Formal Verification at Higher Levels of Abstraction

Daniel Kroening, Oxford University
Sanjit A. Seshia, UC Berkeley

ICCAD Tutorial
November 8, 2007
The Speakers

**Sanjit Seshia**
Electrical Engineering and Computer Sciences
University of California, Berkeley

**Daniel Kroening**
Computing Laboratory
Oxford University

Work described is joint with our students & many collaborators:
R. Bryant, E. Clarke, J. Ouaknine, N. Sharygina, O. Strichman
Level of Abstraction in Design is Increasing

- **System**
- **Behavioral**
- **Register Level**
- **Gate level (netlists)**

SystemC, SystemVerilog, Transactional models, ...

Verilog, VHDL
But Formal Verification is Still Mostly at Bit-Level

- System
- Behavioral
- Register Level
- Gate level (netlists)

Model check
This Talk: Formal Verification at Word-Level or Term-Level

- System
- Behavioral
- Register Level
- Gate level (netlists)

Model check
Outline

- Bit-Vector Decision Procedures
- Term-Level Modeling
- Word-level Predicate Abstraction
- Interpolation
Register-Level Verilog:
module counter_cell(clk, carry_in, carry_out);
input clk;
input carry_in;
output carry_out;
reg value;
assign carry_out = value & carry_in;
initial value = 0;
always @(posedge clk) begin // value = (value + carry_in) % 2;
    case(value)
        0: value = carry_in;
        1: if (carry_in ==0)
            value = 1;
        else value = 0;
    endcase
end
endmodule

Gate Level (netlist):
.model counter_cell
.inputs carry_in
.outputs carry_out
.names value carry_in _n2
.def 0
1 1 1
.names _n2 carry_out$raw_n1
- =_n2
.names value$raw_n3
0
.names _n6
0
.names value _n6 _n7
.def 0
0 1 1
1 0 1
.r value$raw_n3 value
0 0
1 1
.... (120 lines)
Bit-level vs. Word-level

Example bit-level interpolation:

Initial: $i = j + 1$;

$i \leq i + 1$;

$j \leq j + 1$;

- assert $i > j$, $P1$
- assert $i ! = j$, $P2$
- assert $i == j + 1$, $P3$

+ overflow prevention

<table>
<thead>
<tr>
<th></th>
<th>4 bits</th>
<th>8 bits</th>
<th>16 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P1$</td>
<td>37s</td>
<td>&gt;1h</td>
<td>&gt;1h</td>
</tr>
<tr>
<td>$P2$</td>
<td>4s</td>
<td>27s</td>
<td>1:09m</td>
</tr>
<tr>
<td>$P3$</td>
<td>4s</td>
<td>1:34m</td>
<td>2:54m</td>
</tr>
</tbody>
</table>
Verification Tasks of Interest

- Assertion-based Verification (ABV)
- Sequential Equivalence Checking (SEC)

Both for hardware and embedded software
## Contrasting Levels of Formal Verification

<table>
<thead>
<tr>
<th>Level of Abstraction</th>
<th>Computational Engines</th>
<th>Model Generation</th>
<th>Level of Abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit-Level</td>
<td>SAT, BDDs</td>
<td>Transliteration, with optimizations</td>
<td>Word/Term-Level</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Types, Predicates</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **SMT solvers, Predicate abstraction**

**Abstraction based on:**
- Types
- Predicates
## Contrasting Levels of Formal Verification

<table>
<thead>
<tr>
<th>Level of Abstraction</th>
<th>Model Generation</th>
<th>Computational Engines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit-Level</td>
<td>Transliteration, with optimizations</td>
<td>SAT, BDDs</td>
</tr>
<tr>
<td><strong>Word/Term-Level</strong></td>
<td>Abstraction based on:</td>
<td>SMT solvers, Predicate abstraction</td>
</tr>
<tr>
<td></td>
<td>● Types</td>
<td></td>
</tr>
<tr>
<td></td>
<td>● Predicates</td>
<td></td>
</tr>
</tbody>
</table>
Outline

- Term-Level Modeling
- Bit-Vector Decision Procedures
- Word-Level Predicate Abstraction
- Interpolation
## Term/Word-Level Modeling

<table>
<thead>
<tr>
<th>Construct</th>
<th>Word-Level</th>
<th>Term-Level</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data</strong></td>
<td>$x_0, x_1, x_2, \ldots, x_{n-1}$</td>
<td>$x_0, x_1, x_2, \ldots, x_{n-1}$ $\in \mathbb{Z}$</td>
</tr>
<tr>
<td><strong>Function</strong></td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td><strong>Memories</strong></td>
<td>$n$</td>
<td>finite</td>
</tr>
</tbody>
</table>

- Function: $f^n : (x_0, x_1, x_2, \ldots, x_{n-1}) \rightarrow M(a) = w_a$
Motivating Example

