Constraint-Based Test Generation

Vitaly Lagoon
(lagoon@cadence.com)
A Word About Cadence

• Cadence
  – One of the three major EDA (Electronics Design Automation) companies
  – Established in 1988, over 4000 employees
  – Wide variety of technologies and products in
    • Hardware design (inc. logic synthesis, power, timing, routing, etc.)
    • Hardware simulation
    • Formal and simulation-based verification
  – Never heard about us? You surely heard about our customers.
    • Intel, Cisco, TI, Canon, Phillips, Samsung, Nokia, …
Motivation

Chip complexity grows, and so does
• the likelihood of HW bugs
• the cost of HW bugs
• the need for verification
• investment in verification
The Cost of HW Bugs

- Intel FDIV, 1994, ~$475M
- Intel Sandy Bridge, 2011, ~$1B

Silicon Debug, Doug Josephson and Bob Gottlieb, (Paul Ryan)
Formal vs. Simulation

• Formal verification
  + *Proves* properties of a HW design
  + Substantial improvements in performance, capacity and scalability in the last few years
    ✓ Improvements in SAT solvers
    ✓ New approaches using abstraction and Solving Modulo Theories (SMT)
  ✗ Cannot verify (yet?) a full system, only individual units
  ✗ Verification environment cannot be reused for post-silicone testing

• Simulation-based verification
  + High capacity and scalability
  + Verification environment can be reused for post-silicone testing
  + Easy to use
  ✗ Experimentally verifies properties of a HW design

• ~80% of bugs in HW logics are still found through simulation
Verification by Simulation

Specman™

Stimuli Generation
IntelliGen

Signal Interface

DUT (HDL)

Data and Assertion Checkers

Coverage Monitor
Specman

- Cadence’s major test bench automation tool
- Being used in the biggest and most advanced verification environments
- Works with all HDL simulators
- Uses e verification language [Hollander,Morley,Noy ‘01]

http://en.wikipedia.org/wiki/Specman
IntelliGen

• Constraint solver / test generator of Specman
  – Generates randomized tests based on constraint models

• Variety of data types:
  – signed/unsigned numeric, Boolean, string, arrays, pointers

• Variety of constraints:
  – arithmetic, logic, bit-wise, soft constraints, global constraints on arrays

• Based on an FD-core
  – Also integrates a few variations of BDD and SAT

• Includes a visual constraint debugger
Random Test Generation in Specman/e

```c
struct CPU_instruct {
    opcode : [LDA, STA, ADD, SUB, JMP, JGE, JNE, STP](bits:4);
    operand : uint(bits:12);
    keep opcode == STA => operand > 1023;
};

struct CPU_test {
    program : list of CPU_instruction;
    sz : uint[10..20];
    keep program.size() == sz;
    keep program[sz-1].opcode == STP;
    keep for each (instr) in program {
        index<sz-1 => instr.opcode != STP;
        instr.opcode in [JMP, JGE, JNE] => instr.operand < sz;
    }
};
```

<table>
<thead>
<tr>
<th>item</th>
<th>type</th>
<th>opcode</th>
<th>operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>instruction</td>
<td>JMP</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>instruction</td>
<td>JMP</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>instruction</td>
<td>ADD</td>
<td>2301</td>
</tr>
<tr>
<td>3</td>
<td>instruction</td>
<td>JGE</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>instruction</td>
<td>SUB</td>
<td>312</td>
</tr>
<tr>
<td>5</td>
<td>instruction</td>
<td>LDA</td>
<td>2603</td>
</tr>
<tr>
<td>6</td>
<td>instruction</td>
<td>JMP</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>instruction</td>
<td>JGE</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>instruction</td>
<td>JNE</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>instruction</td>
<td>STA</td>
<td>3913</td>
</tr>
<tr>
<td>10</td>
<td>instruction</td>
<td>JMP</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>instruction</td>
<td>SUB</td>
<td>1783</td>
</tr>
<tr>
<td>12</td>
<td>instruction</td>
<td>LDA</td>
<td>2035</td>
</tr>
<tr>
<td>13</td>
<td>instruction</td>
<td>ADD</td>
<td>1310</td>
</tr>
<tr>
<td>14</td>
<td>instruction</td>
<td>JNE</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>instruction</td>
<td>JMP</td>
<td>12</td>
</tr>
<tr>
<td>16</td>
<td>instruction</td>
<td>LDA</td>
<td>3258</td>
</tr>
<tr>
<td>17</td>
<td>instruction</td>
<td>STP</td>
<td>3964</td>
</tr>
</tbody>
</table>

