Collaborative Verification and Testing

Sungmin Cho
EECS, UC Berkeley
Outline

• Motivations and Ideas
  • Pros and Cons of Verification and Testing
  • Combining Verification and Testing
• More advanced research
  • Ketchum by Ho et al.
  • Synergy by Gulavani et al.
The verification approach

• It tries to construct the formal proof that the implementation meets the specification

• Pros
  • Successful proof is easy to find
  • If it is proved to be correct, it is mathematically correct.

• Cons
  • Often inefficient in finding errors
  • State explosion, complex data structure and algorithm
The testing approach

• It tries to find inputs and executions which demonstrate violations of the property

• Pros

  • Works best when errors are easy to find
  • Relatively easy to implement the algorithm

• Cons

  • Often difficult to achieve sufficient coverage
  • The passing the test doesn’t mean that there is no bug
Today’s topics

• Ketchum by Ho et al. (2000, Synopsys)
  • Random Simulation
  • Symbolic Simulation and SAT based BMC
• Synergy by Gulavani et al. (2006, Microsoft)
  • Synergy between verification and testing
    • Testing for finding bugs
    • Verification for proving
  • Synergy between F and A data structure
The motivation for Ketchum

- We’re interested in IDLE/Empty, Write/Normal ...
- We’re also interested in Read/Empty, Write/Full is left unvisited
- Coverage signal: signals that is given and we have a interest in.
- Coverage state: Each combination of Coverage Signals.
Ketchum - basic ideas

• Visit all the (or as many as) states quickly: Automatic Test Generation
  • Random Simulation - Testing
  • Symbolic Simulation - Verification
  • SAT-based BMC - Verification
• Reduce the number of states: Unreachability
  • Identifies as many unreachable coverage states as possible
  • Can find unreachable states fast using projection method
Ketchum Algorithm

- Rectangle - the entire state space
- Stars - Coverage states
- Zig-zag - random simulation
- Circle - Symbolic simulation
Comparisons of search engines

<table>
<thead>
<tr>
<th>Engine</th>
<th>Effective Search Range</th>
<th>Strength</th>
<th>Limitation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random simulation</td>
<td>Long</td>
<td>Deep states</td>
<td>Single trace</td>
</tr>
<tr>
<td>Symbolic simulation</td>
<td>Medium</td>
<td>Designs with fewer inputs</td>
<td>Time, memory, length of trace</td>
</tr>
<tr>
<td>SAT-based BMC</td>
<td>Short</td>
<td>Short hit traces</td>
<td>Time, length of trace</td>
</tr>
</tbody>
</table>

- The algorithm starts with Random simulation
- Extremely fast
- Reaches very deep states
- But, searches along a single trace/line
- They used commercial software for this
Reachable Analysis

• Ketchum uses Reachability Analysis by BDD Based state enumeration

• But, how to check if the newly found states using BDD is visited or not?

  • Mark as ‘unclassified’ for the new coverage states

  • Replace ‘unclassified new coverage states’ with ‘symbolic formula of the coverage signal’

  • If the result after the operation is not null, a new coverage state has been reached by symbolic simulation. We update the unclassified BDD and generate a trace to be used in simulation.
Observations

• The # of symbolic variables that have been used during simulation has “more impact” on the complexity of the symbolic simulation than the # of latches

• The # of symbolic variables is (# of PI * simulation steps)

• So, symbolic simulation is good only for wide range exhaustive search

• The under-approximation of replacing some symbolic variables to constant 1/0.
SAT Based BMC

- Ketchum uses ‘unreachability engine’ to reduce the state space to search

- The targeted coverage states are
  - States that are not reached
  - States that are not proven unreachable
  - Uses SAT based BMC to find them by expanding i steps

- Kethum’s method is good for exhaustive short-range search engine for it has reduced search
Ketchum input/output

• Input
  • Synthesizable MUT (Model under Test)
  • A set of less than 64 ‘coverage signals’

• Output
  • Test sequence to reach as many coverage as possible
  • Identifies as many unreachable coverage as possible
Ketchum Algorithm

while(find all the state) {
    simulation to find states
    if (rate falls below a threshold) {
        SAT-based BMC
        if (does not reach coverage states) {
            Symbolic simulation
            if (reach coverage states) {
                resimulation
            }
        } else // If it finds a state
            resimulation
    }
} // simulation starts again
Interesting results