Pipelined Microprocessor

Sequential Reference Model

- **Does pipelined microprocessor implement sequential reference model?**

- **Strategy**
  - Verify by correspondence checking [Burch & Dill, CAV ’94]
  - Represent machine instructions, data, and pipeline state as bit vectors
    - Functional blocks like ALU abstracted with uninterpreted functions
Term-level and word-level modeling and verification is implemented in the UCLID Verification System (a joint UC Berkeley – CMU project)

http://uclid.eecs.berkeley.edu/wiki

Here we will focus on the computational engine for word-level reasoning

- Decision procedure for bit-vector arithmetic
Focus: Bit-Vector Arithmetic

- Bit Vector Formulas
  - Types: Fixed width data words
  - Arithmetic and relational operations
    - E.g., add/subtract/multiply/divide/mod & comparisons
    - Two’s complement, unsigned, ...
  - Bit-wise logical operations
    - E.g., bit-wise and/or/xor, shift, extract/concatenate
  - Boolean connectives

- Many Applications for both Hardware and Software
  - Formal verification of hardware designs
    - Based on model checking, equivalence checking, theorem proving, ...
  - Software model checking & static analysis
  - Test/exploit generation
  - Generating signatures of malware (worms/viruses/…)

11/8/2007 Daniel Kroening, Sanjit A. Seshia
The Problem

Is $\varphi$ satisfiable?

$\varphi$

$x = y$

$x + 2z \leq 1$

$a$

$w \& 0xFFFF = x$

$x \% 26 = v$

E.g.: Any Verilog/C Boolean expression
Decision Procedures/SMT Solvers

- Core technology for formal reasoning

- Boolean SAT
  - Pure Boolean formula

- SAT Modulo Theories (SMT)
  - Decide more expressive (first-order) logics
  - Example theories
    - Linear arithmetic over reals or integers
    - Functions with equality
    - *Bit vector arithmetic*
    - Array/memory operations
    - Combinations of theories

Most SMT Solvers translate to SAT!
UCLID Experience with SAT Solving

Run-time (sec.)

<table>
<thead>
<tr>
<th>Tool</th>
<th>Time (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grasp (2000)</td>
<td>3600</td>
</tr>
<tr>
<td>zChaff (2001)</td>
<td>766</td>
</tr>
<tr>
<td>BerkMin (2002)</td>
<td>147</td>
</tr>
<tr>
<td>zChaff (2003-04)</td>
<td>118</td>
</tr>
<tr>
<td>Siege (2004)</td>
<td>81</td>
</tr>
<tr>
<td>SatEliteGTI (2005)</td>
<td>46</td>
</tr>
<tr>
<td>Rsat (2007)</td>
<td>19</td>
</tr>
</tbody>
</table>

(on a single benchmark)
BV Decision Procedures: Some History

- B.C. (Before Chaff)
  - String operations (concatenate, field extraction)
  - Linear arithmetic with bounds checking
  - Modular arithmetic

- SAT-Based “Bit Blasting”
  - Generate Boolean circuit based on bit-level behavior of operations
  - Convert to Conjunctive Normal Form (CNF) and check with best available SAT checker
  - Handles arbitrary operations
  - Effective in many applications
    - CBMC [Clarke, Kroening, Lerda, TACAS ’04]
    - Microsoft Cogent + SLAM [Cook, Kroening, Sharygina, CAV ’05]
    - CVC-Lite [Dill, Barrett, Ganesh], Yices [deMoura, et al], STP (early version) [Ganesh & Dill]
Research Challenge

- *Is there a better way than bit blasting?*

- **Requirements**
  - Provide same functionality as with bit blasting
    - Must support all bit-vector operators
  - Exploit word-level structure
  - Improve on performance of bit blasting

- **Current Approaches based on two core ideas:**
  1. **Simplification**: Simplify input formula using word-level rewrite rules and solvers
  2. **Abstraction**: Use automatic abstraction-refinement to solve simplified formula
Bit-Vector Decision Procedures, circa 2007

- Current Techniques with Sample Tools
  - *Proof-based abstraction-refinement* – UCLID [Bryant et al., TACAS ’07]
  - *Solver for linear modular arithmetic* to simplify the formula – STP [Ganesh & Dill, CAV’07]
  - *Counterexample-guided abstraction-refinement, layered approach, rewriting* – MathSAT [Bruttomesso et al, CAV’07]
  - *Automatic parameter tuning* – Spear [Hutter et al., FMCAD ’07]
Abstraction-Refinement Approach

- Deciding Bit-Vector Arithmetic with Abstraction
  - [Bryant, Kroening, Ouaknine, Seshia, Strichman, Brady, TACAS ’07]
  - Use bit blasting as core technique
  - Apply to simplified versions of formula: under and over approximations
  - Generate successive approximations until a solution is found or formula shown unsatisfiable

- Inspired by McMillan & Amla’s proof-based abstraction for finite-state model checking
Approximations to Formula

- **Overapproximation**
  - $\phi +$
  - $\phi \Rightarrow \phi +$
  - More solutions: If unsatisfiable, then so is $\phi$

- **Original Formula**
  - $\phi$

- **Underapproximation**
  - $\phi -$
  - $\phi - \Rightarrow \phi$
  - Fewer solutions: Satisfying solution also satisfies $\phi$

