19. Extending the Capacity of Formal Engines

Jacob Abraham

Department of Electrical and Computer Engineering
The University of Texas at Austin
Verification of Digital Systems
Spring 2018
March 29, 2018

Bottlenecks for Fully Automated Formal Verification of SoCs

Limited tool capacity
- Enormous state spaces of real designs are beyond the capabilities of formal verification tools
- Need innovative techniques to deal with large designs

Possible approaches
- Partitioning the design: divide and conquer
- Exploiting assume-guarantee methods
- Approximation techniques
- Abstracting large designs
Combining Simulation and Formal Verification

Properties involving “deep” states

- Formal tools are unable to reach states that require very large numbers of clocks from the initial state
- Simulation can be used to reach intermediate states
  - Genetic algorithms and related search techniques can be used to get “close” to the desired state
- Using known reachable initial states as the starting state for formal tools may allow properties involving the deep states to be proved

Related problem that must be solved

- Useful definition of “close”
  - Difficult for highly nonlinear digital systems – a single state bit change for one state (Hamming distance of 1) can mean a large number of steps to reach the resulting state

Under Approximation Techniques for Test Generation (Prabhu, 2013)

Techniques

- Bit-width reduction
- Data-path operator approximation
### Bit-Width Reduction

**Approach**
- Restrict domain of variables
- These restrictions may speed up SMT solver
- Approximation $F_{UA}$ of original formula $F$ may have fewer behaviors
- If $F_{UA}$ is satisfiable then so is $F$
- If $F_{UA}$ is unsatisfiable then ease restrictions
- Analyze unsat core to find out which restrictions to ease

**Restriction of bit-vector variables**
- $n$ least significant bits of a bit-vector of width $w$ unconstrained
- $n$ is called the effective bit-width
- The upper $(w - n)$ bits are sign-extended, zero-extended or one-extended

---

### Bit-Width Refinement Process

1. **Faulty RTL Model**
2. **RTL Model Checker**
   - Observability Constraints
   - Dependency Graph
3. **Add restriction clauses $C$**
4. **Refine restriction clauses**
5. **SAT?**
   - Yes
   - Test
   - Yes
   - C used in unsat core?
     - Yes
     - Untestable
     - No
     - No
     - Untestable
   - No
     - Untestable
Refinement Strategies

Local Refinement
- Refine only those bit-vector variables identified by the unsat clause
- On demand refinement
- Disadvantage is that a lot of refinements might be needed for a given propagation path

Dependency Graph Based Refinement
- Refine all the data dependent variables in the fan out cone of the variables in the unsat clause
- More proactive than local refinement
- Refines all the variables in a possible propagation path

Data Path Operator Approximation

Approach
- Hard operators are replaced by partially interpreted functions
- Use rewrite rules to approximate hard operators
- Multiplication
  - \( out = x \times y \) is replaced by \((x = 0 \implies out = 0) \&\& (x = 1 \implies out = y) \&\& (y = 0 \implies out = 0) \&\& (y = 1 \implies out = x)\)
- Addition
  - \( out = x + y \) is replaced by \((x = 0 \implies out = y) \&\& (y = 0 \implies out = x)\)
- This creates an under-approximation to solve several hard bit-vector formulas
Results of Bit-Width Reduction

<table>
<thead>
<tr>
<th>Module</th>
<th>No Abstraction (secs)</th>
<th>Local Refinement (secs)</th>
<th>Graph based Refinement (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>if</td>
<td>23.41</td>
<td>53.03</td>
<td>16.27</td>
</tr>
<tr>
<td>ctrl</td>
<td>21.16</td>
<td>38.74</td>
<td>12.49</td>
</tr>
<tr>
<td>opmuxes</td>
<td>19.33</td>
<td>42.61</td>
<td>10.18</td>
</tr>
<tr>
<td>sprs</td>
<td>18.39</td>
<td>26.31</td>
<td>8.72</td>
</tr>
<tr>
<td>freeze</td>
<td>10.48</td>
<td>32.10</td>
<td>7.15</td>
</tr>
<tr>
<td>rf</td>
<td>22.85</td>
<td>51.42</td>
<td>18.16</td>
</tr>
<tr>
<td>except</td>
<td>38.14</td>
<td>64.75</td>
<td>25.28</td>
</tr>
<tr>
<td>Overall</td>
<td>24.23</td>
<td>35.18</td>
<td>16.82</td>
</tr>
</tbody>
</table>

