cout}) \land \]
\[ (\forall p. (\text{leaf}(p, T) \rightarrow \text{LeafCell}(A, B, S, F_1, F_2, E_1, E_2, p))) \land \]
\[ (\text{node}(p, T) \rightarrow \text{NodeCell}(F_1, F_2, E_1, E_2, p))) \land \]
\[ \text{shape}_\text{cond}(A, B, S, T) \]
Verification of a parameterized cla-adder

Check validity of

\[ \forall A, B, S, \text{cin}, cout. \text{adder}(A, B, S, \text{cin}, cout) \iff \text{cla}(A, B, S, \text{cin}, cout) \]

(Requires to embed WS1S into WS2S)

Checked in 1 second by MONA (Mona at www.brics.dk/mona)

Restrictions:

- only array or tree structures
- only one parameter (two parameters \( \rightarrow \) quantification on binary relations)
No conclusions, just examples!