- **Example Approximation Techniques**
  - **Underapproximating**
    - Restrict word-level variables to smaller ranges of values
  - **Overapproximating**
    - Replace subformula with Boolean variable
Starting Iterations

- Initial Underapproximation
  - (Greatly) restrict ranges of word-level variables
  - Intuition: Satisfiable formula often has small-domain solution
First Half of Iteration

- **SAT Result for** $\varphi_1^-$
  - **Satisfiable**
    - Then have found solution for $\varphi$
  - **Unsatisfiable**
    - Use UNSAT proof to generate overapproximation $\varphi_1^+$
    - (Described later)

- **UNSAT proof:** generate overapproximation
- **If SAT, then done**

Diagram:
- $\varphi_1^+$
- $\varphi$
- $\varphi_1^-$

11/8/2007
Second Half of Iteration

SAT Result for $\varphi_1^+$

- Unsatisfiable
  - Then have shown $\varphi$ unsatisfiable
- Satisfiable
  - Solution indicates variable ranges that must be expanded
  - Generate refined underapproximation

If UNSAT, then done
Iterative Behavior

- **Underapproximations**
  - Successively more precise abstractions of $\varphi$
  - Allow wider variable ranges

- **Overapproximations**
  - No predictable relation
  - UNSAT proof not unique
Overall Effect

- **Soundness**
  - Only terminate with solution on underapproximation
  - Only terminate as UNSAT on overapproximation

- **Completeness**
  - Successive underapproximations approach $\varphi$
  - Finite variable ranges guarantee termination
    - In worst case, get $\varphi_k^- = \varphi$
Generating Overapproximation

- **Given**
  - Underapproximation $\varphi_1^-$
  - Bit-blasted translation of $\varphi_1^-$ into Boolean formula
  - Proof that Boolean formula unsatisfiable

- **Generate**
  - Overapproximation $\varphi_1^+$
  - If $\varphi_1^+$ satisfiable, must lead to refined underapproximation
    - Generate $\varphi_2^-$ such that $\varphi_1^- \Rightarrow \varphi_2^- \Rightarrow \varphi$

UNSAT proof: generate overapproximation
Bit-Vector Formula Structure

- DAG representation to allow shared subformulas

\[
x = y
\]
\[
x + 2z \leq 1
\]
\[
\text{w \& 0xFFFF} = x
\]
\[
x \% 26 = v
\]
Structure of Underapproximation

- Translation to CNF
  - Each word-level variable encoded with vector of Boolean variables
  - Additional Boolean variables represent subformula values
Encoding Range Constraints

- **Explicit**
  - View as additional predicates in formula

- **Implicit**
  - Reduce number of variables in encoding
    - **Constraint**
      - \(0 \leq w < 8\)
      - \(-4 \leq x < 4\)
    - **Encoding**
      - \(0 0 0 \cdots 0 w_2w_1w_0\)
      - \(x_sx_sx_s\cdots x_sx_sx_1x_0\)

- Yields smaller SAT encodings
UNSAT Core

- Subset of clauses that is unsatisfiable
- Variables in unsat core define portion of DAG
- Subgraph that cannot be satisfied with given range constraints

\[ x + 2z \leq 1 \]
\[ w \land 0xFFFF = x \]
\[ x \% 26 = v \]
Generated Overapproximation

- Identify subformulas containing no variables from UNSAT proof
- Replace by fresh Boolean variables
- Remove range constraints on word-level variables
- Creates overapproximation
  - Ignores correlations between values of subformulas

\[ x = y \]
\[ x + 2z \leq 1 \]
\[ a \]
\[ b_1 \]
\[ b_2 \]
Refinement Property

Claim

- $\phi_1^+$ has no solutions that satisfy $\phi_1^-$

- Because $\phi_1^+$ contains portion of $\phi_1^-$, that was shown to be unsatisfiable under range constraints

Implication

- Can only satisfy $\phi_1^+$ by expanding variable ranges

\[
\begin{align*}
    x &= y \\
    x + 2z &\leq 1 \\
    a &\leq b_1 \\
    b_2 &\geq b
\end{align*}
\]
Effect of Iteration

Each Complete Iteration
- Expands ranges of some word-level variables
- Creates refined underapproximation

SAT: Use solution to generate refined underapproximation

UNSAT proof: generate overapproximation
Approximation Methods

- So Far
  - Range constraints
    - Underapproximate by constraining values of word-level variables
  - Subformula elimination
    - Overapproximate by assuming subformula value arbitrary

- General Requirements
  - Systematic under- and over-approximations
  - Way to connect from one to another

- Goal: Devise Additional Approximation Strategies
Function Approximation Example

Motivation
- Multiplication (and division) are difficult cases for SAT

§: Prohibited
- Gives underapproximation
- Restricts values of (possibly intermediate) terms

§: \( f(x, y) \)
- Overapproximate as uninterpreted function \( f \)
- Value constrained only by functional consistency
### Results: UCLID BV vs. Bit-blasting