- After the exhaustive search, the next reachable states are easily found by the random simulation as a next step.
- There are easy-to-transition signals (signals that can find a new state easily) and hard-to-transition signals (signals that can find a new state hard).
- After exhaustive search, the engine manages to reach a hard-to-transition signals.
- Random simulation will bump into different combinations of the easy-to-transition/hard-to-transition.
Unreachability - goal

- Provide fast and robust results without necessarily trying to detect all of the unreachable states.
- If we can find unreachable states, we just skip them to fasten the search.
- For an CPU example
  - The # of coverage states becomes from 1102 to 60
Unreachability - approach

• They could prove a state unreachable
• They could \textit{not} prove a state reachable
• Conservative method - prune model of MUT (Model under test) : Pruned MUT
• New idea - Select latch/combinational logics to include this pruned model
• The pruning can be regarded as an abstraction process
Pruned MUT projection

MUT

Pruned MUT

Coverage

Signal

Reachable

Unreachable
Latch selection

• Using BFS, one can find the latch dependancy to add the result

• After the selection of subset of latches, cutting algorithm to reduce the number of variables in the support of the transitive fan in.

• All the other latches are considered as PI
Unreachability

- We can do better, for it is not the “# of gates” that we want to reduce, but “# of signals” in the support of the transition functions.
The effect of unreachable analysis

- Pruned model + Optimized number of latches
- smaller number of variables
- smaller BDD sizes

I don’t have to visit the unreachable states!
Implementation

• Tools used
  • Main programming - C
  • Simulator - Verilog
  • Symbolic Simulator - BuDDy
  • SAT - GRASP

• We can have a much smaller number of states
• The number of Latch >> Coverage Signal
Results

<table>
<thead>
<tr>
<th>DUT</th>
<th>Latch</th>
<th>Cov sig.</th>
<th>Cov state after Kchm unreachable</th>
<th>Reach cover states Rndm</th>
<th>Reach cover states Kchm</th>
<th>Imp (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IU</td>
<td>4558</td>
<td>17</td>
<td>230 445sec</td>
<td>9 24hr</td>
<td>40 24hr</td>
<td>344</td>
</tr>
<tr>
<td>8051</td>
<td>784</td>
<td>11</td>
<td>896 299sec</td>
<td>317 24hr</td>
<td>597 24hr</td>
<td>88</td>
</tr>
<tr>
<td>DCU</td>
<td>385</td>
<td>25</td>
<td>111 259sec</td>
<td>109 24hr</td>
<td>111 (2\text{min})</td>
<td>2</td>
</tr>
<tr>
<td>SMU</td>
<td>217</td>
<td>16</td>
<td>132 1423sec</td>
<td>104 24hr</td>
<td>132 (45\text{min})</td>
<td>30</td>
</tr>
<tr>
<td>Bus</td>
<td>155</td>
<td>16</td>
<td>342 60sec</td>
<td>44 24hr</td>
<td>342 (75\text{min})</td>
<td>677</td>
</tr>
</tbody>
</table>

- Improvement is from 2% to 677%
- The # of coverage state after unreachable analysis is reduced much.
## Results

<table>
<thead>
<tr>
<th></th>
<th>Cov sig.</th>
<th>Cov state aft Kchm unreach</th>
<th>Reach cover states Rndm</th>
<th>Reach cover states Kchm</th>
<th>Imp (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1H</td>
<td>4558</td>
<td>17</td>
<td>230</td>
<td>9</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>445sec</td>
<td>24hr</td>
<td></td>
</tr>
<tr>
<td>8051</td>
<td>784</td>
<td>11</td>
<td>896</td>
<td>317</td>
<td>597</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>299sec</td>
<td>24hr</td>
<td></td>
</tr>
<tr>
<td>DCU</td>
<td>385</td>
<td>25</td>
<td>111</td>
<td>109</td>
<td>111</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>259sec</td>
<td>24hr</td>
<td>2min</td>
</tr>
<tr>
<td>SMU</td>
<td>217</td>
<td>16</td>
<td>132</td>
<td>104</td>
<td>132</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1423sec</td>
<td>24hr</td>
<td>45min</td>
</tr>
<tr>
<td>Bus</td>
<td>155</td>
<td>16</td>
<td>342</td>
<td>44</td>
<td>342</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>60sec</td>
<td>24hr</td>
<td>75min</td>
</tr>
</tbody>
</table>

- Improvement is from 2% to 677%
- The # of coverage state after unreachable analysis is reduced much.
• Improvement is from 2% to 677%
• The # of coverage state after unreachable analysis is reduced much.
What gives the good result in Ketchum

• The test generation only focus on coverage states that are reachable, so fast and correct in terms of the verification result.