- The average run time is higher for local refinement – too fine grained
- Dependency graph based refinement strategy gives better results

Results of Data-Path Operator Approximation

- Operators reduced: `bvmul`, `bvadd`, `bvsb`, `bvshl`, `bvshr`, `bvxor`, `bvor`
- Only operators in module `or1200.alu` were reduced

<table>
<thead>
<tr>
<th>Module</th>
<th>No Abstraction (secs)</th>
<th>Operator approximation (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>if</td>
<td>23.41</td>
<td>6.79</td>
</tr>
<tr>
<td>ctrl</td>
<td>21.16</td>
<td>8.42</td>
</tr>
<tr>
<td>opmuxes</td>
<td>19.33</td>
<td>7.84</td>
</tr>
<tr>
<td>sprs</td>
<td>18.39</td>
<td>8.19</td>
</tr>
<tr>
<td>freeze</td>
<td>10.48</td>
<td>5.91</td>
</tr>
<tr>
<td>rf</td>
<td>22.85</td>
<td>10.72</td>
</tr>
<tr>
<td>except</td>
<td>38.14</td>
<td>12.29</td>
</tr>
<tr>
<td>Overall</td>
<td>24.23</td>
<td>8.86</td>
</tr>
</tbody>
</table>

- Reducing the operators in control logic is counterproductive
- Difficult to automatically identify operators in data-path logic
Slicing – A Powerful Automatable Abstraction Technique

**Program slicing** was introduced by Weiser (1981) to the software community.

Originally used to help in debugging programs.

Other applications have been testing (reduce cost of regression testing after modifications (only run those tests that needed), parallelization, integration (merging two programs A and B that both resulted from modifications to base).

**Program slicing**

- A decomposition technique that extracts statements relevant to a particular computation from a program.
- Based on a “Slicing Criterion” \(<s, v>\)
- Provides a useful abstraction of a program (design), obtained by analyzing the program text through static analysis.

Acknowledgments: B. K. Dash, S. Vasudevan, V. Vedula

Slicing Process

- A slice is constructed by deleting those parts of a program which is irrelevant to the program with respect to the chosen set of variables at the chosen point.
- A slice is taken from a program with the slicing criterion \(<s, v>\), which specifies a location (statement s) and a variable (v).
- Intermediate Representation of programs for slicing
  - Control Flow Graph (CFG) (Solve data flow equations)
  - Program Dependence Graph (PDG) (Slice is computed as graph reachability problem)
Example Program

1 main()
2 {
3    int i, sum;
4    sum = 0;
5    i = 1;
6    while(i <= 10)
7    {
8        sum = sum + 1;
9        ++ i;
10    }
11    Cout << sum;
12    Cout << i;
13 }

Example: Slicing w.r.t. <12, i>

1 main()
2 {
3    int i, sum;
4    sum = 0;
5    i = 1;
6    while(i <= 10)
7    {
8        sum = sum + 1;
9        ++ i;
10    }
11    Cout << sum;
12    Cout << i;
13 }
Slicing Classifications

- Types of slices
  - Static
  - Dynamic

- Direction of slicing
  - Backward
  - Forward

- Executability of slice
  - Executable
  - Closure

- Levels of slices
  - Intraprocedural
  - Interprocedural

Weiser’s classic slices were “Executable Backward Static Slices”
Complexity of Slicing

If the slice is to be exact, i.e., for variables of interest, the behavior of the slice mirrors the behavior of the original program, and vice-versa, the problem is undecidable.

Equivalent to the halting problem of a Turing machine.

For finite systems (such as hardware designs), the undecidability is not an issue.

The problem may still be very complex, but we can find a conservative (overapproximation) slice, so that all the relevant behavior with respect to the set of variables is represented in the slice.

In hardware verification, the set of variables for slicing would be the variables in the property to be verified.

In this case, the conservative static slice could be used to reduce the search space for verification.