<table>
<thead>
<tr>
<th>Formula</th>
<th>Ans.</th>
<th>Bit-Blasting Run-time (sec.)</th>
<th>UCLID Run-time (sec.)</th>
<th>STP (sec.)</th>
<th>Yices (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Enc.</td>
<td>SAT</td>
<td>Total</td>
<td>Enc.</td>
</tr>
<tr>
<td>Y86-std</td>
<td>UNSAT</td>
<td>17.91</td>
<td>TO</td>
<td>TO</td>
<td>23.51</td>
</tr>
<tr>
<td>Y86-btnft</td>
<td>UNSAT</td>
<td>17.79</td>
<td>TO</td>
<td>TO</td>
<td>26.15</td>
</tr>
<tr>
<td>s-40-50</td>
<td>SAT</td>
<td>6.00</td>
<td>33.46</td>
<td>39.46</td>
<td>106.32</td>
</tr>
<tr>
<td>BBB-32</td>
<td>SAT</td>
<td>37.09</td>
<td>29.98</td>
<td>67.07</td>
<td>19.91</td>
</tr>
<tr>
<td>rfunitflat-64</td>
<td>SAT</td>
<td>121.99</td>
<td>32.16</td>
<td>154.15</td>
<td>19.52</td>
</tr>
<tr>
<td>C1-P1</td>
<td>SAT</td>
<td>2.68</td>
<td>45.19</td>
<td>47.87</td>
<td>2.61</td>
</tr>
<tr>
<td>C1-P2</td>
<td>UNSAT</td>
<td>0.44</td>
<td>TO</td>
<td>TO</td>
<td>2.24</td>
</tr>
<tr>
<td>C3-OP80</td>
<td>SAT</td>
<td>14.96</td>
<td>TO</td>
<td>TO</td>
<td>14.54</td>
</tr>
<tr>
<td>egt-5212</td>
<td>UNSAT</td>
<td>0.064</td>
<td>0.003</td>
<td>0.067</td>
<td>0.163</td>
</tr>
</tbody>
</table>

[results on 2.8 GHz Xeon, 2 GB RAM]

- **UCLID always better than bit blasting**
- **Generally better than other available procedures**
- **SAT time is the dominating factor**
Discussion

- SAT-Based Methods are Effective
  - Bit blasting is only way to capture full set of operations
  - SAT solvers are good & getting better
    - On many UCLID benchmarks, have been getting 2X or better speedup each year since 2000 just from advances in SAT! (see earlier slide)

- Abstraction / Refinement Allows Better Scaling
  - Take advantage of cases where formula easily satisfied or disproven
Current Techniques with Sample Tools

- **Proof-based abstraction-refinement** – UCLID [Bryant et al., TACAS ’07]
- **Solver for linear modular arithmetic** to simplify the formula – STP [Ganesh & Dill, CAV’07]
- **Counterexample-guided abstraction-refinement, layered approach, rewriting** – MathSAT [Bruttomesso et al, CAV’07]
- **Automatic parameter tuning** – Spear [Hutter et al., FMCAD ’07]
STP’s approach

Input Formula $\phi$

- **Substitution**
- **Simplification**

**Linear equality solving**

**Bit-Blast**

**Refine Array Axioms**

**SAT Solving**

[ SAT / UNSAT ]

Simplify as much as possible then Bit-Blast. Add array axioms on demand.

[Ganesh & Dill, CAV ’07]
STP’s Linear Solver

- Critical step: greatly reduces number of variables and constraints in final SAT problem

- Solver for linear equalities mod power of 2
  - Solve system \( A \mathbf{x} = \mathbf{b} \pmod{2^k} \), online
  - Similar to earlier work by Huang & Cheng, TCAD’01

- General idea:
  1. Solve for a variable with odd coefficient using multiplicative inverse of coefficient, substitute it out of other equations
  2. If no odd coefficient, divide equation by power of 2 and solve for bit-extracted-part of a variable
Example

\[ 3x + 4y + 2z = 0 \]
\[ 2x + 2y = -2 \]
\[ 2x + 4y + 2z = 0 \]

\[ x, y, z \text{ are all 3 bits wide – solve mod 8} \]

Steps:

1. Pick an equation that’s solvable
   \[ \sum_i a_i x_i = c_i \pmod{2^b} \] solvable iff \( \gcd\{a_1, a_2, \ldots, a_n, 2^b\} \) divides \( c_i \)

2. If it has an odd coefficient \( a_i \), express \( x_i \) in terms of the others
   - Multiply throughout by multiplicative inverse of \( c_i \)

3. Substitute \( x_i \) out of all other equations
Example (contd.)

\[3x + 4y + 2z = 0\]
\[2x + 2y = -2\]
\[2x + 4y + 2z = 0\]

Solvable, has odd coefficient
Multiply throughout by \(3^{-1} \mod 8 = 3\)

solve for \(x\)
\[x = 4y + 6z\]
\[2x + 2y = -2\]
\[2x + 4y + 2z = 0\]

eliminate \(x\)
\[2y + 4z = -2\]
\[4y + 6z = 0\]

No odd coefficient!
Divide by 2, solve for variables expressing lower two bits (mod 4)