IntelliGen
IntelliGen.
The Requirements

- Powerful and flexible constraint language
  - Mixed integer and bitwise models
  - Mixed declarative and procedural code in models (function calls)
  - Non-scalar data: structures, arrays, strings
  - Soft constraints for modeling preferences
  - Directives for controlling randomness and distribution

- High scalability
  - Huge models (hundreds of thousands variables and constraints)
  - Solving the same problem many times (long runs)

- Find many random solutions
  - Try to meet distribution requirements
  - Be *random-stable* as much as possible

- Typical problems are not hard
  - Extensive search is usually not required
  - Backtracks typically indicate bad modeling
IntelliGen.
The Integrated Framework of Solvers

- There is no “best” solving technology.
  - A combined approach is required in practice
- BDD
  + Bit-level, fast, complete control over the distribution
  ✗ Capacity problems
- SAT
  + Fast, scalable
  ✗ Translation to CNF is expensive, limited control of randomness
- Finite-Domain solver
  + Word-level, fast, scalable, extendable
  ✗ Limited control over distribution
  ✗ Bad in proving UNSAT
- Local Search / SMT / ILP… ?
Solving Technologies: BDD

• Build the BDD once
• Generate solutions as random walks from the root to TRUE

+ Good performance when generating many solutions
+ Complete control over the distribution

× Limited capacity
× Very sensitive to the order of variables
× Infeasible for some types of constraints

\[
\langle a_3, a_2, a_1, a_0 \rangle \leq \langle b_3, b_2, b_1, b_0 \rangle
\]
Solving Technologies: SAT

• Convert the constraint model to a CNF
• Give it to a state-of-the-art SAT solver

+ High capacity
+ High performance
+ Improvements are free

× Randomization is tricky, no guarantee of distribution
× High bootstrap cost
× Loss of high-level context:
  – Role and association of bits
  – No GAC (loss of propagation) e.g., all-different
    [Bessiere, Katsirelos, Narodytska, Walsh, 09]

\[ (-a_3 \lor b_3) \land \\
(-a_3 \lor r_2) \land \\
(b_3 \lor r_2) \land \\
(-r_2 \lor -a_2 \lor b_2) \land \\
(-r_2 \lor -a_2 \lor r_1) \land \\
(-r_2 \lor b_2 \lor r_1) \land \\
(-r_1 \lor -a_1 \lor b_1) \land \\
(-r_1 \lor -a_1 \lor r_0) \land \\
(-r_1 \lor b_1 \lor r_0) \land \\
(-r_0 \lor -a_0 \lor b_0) \land \\
\langle a_3,a_2,a_1,a_0 \rangle \leq \langle b_3,b_2,b_1,b_0 \rangle \]
Solving Technologies: FD Solver

- Maintain *domains* of variables as sets of intervals
  - $x: [1..100]$, $y: [1,3,5..8]$; $z: [1..100,1000..2000]$;
- Use propagators to enforce domain consistency
- Combine search and propagation
- *Randomize the choice of variables and values*

+ Cheap on simple problems
+ Scales very well to large problems
+ Word-level processing
  - Easy to extend: new constraints, global constraints, randomization policies
  - Easy to explain e.g., in constraint debugger

- Limited control over randomness and distribution
- Bad in proving UNSAT
IntelliGen’s FD Solver

\textbf{Solve}(V,C) : [TRUE, FALSE]

\textbf{if} all variables in V are assigned \textbf{then return} TRUE

choose an unassigned variable \(x\) in \(V\)

\textbf{repeat}

\hspace{1em} choose a value \(k\) from the domain of \(x\)

\hspace{1em} \textbf{if} Propagate\((V\cdot[x/k], C) \) \&\& \textbf{Solve}(V, C) \textbf{then}

\hspace{2em} \textbf{return} TRUE

\textbf{else}

\hspace{1em} undo the last reduction

\hspace{1em} remove \(k\) from the domain of \(x\)

\textbf{until} range of \(x\) is empty

\textbf{return} FALSE
Propagation

- Removes inapplicable values from domains
  - $x: [1..100]$  
    $y: [1..100]$  
    $x < y$  
    $x: [1..99]$  
    $y: [2..100]$  

- Prunes the model by removing trivially satisfied constraints
  - $x: [1..10,100]$  
    $y: [15..100]$  
    $x < y$  
    $x: [1..10]$  
    $y: [15..100]$  

- Fails if domains are inconsistent
  - $x: [10,100]$  
    $y: [1..10]$  
    $x < y$  
    $\text{FAILURE}$
More On Propagation

• A more interesting example
  
x:[1..100], y:[1..100], x<y && x*x<300 && x+y>40 \rightarrow x:[1..17], y:[24..100], x+y>40

• IntelliGen includes propagation algorithms for
  
  – Relations  
  – Boolean logic  
  – Arithmetic  
  – Bitwise operations  
  – Arrays (global) constr.  
  – Pointers, strings, etc.