Slicing for Hardware Designs: Example

![Diagram of slicing in hardware designs]

Department of Electrical and Computer Engineering, The University of Texas at Austin
J. A. Abraham, March 29, 2018
Slicing for Hardware Designs

- Have to deal with concurrency, unlike conventional software
- Multiple processes related to a module or set of signals of interest
- Subset of processes would be of interest
- Derive constraint slices of individual processes in environment
- Iterative steps to derive slices

Basic Theory

- Lemma: Union of constraint slices of relevant modules provides the abstracted environment

Slicing Vs Cone of Influence (COI)

- Slicing is a behavioral abstraction; leverages the semantics of the programming language (HDL in this case)
- COI is a structural abstraction; works on the logic representation as a kriple structure
- Slicing enables the design analysis at the RT-level
- COI supports analysis at a later stage in the design cycle
- Slicing can handle significantly larger design sizes
- COI is limited in capacity
- Slicing tools are tied to the programming language (Verilog, VHDL etc.)
- COI is generic as it works on the netlist representation
Review: Symbolic Trajectory Evaluation

Symbolic simulation based model checking
Limited form of temporal logic for specification
Properties specified over bounded length of sequences called trajectories
Assertions are of the form $[A \Rightarrow C]$, where
- $A$ is the antecedent (input constraints)
- $C$ is the consequent (expected outputs)
- $A$ and $C$ are trajectory formulas

Checks if every sequence of states that satisfies $A$ also satisfies $C$

Widely used in the industry for data-path verification

Industrial Test Case

Floating-point adder in Pentium-4 selected as test case
Constraint slices derived on an embedded module
Module+constraint slices provided to STE engine
STE assertions (divided into 32 sets/case) verified
Capacity and performance compared with original (un-sliced) design
Reduction in Peak BDD Size – I

Node size (in Millions) Vs Assertion run

Reduction in Peak BDD Size – II

Node size (in Millions) Vs Assertion run
## Performance Improvement

<table>
<thead>
<tr>
<th>Case #</th>
<th>Original (secs)</th>
<th>Slice (secs)</th>
<th>Speed-up</th>
<th>Case #</th>
<th>Original (secs)</th>
<th>Slice (secs)</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>2624.6</td>
<td>674.87</td>
<td>3.9</td>
<td>17</td>
<td>1069.67</td>
<td>256.2</td>
<td>4.2</td>
</tr>
<tr>
<td>02</td>
<td>2176.27</td>
<td>465.11</td>
<td>4.7</td>
<td>18</td>
<td>1431.31</td>
<td>310.86</td>
<td>4.6</td>
</tr>
<tr>
<td>03</td>
<td>2031.71</td>
<td>495.04</td>
<td>4.1</td>
<td>19</td>
<td>1886.65</td>
<td>367.93</td>
<td>5.1</td>
</tr>
<tr>
<td>04</td>
<td>2125.78</td>
<td>536.67</td>
<td>4.0</td>
<td>20</td>
<td>2404.24</td>
<td>490.95</td>
<td>4.9</td>
</tr>
<tr>
<td>05</td>
<td>2346.91</td>
<td>585.96</td>
<td>4.0</td>
<td>21</td>
<td>1685.77</td>
<td>324.51</td>
<td>5.2</td>
</tr>
<tr>
<td>06</td>
<td>2521.91</td>
<td>634.09</td>
<td>4.0</td>
<td>22</td>
<td>1408.51</td>
<td>228.47</td>
<td>6.2</td>
</tr>
<tr>
<td>07*</td>
<td>3835.92</td>
<td>1317.03</td>
<td>2.9</td>
<td>23</td>
<td>1632.04</td>
<td>281.47</td>
<td>5.8</td>
</tr>
<tr>
<td>08*</td>
<td>4585.92</td>
<td>982.79</td>
<td>4.7</td>
<td>24</td>
<td>2021.0</td>
<td>365.33</td>
<td>5.5</td>
</tr>
<tr>
<td>09</td>
<td>2406.41</td>
<td>633.27</td>
<td>3.8</td>
<td>25</td>
<td>2418.92</td>
<td>458.89</td>
<td>5.3</td>
</tr>
<tr>
<td>10</td>
<td>2173.76</td>
<td>575.23</td>
<td>3.8</td>
<td>26</td>
<td>2699.27</td>
<td>534.07</td>
<td>5.1</td>
</tr>
<tr>
<td>11</td>
<td>2054.84</td>
<td>527.37</td>
<td>3.9</td>
<td>27</td>
<td>3107.4</td>
<td>704.64</td>
<td>4.4</td>
</tr>
<tr>
<td>12</td>
<td>1864.63</td>
<td>490.22</td>
<td>3.8</td>
<td>28</td>
<td>692.81</td>
<td>209.39</td>
<td>3.3</td>
</tr>
<tr>
<td>13</td>
<td>2041.06</td>
<td>453.69</td>
<td>4.5</td>
<td>29</td>
<td>652.34</td>
<td>188.29</td>
<td>3.5</td>
</tr>
<tr>
<td>14</td>
<td>1735.96</td>
<td>483.96</td>
<td>3.6</td>
<td>30</td>
<td>922.51</td>
<td>239.17</td>
<td>3.9</td>
</tr>
<tr>
<td>15</td>
<td>587.33</td>
<td>181.48</td>
<td>3.2</td>
<td>31</td>
<td>1242.07</td>
<td>284.3</td>
<td>4.4</td>
</tr>
<tr>
<td>16</td>
<td>785.84</td>
<td>211.38</td>
<td>3.7</td>
<td>32</td>
<td>1658.46</td>
<td>341.88</td>
<td>4.9</td>
</tr>
</tbody>
</table>

