Efficient Techniques for Directed Test Generation Using Incremental Satisfiability

Prabhat Mishra and Mingsong Chen

Embedded Systems Lab
Computer and Information Science and Engineering
University of Florida, USA
Outline

- Introduction
- Related Work
- Test Generation Using Incremental SAT
  - Clustering of similar properties
  - Name substitution for computation of intersection
  - Identify and reuse of common conflict clauses
- Experiments
- Conclusion
Introduction

* Functional verification is a major bottleneck
  - Increasing design complexity
  - Decreasing time-to-market

* Directed tests can reduce validation effort
  - Same coverage goal can be reached using small number of directed tests

* Model checking based test generation
  - Automated generation of directed tests
  - Unsuitable for large designs
    - State space explosion
  - Need to reduce test generation time (complexity)
Motivation

- SAT-based bounded model checking (BMC) can address state space explosion
  - Searches within a bound
  - CNF can be smaller than BDD
  - SAT has many heuristic decision algorithms
  - Exploit the similarity of SAT instances

- Existing approaches exploit similarity for the same test generation instance
  - Same property with different bounds

- We extend incremental SAT to exploit test generation involving multiple properties
SAT-based Bound Model Checking

- For every finite model and a LTL property $\phi$ there exists $k$ such that:

$$M \models_k \phi \rightarrow M \models \phi$$

- Test generation needs to consider safety properties.

- The safety property $P$ is valid up to cycle $k$ iff $\Omega(k)$ is not satisfiable.

$$\Omega(k) = I(S_0) \land \bigwedge_{i=0}^{k-1} R(S_i, S_{i+1}) \land \bigvee_{i=0}^{k} \neg P(s_i)$$

- If $\Omega(k)$ is satisfiable, then we can get an assignment which can be translated to a test.
Conflict clause can be treated as the knowledge learned during the SAT solving. It is a restriction of the variable assignment.
Incremental SAT

- Given two CNF formulas (sets of clauses) $S_1$ and $S_2$, the following statement holds.
  
  (1) Let $\pi$ be the conflict clause learned from $S_1$, then:
  
  $S_1$ is satisfiable iff $S_1 \land \pi$ is satisfiable
  
  (2) Let $\varphi_0 \equiv S_1 \cap S_2$, if $\pi$ is a conflict clause learned from $\varphi_0$ then:
  
  $S_1$ is satisfiable iff $S_1 \land \pi$ is satisfiable.
  $S_2$ is satisfiable iff $S_2 \land \pi$ is satisfiable.

- So when checking $S_2$, we can reuse the knowledge $\pi$ learned during checking $S_1$.

- Currently, the incremental SAT is used for checking the same property with different bounds.
The goal of our approach is to reduce the overall functional validation effort by reducing the test generation time for directed tests.

The basic idea is to learn from solving one property and sharing learning (through conflict clauses) for solving the similar properties in the cluster.

This paper focuses on test generation for safety properties. We assume that the bound for each property can be pre-determined based on the structure of the model.
Workflow of Our Method

1. **Cluster** the properties based on similarity

2. **for** each cluster \( i \), of properties
   
   ① **Select** base property \( p^i_1 \), and generate \( \text{CNF}^i_1 \)
   
   ② **for** each \( \text{CNF}^i_j \) of \( p^i_j \) \( (j \neq 1) \) in cluster \( i \)
      
      a) Perform **name substitution** on \( \text{CNF}^i_j \)
      
      b) Compute **intersection** \( \text{INT}^i_j \) between \( \text{CNF}^i_1 \) and \( \text{CNF}^i_j \)
      
      c) Mark the clauses of \( \text{CNF}^i_1 \) using \( \text{INT}^i_j \)
   
   ③ **Solve** \( \text{CNF}^i_1 \) to get the conflict clauses \( \text{CC}^i_1 \) and \( \text{test}^i_1 \)
   
   ④ **for** each \( \text{CNF}^i_j \) \( (j \neq 1) \)
      
      a) \( \text{CNF}^i_j = \text{CNF}^i_j + \text{Filter} \left( \text{CNF}_i, j \right) \)
      
      b) **Solve** \( \text{CNF}^i_j \) and get the \( \text{test}^i_j \)

**endfor**

**endfor**
Property Clustering

- More intersections imply more conflict clause forwarding. However, for n properties, clustering based on intersection need $n(n-1)/2$ comparisons.

- A simple and natural way to cluster properties is to exploit the structural and behavior similarity.

- Rules used for base property selection.
  - Variable and/or sub-expression overlap
  - Small bound.
The DIMACS file contains the mapping between the CNF variable and the variables of the model.

E.g. C 8 => V1_var [6]

Variable 8 is used to refer to 7th bit of variable var in the specification in time step 1.

Name substitution can get more intersection.