• Back-bone of Ketchum is an off the shelf commercial simulator that is very efficient.

• As a result - it has the 10x higher capacity/coverage result.
Synergy

- Software verification method
- Synergy between testing method and verification method
- Synergy between F and A data structure
- Testing to find bugs
  - Testing can help refine verification
- Verification to find proof
  - Verification can help grow the test results
Counter example guided partition refinement - SLAM

• Find error, and refine on and on
• Might have too big counter-examples
• Loop causes problems in this case
• Case split works well
DART - Directed Testing

- Exhaustively generates input vectors

- Normally, not work well with many branches
Lee-Yannakakis algorithm

- $T :=$ initial state
- $S := \{\text{Initial, Error, } S \setminus (\text{Initial + Error})\}$
- Loop
  - Error: If $S$ in $S$ is included in Error and $S$ and $T$ has common set
  - Find a new state $s$ that is reachable from $T$
  - If you can find it, add it to $T$
  - If you can’t find it
    - Refine
    - If you can’t refine, it’s a Proof
Lee-Yannakakis Algorithm

\textsc{Lee-Yannakakis}(P = (\Sigma, \sigma^I, \rightarrow), \psi)
Assumes: \sigma^I \cap \psi = \emptyset.
Returns:
(“fail”, t), where t is an error trace of P reaching \psi; or
(“pass”, \Sigma_{\sim}), where \Sigma_{\sim} is a proof that P cannot reach \psi.

1: T := \sigma^I
2: \Sigma_{\sim} := \{\sigma^I, \psi, \Sigma \setminus (\sigma^I \cup \psi)\}
3: loop
4: for all S \in \Sigma_{\sim} do
5: if S \cap T \neq \emptyset and S \subseteq \psi then
6: choose s \in S \cap T
7: t := \text{TestFromWitness}(s)
8: return (“fail”, t)
9: end if
10: end for
11: choose S \in \Sigma_{\sim} such that S \cap T = \emptyset and
12: there exist s \in S and t \in T with t \rightarrow s
13: if such S \in \Sigma_{\sim} and s, t \in \Sigma exist then
14: T := T \cup \{s\}
15: parent(s) := t
16: else
17: choose P, Q \in \Sigma_{\sim} such that P \cap T \neq \emptyset and
18: Pre(Q) \cap P \neq \emptyset and P \not\subseteq Pre(Q)
19: if such P, Q \in \Sigma_{\sim} exist then
20: \Sigma_{\sim} := (\Sigma_{\sim} \setminus \{P\}) \cup \{P \cap Pre(Q), P \setminus Pre(Q)\}
21: else
22: return (“pass”, \Sigma_{\sim})
23: end if
24: end if
25: end loop
LY vs. Synergy

- Synergy is based on the LY algorithm
  - Loop structure, fail test, refinement
  - The idea of stability (bisimulation) is not used in Synergy
- \(<P,Q>\) is stable if
  - \(P\) and Pre(Q) = NULL or
  - \(P\) included in Pre(Q)
- If not stable, refinement is needed

\[ \text{Pre}(S_k) = \{ s \in \Sigma \mid \exists s' \in S_k, s \rightarrow s' \} \]
LY vs. Synergy

• Synergy doesn’t attempt to find a part of the bisimilarity quotient

• When Synergy terminates with a proof, the partition does not necessarily form a bisimilarity quotient

• The distinguishing feature of the SYNERGY algorithm is the simultaneous search for a test case to witness an error and a partition to witness a correctness proof
Synergy data structure

- F structure
  - Forest to store the findings in Testing
  - When there is an abstract path, it is added to F structure

- A structure
  - Abstract to store the refinements in Verification
  - A is refined more and more by looking into F structure - F gives hints how to refine
• There exists a frontier($S_0, S_1, ..., S_n$) such that (a) $0 \leq k \leq n$, and (b) $S_i$ and $F = 0$ for all $k \leq i \leq n$ and (c) $S_j$ and $F$ not 0 for all $0 \leq j < k$

• The trace with frontier is “ordered trace”

• Frontier is a mark in A structure used to direct F structure to know what attempt it has to do
F, A structure and Frontier
Synergy API