*indicates failing assertion

## Antecedent Conditioned Slicing

$h: G (A => C)$

**Antecedent**

$X^n C$

**Consequent**

- if (A) $C = 1$
- else $C = 0$

**Semantic analysis**

- if (A) $C = 1$
- else $C = 0$

**Static slicing on A, C**

**Antecedent conditioned slicing on <A= true>, A, C**

**Variable dependency analysis**

Department of Electrical and Computer Engineering, The University of Texas at Austin

J. A. Abraham, March 29, 2018
Example of Antecedent Conditioned Slicing – I

always @ (clk) begin
  case(insn)
    f_add: dec = d_add; d_add: ex = e_add; e_add: res = a+b;
    f_sub: dec = d_sub; d_sub: ex = e_sub; e_sub: res = a-b;
    f_and: dec = d_and; d_and: ex = e_and; e_and: res = a\&b;
    f_or: dec = d_or; d_or: ex = e_or; e_or: res = a|b;
  endcase
end

h = [G((insn == f_add) ⇒ XX(res == a+b))]

Example of Antecedent Conditioned Slicing – II

Diagram illustrating the antecedent conditioned slicing.
19. Extending the Capacity of Formal Engines

Example of Antecedent Conditioned Slicing – III

Example of Antecedent Conditioned Slicing – IV
Example of Antecedent Conditioned Slicing – V

Example of Antecedent Conditioned Slicing – VI
Example of Antecedent Conditioned Slicing – VII

Example of Antecedent Conditioned Slicing – VIII
Example of Antecedent Conditioned Slicing – IX

```
insn?

f_add

dec = d_add

dec?
d_add

ex = e_add

ex?
e_add

res = a + b
```