Final solved system:
\[x = 4y + 6z\]
\[y[1:0] = 2z[1:0] + 3\]
\[z[1:0] = 2\]

Note this technique solves 1 equation at a time \(\rightarrow\) online
Final Solutions

- Original variables: 3-bit unsigned integers
  \[ x: [x_2 \ x_1 \ x_0] \quad y: [y_2 \ y_1 \ y_0] \quad z: [z_2 \ z_1 \ z_0] \]

- Solutions

\[
\begin{align*}
y & = 3 \mod 4 \\
z & = 2 \mod 4
\end{align*}
\]

- In bit-vector form:
  \[ y: [y_2 \ 1 \ 1] \quad z: [z_2 \ 1 \ 0] \]

- Back Substitution to solve for \( x \)

\[
\begin{align*}
x & = 4y + 6z \mod 8 \\
x & = 0 \mod 8
\end{align*}
\]

- Constrained variables
  \[ x: [0 \ 0 \ 0] \]
Current Techniques with Sample Tools

- *Proof-based abstraction-refinement* – UCLID [Bryant et al., TACAS ’07]
- *Solver for linear modular arithmetic* to simplify the formula – STP [Ganesh & Dill, CAV’07]
- *Counterexample-guided abstraction-refinement, layered approach, rewriting* – MathSAT [Bruttomesso et al., CAV’07]
- *Automatic parameter tuning* – Spear [Hutter et al., FMCAD ’07]
The MathSAT Approach

[Bruttomesso et al., CAV ’07]

- **Pre-processing simplifications**
  - Propagation of unconstrained (fanout = 1) variables
  - Transforming term ITEs to Boolean ITEs
  - Constant propagation
  - Propagating extraction operators through concatenation and bit-wise operators

- **Counterexample guided abstraction-refinement** loop (CEGAR)
  - SAT solver communicates with theory solver (lazy SMT)

- **Layered approach** for theory solver
  - First invoke EUF solver
  - Then use bit-vector rewrite rules
  - Finally use solver based on SAT + Integer linear arith.
Concluding Points

- Resurgence of interest and results in word-level solvers (bit-vector decision procedures)

- Major ideas:
  - Abstraction-Refinement
    - Proof-based
    - CEGAR
  - Word-level Simplifications
    - Linear solver mod power of 2
    - Rewrite rules for bit-vector operators
  - Layered approach (MathSAT) / Function abstraction (UCLID)

- Next step: explore how best to combine term-level and word-level modeling
Outline

- Term-Level Modeling
- Bit-Vector Decision Procedures
- Word-Level Predicate Abstraction
- Interpolation
Word-level Circuit Models

Need to extract \textit{word-level formula} for
1. Initial state
2. Transition relation
3. Property
The SLAM Story

- Microsoft blames most Windows crashes on *third party device drivers*
- The Windows device driver API is quite complicated
- Low-level C code

- **SLAM**: Formal tool to automatically check device drivers for certain errors
- Shipped with Device Driver Development Kit (DDK)
Predicate Abstraction

- Circuits have too many state variables

  $\rightarrow$ State Space Explosion

- Graf/Saïdi 97: Predicate Abstraction
  
- Idea: Only keep track of *predicates* on data

  $p_1(s), \ldots, p_n(s)$

- Abstraction function:

  $h(s) = (p_1(s), p_2(s), \ldots, p_n(s))$
Predicate Abstraction

Concrete States:

Predicates:

\[ p_1(s) = (s.x > s.y) \]
\[ p_2(s) = (s.y = 0) \]

Abstract transitions?
Under- vs. Overapproximation

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

- Existential abstraction:

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

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

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

  \[\Rightarrow \text{Preserves safety properties}\]
Predicate Abstraction

Abstract Transitions:

\[ p_1, p_2 \]

\[ \neg p_1, p_2 \]

\[ p_1, \neg p_2 \]

\[ \neg p_1, \neg p_2 \]

Property holds. Ok.

Assertion:

\[ p_1 \lor \neg p_2 \iff (s.x > s.y) \lor (s.y \neq 0) \]
Predicate Abstraction

Abstract Transitions:

Assertion:

\[ p_1 \iff (s.x > s.y) \]

This trace is spurious!
Predicate Abstraction

Abstract Transitions:

Assertion:  
\[ p_1 \iff (s.x > s.y) \]

New Predicates:  
\[ p_1(s) = (s.x > s.y) \]
\[ p_2(s) = (s.x = 2) \]
Predicate Abstraction for Circuitry

- 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!
Existential Abstraction

Predicates

- $p_1$ $\iff i = 1$
- $p_2$ $\iff i = 2$
- $p_3$ $\iff \text{even}(i)$

Transition Relation

- $i \leq i + 1$

Current Abstract State

<table>
<thead>
<tr>
<th>$p_1$</th>
<th>$p_2$</th>
<th>$p_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Next Abstract State

<table>
<thead>
<tr>
<th>$p'_1$</th>
<th>$p'_2$</th>
<th>$p'_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Formula