- CreateAbstractProgram
  - Given partition, returns Program
- GetAbstractTrace
  - Searches for abstract error trace
- Frontier
- TestFromWitness
- GetOrderedAbstractTrace
  - Given trace, returns \(<\text{Terr}, k> \) \(k=\text{frontier}\)
- RefineWithGeneralization

\(<\Sigma_\sim, \sigma^I_\sim, \rightarrow_\sim>\)
Synergy algorithm overview

- Fail - return with an error trace \( t \)
  - Same as LY
- Pass - return with a proof that cannot reach error states
  - If GetAbstractTrace return null
  - It means that there is no abstract/concrete error trace that leads to Error
- Basic algorithm is (almost) same as LY.
  - F & A data structure is used
  - Refine procedure is based on \( S_{k-1} \) and \( S_k \)
Synergy($P = \langle \Sigma, \sigma^f, \rightarrow \rangle, \psi$)
Assumes: $\sigma^f \cap \psi = \emptyset$.
Returns:
("fail", $t$), where $t$ is an error trace of $P$ reaching $\psi$; or
("pass", $\Sigma_{\approx}$), where $\Sigma_{\approx}$ is a proof that $P$ cannot reach $\psi$.

1: $F := \emptyset$
2: $\Sigma_{\approx} := \{\sigma^f, \psi, \Sigma \setminus (\sigma^f \cup \psi)\}$
3: loop
4: for all $S \in \Sigma_{\approx}$ do
5: if $S \cap F \neq \emptyset$ and $S \subseteq \psi$ then
6: choose $s \in S \cap F$
7: $t := \text{TestFromWitness}(s)$
8: return ("fail", $t$)
9: end if
10: end for
11: $(\Sigma_{\approx}, \sigma^f_{\approx}, \rightarrow_{\approx}) := \text{CreateAbstractProgram}(P, \Sigma_{\approx})$
12: $\tau = \text{GetAbstractTrace}((\Sigma_{\approx}, \sigma^f_{\approx}, \rightarrow_{\approx}), \psi)$
13: if $\tau = \epsilon$ then
14: return ("pass", $\Sigma_{\approx}$)
15: else
16: $(\tau_{err}, k) := \text{GetOrderedAbstractTrace}(\tau, F)$
17: $t := \text{GenSuitableTest}(\tau_{err}, F)$
18: let $S_0, S_1, \ldots, S_n = \tau_{err}$ in
19: if $t = \epsilon$ then
20: $\Sigma_{\approx} := (\Sigma_{\approx} \setminus \{S_{k-1}\}) \cup$
21: $\{S_{k-1} \cap \text{Pre}(S_k), S_{k-1} \setminus \text{Pre}(S_k)\}$
22: else
23: let $s_0, s_1, \ldots, s_m = t$ in
24: for $i = 0 \text{ to } m$ do
25: if $s_i \notin F$ then
26: $F := F \cup \{s_i\}$
27: parent($s_i$) := if $i = 0$ then $\epsilon$ else $s_{i-1}$
28: end if
29: end for
30: end if
31: end if
32: /*
33: The following code is commented out,
34: and is explained in Section 5:
35: $\Sigma_{\approx} := \text{RefineWithGeneralization}(\Sigma_{\approx}, tt)$
36: */
37: end loop

• Every line corresponds to each state
• Refinement corresponds to a state(line) with variables
• We can get refinement more and more with more variables
Example of Synergy

• error() occurs if $a \leq 0$

• First, $F$ is empty

• $A$ is $\{0,1,2,5,6\}$ and $Front$ is at position 0(‘0’)

• Generate vector to go 0,1:
  Let’s say 10

• $F$ has $\{0,1,2,3,4,5\}$ and Frontier is 3(‘5’)

• Generate vector to go 5,6 :
  Let’s say -10

• We find an error trace

```c
void foo(int a)
{
    int i, c;
    0:   i = 0;
    1:   c = 0;
    2:   while (i < 1000) {
    3:       c = c + i;
    4:       i = i + 1;
    5:   }
    6:   assume(a <= 0);
    7:   error();
}
```
Example of Synergy

• Line 7: means that $x == y$ when out of the loop

• $lock.state = U$ when $x == y$

• So, there should be no error
Example

- GetOrderedAbstractTrace returns \(<0,1,2,3,4,7,8,9>,0\>
- \(F\) tries to generate vector: \(y = 10\)
- \(F\) has \(<0,1,2,3,4,5,6,7,8>\), so the frontier is 6 (line 8)
- There is no way to get another \(F\), so refinement is needed
  - \(<0,1,2,3,4,5,6,7,<8,p>,9>,6\> Refinement is processed until there is no refinement available
  - \(<0,1,2,3,4,5,6,<7,q>,<8,p>,9>,7\>
- It proves that the program has “passed”
Soundness of Synergy