<table>
<thead>
<tr>
<th>C 1 =&gt; a_1</th>
<th>C 2 =&gt; b_1</th>
<th>C 3 =&gt; a_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>p cnf 3 3</td>
<td>p cnf 6 4</td>
<td>p cnf 6 4</td>
</tr>
<tr>
<td>-1 2 0</td>
<td>5 -4 0</td>
<td>2 -1 0</td>
</tr>
<tr>
<td>3 2 0</td>
<td>5 6 0</td>
<td>2 3 0</td>
</tr>
<tr>
<td>1 3 0</td>
<td>1 4 0</td>
<td>4 1 0</td>
</tr>
<tr>
<td></td>
<td>2 -3 0</td>
<td>5 -6 0</td>
</tr>
</tbody>
</table>

DIMACS f1  | DIMACS f2  | DIMACS f2’
Identification of Common Conflict Clauses

Conflict Clause

\[ \neg X_1 \lor X_5 \lor X_6 \lor \neg X_7 \]

Conflict Side Clauses

<table>
<thead>
<tr>
<th>Clauses</th>
<th>Group ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \neg X_2 \lor X_3 \lor X_8 ]</td>
<td>0 1 1 1</td>
</tr>
<tr>
<td>[ X_3 \lor \neg X_7 \lor \neg X_8 ]</td>
<td>1 0 1 0</td>
</tr>
<tr>
<td>[ X_2 \lor \neg X_3 \lor X_6 ]</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>[ \neg X_3 \lor \neg X_4 ]</td>
<td>1 0 1 0</td>
</tr>
<tr>
<td>[ \neg X_1 \lor X_4 \lor X_5 ]</td>
<td>1 1 1 0</td>
</tr>
</tbody>
</table>

Let \( \land \) be the bit “AND” operation. \((0111 \land 1010 \land 1111 \land 1010 \land 1110) = 0010\). So the conflict clause \((\neg X_1 \lor X_5 \lor X_6 \lor \neg X_7)\) can be reused for property 2.
This case study is a on-line stock exchange system. The activity diagram consists of 27 activities, 29 transitions and 18 key paths.

<table>
<thead>
<tr>
<th>Clusters (properties)</th>
<th>Preprocess Time</th>
<th>zChaff (sec.)</th>
<th>Our method (sec.)</th>
<th>Improv. Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cluster 1 (2)</td>
<td>3.79</td>
<td>59.82</td>
<td>4.43</td>
<td>13.50</td>
</tr>
<tr>
<td>Cluster 2 (4)</td>
<td>11.98</td>
<td>78.13</td>
<td>13.68</td>
<td>5.72</td>
</tr>
<tr>
<td>Cluster 3 (4)</td>
<td>11.81</td>
<td>161.91</td>
<td>40.50</td>
<td>4.00</td>
</tr>
<tr>
<td>Cluster 4 (4)</td>
<td>12.70</td>
<td>144.12</td>
<td>51.80</td>
<td>2.78</td>
</tr>
<tr>
<td>Cluster 5 (4)</td>
<td>12.76</td>
<td>426.09</td>
<td>75.34</td>
<td>5.66</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>4.08</strong></td>
<td><strong>48.33</strong></td>
<td><strong>10.32</strong></td>
<td><strong>4.68</strong></td>
</tr>
</tbody>
</table>
Case Study 2: MIPS Processor

The Architecture

MIPS Processor
- 20 nodes
- 24 edges
- 91 instructions
Case Study 2: MIPS Processor

- The processor has five pipeline stages: fetch, decode, execute, memory and writeback. The execute stage has four execution path, 1 stage integer ALU, 7 stages multiplier, 4 stage floating point adder and one multi-cycle divider.

<table>
<thead>
<tr>
<th>Clusters</th>
<th>CNF Clauses</th>
<th>Intersection Size</th>
<th>zChaff</th>
<th>Our Method</th>
<th>Improv. Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLALU</td>
<td>460994</td>
<td>457168</td>
<td>19.35</td>
<td>5.10</td>
<td>3.79</td>
</tr>
<tr>
<td>CLFADD</td>
<td>592119</td>
<td>67894</td>
<td>61.61</td>
<td>42.46</td>
<td>1.45</td>
</tr>
<tr>
<td>CLMUL</td>
<td>854368</td>
<td>522283</td>
<td>718.85</td>
<td>159.21</td>
<td>4.51</td>
</tr>
<tr>
<td>CLDIV</td>
<td>526517</td>
<td>457160</td>
<td>35.07</td>
<td>8.19</td>
<td>4.28</td>
</tr>
<tr>
<td>Average</td>
<td>608504</td>
<td>376126</td>
<td>208.72</td>
<td>53.74</td>
<td>3.88</td>
</tr>
</tbody>
</table>
Conclusions

- Functional validation is a major bottleneck
- Test generation using SAT-based BMC
  - Incremental SAT involving one property (test)
- Directed test generation using Incremental SAT
  - Share learning across multiple properties
    - Clustering of similar properties
    - Name substitution for computation of intersection
    - Identify and reuse of common conflict clauses
  - Reduces test generation time and complexity
    - Four times improvement in test generation time for both software and hardware designs
Thank you !