- $i' = i + 1$

Query

- $i \neq 1 \land i \neq 2 \land \overline{\text{even}(i)} \land$
- $i' = i + 1 \land$
- $i' \neq 1 \land i' \neq 2 \land \overline{\text{even}(i')}$
Existential Abstraction

Predicates

\[ p_1 \iff i = 1 \]
\[ p_2 \iff i = 2 \]
\[ p_3 \iff \text{even}(i) \]

Transition Relation

\[ i' = i + 1 \]

Formula

\[ p' = p \]

Query

\[ i' \neq 1 \land i' \neq 2 \land \text{even}(i) \land \\
\quad i' = i + 1 \land \\
\quad i' \neq 1 \land i' \neq 2 \land \text{even}(i') \]

Current Abstract State

<table>
<thead>
<tr>
<th>( p_1 )</th>
<th>( p_2 )</th>
<th>( p_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Next Abstract State

<table>
<thead>
<tr>
<th>( p'_1 )</th>
<th>( p'_2 )</th>
<th>( p'_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Predicate Abstraction for Circuitry

- A precise existential abstraction can be way too slow
- Use an over-approximation instead
  - Fast to compute
  - But has additional transitions

- *Predicate partitioning*
  
  (DAC 2005, IEEE TCAD 2008)
Predicate Abstraction for Circuitry

- How do we get the predicates?
- Automatic abstraction refinement!

[Kurshan et al. ’93]
[Ball, Rajamani ’00]
[Clarke et al. ’00]
What I am Going to Show

- Apply predicate abstraction at RT-Level
  - Allows abstraction using word-level predicates
  - Example: $x < y - z$, $x = \{z, z\}$

- Use a SAT solver for computing abstraction
  - Semantics of bit-wise operators taken into account

- Obtaining suitable word level predicates
  - Syntactic weakest pre-conditions of Verilog statements
  - From word-level proofs
Abstraction Refinement Loop

Initial Abstraction

Verification

Actual Circuit

Abstract Model

Model Checker

Property holds

No error or bug found

Simulation successful

Bug found

Counterexample

Spurious counterexample

Refinement

[Clarke et al. ’00]

[Kurshan et al. ’93]

[Ball, Rajamani ’00]

[Clarke et al. ’00]
An example

module main (clk)
input clk;
reg [10:0] x, y;
initial x= 100, y = 200;
always @ (posedge clk) begin
  x <= y;
  y <= x;
end
endmodule

Assertion:
AG (x = 100 ∨ x = 200)
Abstraction Refinement Loop

- **Actual Circuit** → **Abstract Model**
  - **Initial Abstraction**
  - **Verification**
    - **Model Checker**
      - No error or bug found
      - Property holds
      - Counterexample
      - Simulation successful
      - Bug found
    - **Simulator**
      - Spurious counterexample

[References:
- Kurshan et al. '93
- Ball, Rajamani '00
- Clarke et al. '00]
module main (clk)
input clk;
reg [10:0] x, y;
initial x = 100, y = 200;
always @ (posedge clk)
begin
    x <= y;
    y <= x;
end
endmodule

Assertion:
AG (x = 100 ∨ x = 200)

Initial set of predicates:
{x = 100, x = 200}

Transition relation:
x' = y ∧ y' = x

Verilog program
Computing Most Precise Abstraction

Current state

\[<x = 100, \ x = 200>\]

Transition Relation

\[x' := y\]
\[y' := x\]

Next state

\[<x' = 100, \ x' = 200>\]

\[\exists x, y, x', y' : \]
\[(p_1 \iff x = 100) \land (p_2 \iff x = 200) \land\]
\[x' = y \land y' = x \land\]
\[(p_1' \iff x' = 100) \land (p_2' \iff x' = 200)\]

Computing abstract transitions
Obtain transitions

Equation passed to the SAT solver

\[(p_1 \iff x = 100) \land (p_2 \iff x = 200) \land x' = y \land y' = x \land (p'_1 \iff x' = 100) \land (p'_2 \iff x' = 200)\]

Computing abstract transitions

... and so on ...
Abstract Model

Verilog program

module main (clk)
  input clk;
  reg [10:0] x, y;
  
  initial x= 100, y= 200;

  always @ (posedge clk)
  begin
    x <= y;
    y <= x;
  end
endmodule

Assertion:
AG (x = 100 ∨ x = 200)

Initial set of predicates:
{x = 100, x = 200}

Initial state

Failure state
Abstraction Refinement Loop

Actual Circuit → Abstract Model → Model Checker

Initial Abstraction → Verification

Verification:
- No error or bug found
- Property holds
- Counterexample
- Simulation successful
- Bug found