• Theorem - Suppose that we run the Synergy algorithm on a Program P and Property Error

• If Synergy returns ("pass",Sigma), then the partition Sigma with respect to Error, and thus is a proof that P cannot reach Error

• If Synergy returns ("fail", t) then t is an error trace

• Which means that every found proof and error is valid
Problem in Synergy

- Refinement step on line 20-21 unable to find the "right split"
  - predicate with $y < 0$
  - then, $y + x < 0$
  - then, $y + 2x < 0$
  - for ever until memory limit
- `RefineWithGeneralization()` function is needed for solving this kind of problem

```c
void foo()
{
    int x, y;
    1:  x = 0;
    2:  y = 0;
    3:  while (y >= 0) {
        4:      y = y + x;
    }
    5:  assert(false);
}
```
Comparison with other tools

• Synergy works well with if
  • Overcome the problem of SLAM

• Synergy works well with branch
  • Overcome the problem of DART

• Synergy solves the problem that LY can or can’t solve
## Results

<table>
<thead>
<tr>
<th>Program</th>
<th>Synergy</th>
<th></th>
<th>Slam</th>
<th></th>
<th>Lee-Yannakakis</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>iters</td>
<td>time</td>
<td>iters</td>
<td>time</td>
<td>iters</td>
<td>time</td>
</tr>
<tr>
<td>test1.c</td>
<td>9</td>
<td>3.92</td>
<td>4</td>
<td>1.70</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>test2.c</td>
<td>6</td>
<td>7.88</td>
<td>4</td>
<td>1.55</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>test3.c</td>
<td>5</td>
<td>2.19</td>
<td>13</td>
<td>8.032</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>test4.c</td>
<td>2</td>
<td>2.67</td>
<td>12</td>
<td>3.52</td>
<td>22</td>
<td>8.08</td>
</tr>
<tr>
<td>test5.c</td>
<td>2</td>
<td>1.28</td>
<td>1</td>
<td>0.90</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>test6.c</td>
<td>1</td>
<td>1.45</td>
<td>1</td>
<td>1.27</td>
<td>1</td>
<td>1.75</td>
</tr>
<tr>
<td>test7.c</td>
<td>6</td>
<td>2.11</td>
<td>4</td>
<td>1.11</td>
<td>6</td>
<td>2.06</td>
</tr>
<tr>
<td>test8.c</td>
<td>2</td>
<td>1.28</td>
<td>2</td>
<td>1.19</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>test9.c</td>
<td>3</td>
<td>1.39</td>
<td>1</td>
<td>1.19</td>
<td>3</td>
<td>1.42</td>
</tr>
<tr>
<td>test10.c</td>
<td>3</td>
<td>1.52</td>
<td>1</td>
<td>1.25</td>
<td>3</td>
<td>1.52</td>
</tr>
<tr>
<td>test11.c</td>
<td>2</td>
<td>1.30</td>
<td>13</td>
<td>5.03</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>test12.c</td>
<td>7</td>
<td>2.30</td>
<td>13</td>
<td>10.25</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>test13.c</td>
<td>12</td>
<td>3.17</td>
<td>2</td>
<td>1.31</td>
<td>12</td>
<td>3.18</td>
</tr>
<tr>
<td>test14.c</td>
<td>1</td>
<td>1.0625</td>
<td>12</td>
<td>3.453</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>test15.c</td>
<td>3</td>
<td>5.98</td>
<td>*</td>
<td>*</td>
<td>3</td>
<td>5.65</td>
</tr>
<tr>
<td>test16.c</td>
<td>3</td>
<td>9.20</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>test17.c</td>
<td>2</td>
<td>2.28</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>test18.c</td>
<td>24</td>
<td>13.41</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>test19.c</td>
<td>24</td>
<td>10.84</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>test20.c</td>
<td>22</td>
<td>9.42</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>
How about verilog code?

• Synergy doesn’t have the function testing.
• Verilog’s instantiation is easily adapted.
• The real problem is how to deal with the parallelization process of Verilog.
• For the F structure, the state graph is bigger than the state graph for Verilog.