  \(== != < > <= >=\)
  
  \(\text{and or not} \Rightarrow :?\)
  
  \(+ - * / \%
  
  \(<> | \& ~\)
  
  \(\text{my}_\text{list}.\text{all}_\text{different}()\)
  
  \((p==p1 \text{ or } p==p2 \text{ or } p==\text{NULL})\)
The Hybrid Domain Representation

**Problem:** intervals are inadequate for representing bitwise information

- The interval representation the domain of \( x \) is \([0..2, 4..6, 8..10, 12..14...]\)
- interval representation has a billion fragments!!!
- … and bitmask representation can’t help!

**Solution:**

- Use a hybrid domain representation combining intervals and BDDs
- Do lazy updates between the two
BDD Propagation [Lagoon and Stuckey, CP’04]

• Assume a binary constraint \( c(x,y) \) represented by a BDD
• Assume two domains \( d(x) \) and \( d(y) \) of \( x \) and \( y \)
• The updated domains \( d'(x) \) and \( d'(y) \) are computed as

\[
d'(x) = d(x) \land [c(x, y) \land d(y)]_x
\]

\[
d'(y) = d(y) \land [c(x, y) \land d(x)]_y
\]

• The propagation fails if we get FALSE in any conjunction
• The propagator becomes redundant if

\[
d(x) \land d(y) \Rightarrow c(x, y)
\]

• Straightforward extension to any number of variables
Many Refinements of the Search Mechanism

• Heuristic choice of variable/value
  – Take into account domain size, role, connectivity, etc.

• Smart graph-based backtrack mechanisms
  – Save only the relevant domains before each assignment
  – Save on backtracking through independent sub-graphs

• Restarts

• Local and global backtrack limits
  – Get out of unproductive corners of the hyperspace quickly
  – Stop and signal an error rather than take forever in search
The types of “constraint bugs”
1. It can’t find me a solution
2. It finds a solution with unexpected values
3. It never finds a solution with expected values
4. It takes way too long/forever to find a solution

The main principles of constraint debugging
- Visualization
  • See the information you need in a clear and accessible way
- Navigation
  • Get to the information you need
- Minimization
  • See only the relevant information
Visualization

Menu Bar
- Edit
- View
- Navigate
- Run
- Options

Process Tree

Variables Pane

Generated Tree

Constraints Pane

Details Pane

© 2012 Cadence Design Systems, Inc. All rights reserved.
GenDebugger: Navigation

• Re-use the standard concepts of procedural debugging
  – Generation breakpoints / trace-points
    • Stop at any solving step, or at a specific partition, data type, object, field, variable, etc.
  – Step-by-step debugging
    • Step through atomic operations of the FD solver: propagation, assignment, backtrack
    • Walk into or walk over the generation of nested objects
    • Continue to the end of the context or to the next breakpoint

• GUI
  – See the hierarchy of objects, the time line of the generation, the relevant variables, and constraints, the source code
  – Access all constraints of a variable, all variables of a constraint, all past steps for a variable
GenDebugger: Minimization

- It is essential to minimize explanations:
  - In debugger, explaining propagations
  - In error messages explaining infeasibility

- Conservative minimization
  - Mark only the constraints that caused domain changes or failure
    + Does not cost anything
    + Sufficient in most cases
    ✗ Explanations may have redundancy

- Aggressive minimization
  - Try to remove constraints one by one
    + Produces a minimal set
    ✗ Significant performance overhead

```c
struct my_data {
    x : uint;
    y : uint;
    z : uint;
    keep x[4:0] == 0b11111;
    keep x<y;
    keep y<z;
    keep x+y<20;
    keep x+z<20;
};
```
Conclusion

- Verification is important
- Testing/simulation is (still) the main workhorse of verification
- CP is the backbone of today’s test generation
- The requirements differ from the “classic” CP
  - Problems are often easy (not much search)
  - Problems are often HUGE (hundreds of thousands elements)
  - Many random solutions required
  - Need to support many data types, including non-scalar
- There is no “best” constraint solving technology
  - Need to combine FD, BDD, SAT, etc. in a unified framework
  - Make different technologies benefit from each other
- Constraint debugging and debuggers are necessary
  - But mostly overlooked by the research community
Questions