Refinement:
- Spurious counterexample

Abstraction refinement

[Kurshan et al. ’93]
[Ball, Rajamani ’00]
[Clarke et al. ’00]
module main (clk)
input clk;
reg [10:0] x, y;

initial x= 100, y= 200;

always @ (posedge clk)
begin
  x <= y;
  y <= x;
end
endmodule

Verilog program
Model checking

module main (clk)
input clk;
reg [10:0] x, y;

initial x= 100, y= 200;

always @ (posedge clk)
begin
  x <= y;
  y <= x;
end
endmodule

Verilog program
Abstraction Refinement Loop

Actual Circuit → Abstract Model → Model Checker

Initial Abstraction → Verification

Simulation successful → No error or bug found

Property holds

Counterexample

Refinement

Spurious counterexample

Simulation successful → Bug found

[Clarke et al. ‘00]

[Kurshan et al. ‘93]

[Ball, Rajamani ’00]

[Clarke et al. ’00]
Simulation

Verilog program

module main (clk)
input clk;
reg [10:0] x, y;
initial x= 100, y= 200;
always @ (posedge clk)
begin
  x <= y;
  y <= x;
end
endmodule

Counterexample is spurious

Abstract counterexample

Initial state

Failure state

<x = 100, x = 200>

1,0 → 0,0

Counterexample is spurious
Abstraction Refinement Loop

[Abstraction Refinement Loop Diagram]

Actual Circuit ➞ Abstract Model ➞ Model Checker ➞ Simulator

Initial Abstraction ➞ Verification

- No error or bug found
- Property holds
- Simulation successful
- Bug found

Refinement

- Abstraction refinement
- Spurious counterexample

References:
- [Kurshan et al. ’93]
- [Ball, Rajamani ’00]
- [Clarke et al. ’00]
Refinement

Let length of spurious counterexample be \( k \)

Take \textbf{weakest pre-condition of property} for \( k \) steps \textbf{with respect to transition function}
Refinement

Property

$AG (x = 100 \lor x = 200)$

New predicates

$y = 100, y = 200$

(y = 100 \lor y = 200)

$x' := y$

$y' := x$

$(x' = 100 \lor x' = 200)$

Holds after one step

weakest precondition

spurious counterexample

length = 1

Property

$(y = 100 \lor y = 200)$
Assertion:
\( AG (x = 100 \text{ or } x = 200) \)

Updated set of predicates:
\{x = 100, x = 200, y=100, y=200\}

New abstraction

Initial state

Model check
Model checking

Verilog program

module main (clk)
input clk;
reg [10:0] x, y;

initial x= 100, y= 200;

always @ (posedge clk)
begin
  x <= y;
  y <= x;
end
endmodule

Assertion:
AG (x = 100 or x = 200)

Updated set of predicates:
{x = 100, x = 200, y=100, y=200}

New abstraction

Initial state

Property holds!
Result

Verilog program

module main (clk)
input clk;
reg [10:0] x, y;

initial x= 100, y= 200;

always @ (posedge clk)
begin
  x <= y;
  y <= x;
end
endmodule

Assertion
AG (x = 100 or x = 200)

Property holds!
Making it work in practice

- Predicate Abstraction Computation
  - Handling a large number of predicates

- Refinement
  1. Good predicates but inexact abstraction (due to over approximation of most precise abstraction)
  2. Insufficient predicates
Other Approaches

- SAT-Based Predicate Abstraction [Wang et al.]
  - Works at netlist level
  - Refinement introduces bit-level predicates

- Vapor tool [Andraus et al.]
  - Works on RT-level designs
  - Abstraction to CLU models
    (equality of terms, uninterpreted functions, predicates)

- Lots of other related work
Benchmarks

- USB 2.0 function core from opencores.org
- 4000 lines of RTL Verilog

Checked three properties:
1. DMA module simulates state machine on left. (USB1)
2. Every state transitions to IDLE state when abort signal is on. This property fails. (USB2)
3. Every state expect MEM_WR2 transitions to IDLE state when abort signal is on. (USB3)
Benchmarks

- Ethernet MAC from opencores.org
- 5000 lines of RTL Verilog

Checked three properties:

1. Transmit module simulates state machine on left. (ETH0)
2. Checks transitions out of state BackOff (ETH1)
3. Checks transitions out of state Jam (ETH2)
Other Benchmarks

- ICRAM (Instruction Cache RAM)
  - ICache from SUN PicoJava II
  - Contains a memory whose size can be varied
  - Two properties related to writing to and reading from memory

- AR benchmarks
  - Simple arithmetic benchmark with two registers
  - Can vary the size of registers
### Experimental results

A dash "-" indicates a timeout of 2 hours.