Example of Antecedent Conditioned Slicing – X

```
always @ (clk) begin
    case(insn)
        f_add: dec = d_add;
        endcase
    end

always @ (clk) begin
    case(dec)
        d_add: ex = e_add;
        endcase
    end

always @ (clk) begin
    case(ex)
        e_add: res = a+b;
        endcase
    end

Single instruction behavior for f_add instruction

h = [G((insn == f_add) ⇒ XX(res == a+b))]
Checking the Truth of the Antecedent

Symbolic simulation of all nodes in each process
Expression computation over all processes in the program
Handles global predicates
Symbolic simulation of the antecedent
Looking forward in time
Depends on $n$ in $(A \implies XnC)$
Decision procedure for checking truth of antecedent
Could be arbitrarily hard
Path traversal of all processes
Pruning non-retained nodes

Worst case: retain all nodes

Complexity of Antecedent Conditioned Slicing
Correctness of Antecedent Conditioned Slicing

Theorem
An LTL formula $h$ of the type, where $h$ is one of

- $\Box (a \implies c)$
- $\Box (a \implies X^n c)$
- $\Box (a \implies F^k c)$

holds on the original program iff it holds on the antecedent conditioned slice.

Proof Intuition
- For a Kripke structure of the slice, all states satisfy $a \implies c$.
- These include states of the original Kripke structure that satisfy $a$.
- Thus all states of the original that satisfy $a$ must satisfy $h$.
- All states of the original that satisfy $\neg a$, satisfy $a \implies c$ vacuously.

Experiments with Antecedent Conditioned Slicing

USB 2.0 Function Core
- Verilog implementation from www.opencores.org
- Properties from specification document
- Safety properties expressed in LTL ($\Box (a \implies c)$)
- Verification engine: Cadence-BMC (bound of 24-50 steps)
Example USB Properties

\[ G((\text{crc5err} \lor \neg (\text{match}) \implies \neg (\text{send_token})) \]

If a packet with a bad CRC5 is received, or there is an endpoint field mismatch, the token is ignored.

\[ G((\text{state} == \text{SPEED}_\text{NEG}_\text{FS}) \implies X((\text{mode hs}) \land (T_{1_{gt}}_{3.0ms}) \implies (\text{next state} == \text{RES_SUSPEND})) \]

If the machine is in the speed negotiation state, then in the next clock cycle, if it is in high speed mode for more than 3 ms, it will go to the suspend state.

\[ G((\text{state} == \text{RESUME}_\text{WAIT}) \land \neg (\text{idle cnt clr}) \implies F(\text{state} == \text{NORMAL})) \]

If the machine is waiting to resume operation and a counter is set, eventually (after 100 mS) it will return to normal operation.

Results on Temporal USB Properties

CPU seconds, on a 450 MHz dual UltraSPARC-II with 1 GB RAM
Verification of Pipelined Processors

- Specification state includes internal registers and memory contents
- Implementation state includes the above as well as the pipeline state
  - Many more bits to deal with in the implementation, and current tools may not be able to deal with the state explosion
- Key idea
  - Can reduce implementation state to specification state by flushing pipeline
  - Can flush pipeline with series of NOPs or stall cycles

Pipelined Correspondence (Burch and Dill, 1994)
Verification of Processors Using Antecedent Conditioned Slicing

Verification of single-instruction issue, multi-stage pipelined processors
- Properties are at the Instruction level (not for an internal block in the design)
- Antecedent conditioned slicing provides an automatic decomposition strategy
- Individual “instruction machines"
- Verified all the instructions of the OR1200 embedded processor (www.opencores.org)

Single Instruction Verification
- Obtain single instruction machine by antecedent conditioned slicing
  - Antecedent in the first cycle is the instruction word
  - The antecedents in the successive cycles are typically control signals whose values need to be specified for every cycle
- Property is $G(I \implies R)$ where
  - $I = i_1 X i_2 X i_3 \ldots X_n i_n$
    - $i_t$ represents the antecedent in pipeline stage $t$
  - $R$ is the result of $I$ in terms of its target register values
- The behavior of the instruction is thus expressed as an antecedent consequent pair

Model checking of instruction $I$

Department of Electrical and Computer Engineering, The University of Texas at Austin
J. A. Abraham, March 29, 2018
Single Instruction Verification

Lemma: Instructions should write back only to target register only on writeback stage

Interaction Between Instructions

Lemma: Instructions should write back only to target register only on writeback stage
Notion of Processor Correctness

Theorem
The instruction slices, when executed in the same sequence as the corresponding instructions in the original pipelined machine, will produce the same result as the original pipeline.

Results of OR1200 Verification

CPU seconds, 3GHz Pentium 4 processor with 1 GB RAM
SMV would not even compile the design without slicing

<table>
<thead>
<tr>
<th>Instruction Class</th>
<th>Instruction</th>
<th>SMV time (seconds)</th>
<th>Memory Usage (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSU</td>
<td>l.id</td>
<td>35.85</td>
<td>29104</td>
</tr>
<tr>
<td>LSU</td>
<td>l.lws</td>
<td>33.91</td>
<td>28873</td>
</tr>
<tr>
<td>LSU</td>
<td>l.sd</td>
<td>38.32</td>
<td>30941</td>
</tr>
<tr>
<td>SHF/ROT</td>
<td>l.sll</td>
<td>26.81</td>
<td>23771</td>
</tr>
<tr>
<td>SHF/ROT</td>
<td>l.srl</td>
<td>27.83</td>
<td>23771</td>
</tr>
<tr>
<td>SHF/ROT</td>
<td>l.ror</td>
<td>27.83</td>
<td>26919</td>
</tr>
<tr>
<td>SPRS</td>
<td>l.mfspr</td>
<td>226.97</td>
<td>50696</td>
</tr>
<tr>
<td>SPRS</td>
<td>l.mtspr</td>
<td>212.27</td>
<td>48627</td>
</tr>
</tbody>
</table>
Key References


S. Vasudevan, ”High Level Static Analysis of System Descriptions for Taming Verification Complexity”, PhD Thesis, The University of Texas at Austin, 2007