A star "*" indicates model checker terminated due to large number of BDD variables.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Latches</th>
<th>Time</th>
<th>Abs</th>
<th>MC</th>
<th>Ref</th>
<th>P</th>
<th>I</th>
<th>CSMV Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>USB1</td>
<td>545</td>
<td>42</td>
<td>1</td>
<td>2</td>
<td>29</td>
<td>17</td>
<td>62/0</td>
<td>149</td>
</tr>
<tr>
<td>USB2</td>
<td>545</td>
<td>599</td>
<td>47</td>
<td>147</td>
<td>386</td>
<td>116</td>
<td>146/22</td>
<td>1349</td>
</tr>
<tr>
<td>USB3</td>
<td>545</td>
<td>446</td>
<td>46</td>
<td>73</td>
<td>317</td>
<td>114</td>
<td>123/20</td>
<td>2594</td>
</tr>
<tr>
<td>ETH0</td>
<td>359</td>
<td>44</td>
<td>2</td>
<td>3</td>
<td>30</td>
<td>21</td>
<td>55/0</td>
<td>-</td>
</tr>
<tr>
<td>ETH1</td>
<td>359</td>
<td>127</td>
<td>8</td>
<td>8</td>
<td>102</td>
<td>93</td>
<td>49/2</td>
<td>-</td>
</tr>
<tr>
<td>ETH2</td>
<td>359</td>
<td>161</td>
<td>8</td>
<td>16</td>
<td>127</td>
<td>94</td>
<td>109/2</td>
<td>21</td>
</tr>
<tr>
<td>M2KB</td>
<td>16427</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>1/0</td>
<td>28</td>
</tr>
<tr>
<td>M8KB</td>
<td>65694</td>
<td>28</td>
<td>0</td>
<td>0</td>
<td>27</td>
<td>3</td>
<td>1/0</td>
<td>*</td>
</tr>
<tr>
<td>M16KB</td>
<td>131117</td>
<td>34</td>
<td>0</td>
<td>0</td>
<td>34</td>
<td>3</td>
<td>1/0</td>
<td>*</td>
</tr>
<tr>
<td>N2KB</td>
<td>16427</td>
<td>93</td>
<td>0</td>
<td>0</td>
<td>93</td>
<td>11</td>
<td>9/0</td>
<td>*</td>
</tr>
<tr>
<td>N8KB</td>
<td>65694</td>
<td>490</td>
<td>0</td>
<td>0</td>
<td>490</td>
<td>11</td>
<td>9/0</td>
<td>*</td>
</tr>
<tr>
<td>N16KB</td>
<td>131117</td>
<td>790</td>
<td>0</td>
<td>0</td>
<td>789</td>
<td>11</td>
<td>9/0</td>
<td>*</td>
</tr>
<tr>
<td>AR200</td>
<td>400</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>3/2</td>
<td>672</td>
</tr>
<tr>
<td>AR3000</td>
<td>6000</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>12</td>
<td>3</td>
<td>3/2</td>
<td>-</td>
</tr>
<tr>
<td>AR4000</td>
<td>8000</td>
<td>17</td>
<td>0</td>
<td>0</td>
<td>16</td>
<td>3</td>
<td>3/2</td>
<td>-</td>
</tr>
</tbody>
</table>

### Hardware

- **USB
- Ethernet
- MAC
- ICRAM
- AR**
Predicates from Word-Level Proofs

- Predicates from weakest-preconditions not always ideal
- Idea: use a word-level proof-engine to get better ones (DATE 2007)
Example: Word-level Proof

reg [6:0] c;
initial c=0;

always @(posedge clk)
    if(c!=64 && issue && !retire)
        c=c+1;
    else if(c!=0 && !issue && retire)
        c=c-1;

ROB

head

tail
Example: Word-level Proof (I)

```verilog
reg [6:0] c;
initial c=0;

always @(posedge clk)
    if(c!=64 && issue && !retire)
        c=c+1;
    else if(c!=0 && !issue && retire)
        c=c-1;
```

\[
s'.c = \begin{cases} 
    s.c + 1 & : \; s.c \neq 64 \land s.\text{issue} \land \neg s.\text{retire} \\
    s.c - 1 & : \; s.c \neq 0 \land \neg s.\text{issue} \land s.\text{retire} \\
    s.c & : \; \text{otherwise}
\end{cases}
\]
Example: Word-level Proof (II)

\[ s'.c = \begin{cases} 
  s.c + 1 & : \ s.c \neq 64 \land s.issue \land \neg s.retire \\
  s.c - 1 & : \ s.c \neq 0 \land \neg s.issue \land s.retire \\
  s.c & : \ otherwise 
\end{cases} \]

\[(s.c \neq 64 \land s.issue \land \neg s.retire) \rightarrow s'.c = s.c + 1 \]
\[\land\ (s.c \neq 0 \land \neg s.issue \land s.retire) \rightarrow s'.c = s.c - 1 \]
\[\land\ (\neg(s.c \neq 64 \land s.issue \land \neg s.retire) \land \neg(s.c \neq 0 \land \neg s.issue \land s.retire)) \rightarrow s'.c = s.c \]
Example: Word-level Proof (III)

Property: \( c \neq 127 \)

+ predicate \( c = 127 \)

Spurious counterexample of length 2
Example: Word-level Proof (IV)

- Refinement with weakest precondition produces predicate $c = 126$

- Requires 64 refinement iterations

- Instead: Give simulation instance to word-level prover and perform BFS proof-search

- Generates predicate $c \leq 64$, which shows the property
Summary: Predicate Abstraction

- Verification at register level without going to netlists
- Predicate abstraction using word-level predicates
- Handling large no. of predicates (predicate clustering)
- Weakest pre-conditions or proofs for obtaining new predicates

VCEGAR

http://www.cs.cmu.edu/~modelcheck/vcegar
Questions?