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

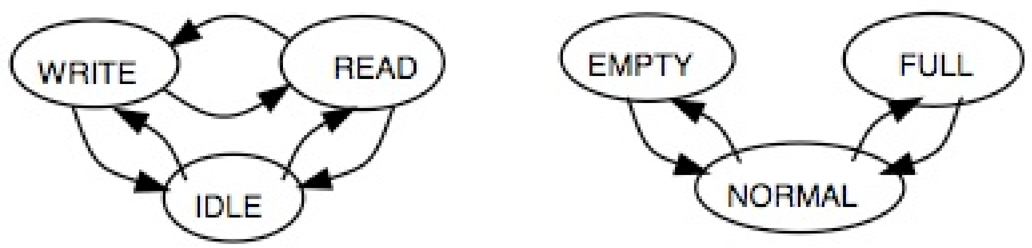
The testing approach

- 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

Engine	Effective Search Range	Strength	Limitation
Random simulation	Long	Deep states	Single trace
Symbolic simulation	Medium	Designs with fewer inputs	Time, memory, length of trace
SAT-based BMC	Short	Short hit traces	Time, length of trace

- *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 'unclassifed' 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 ‘unreachablity 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

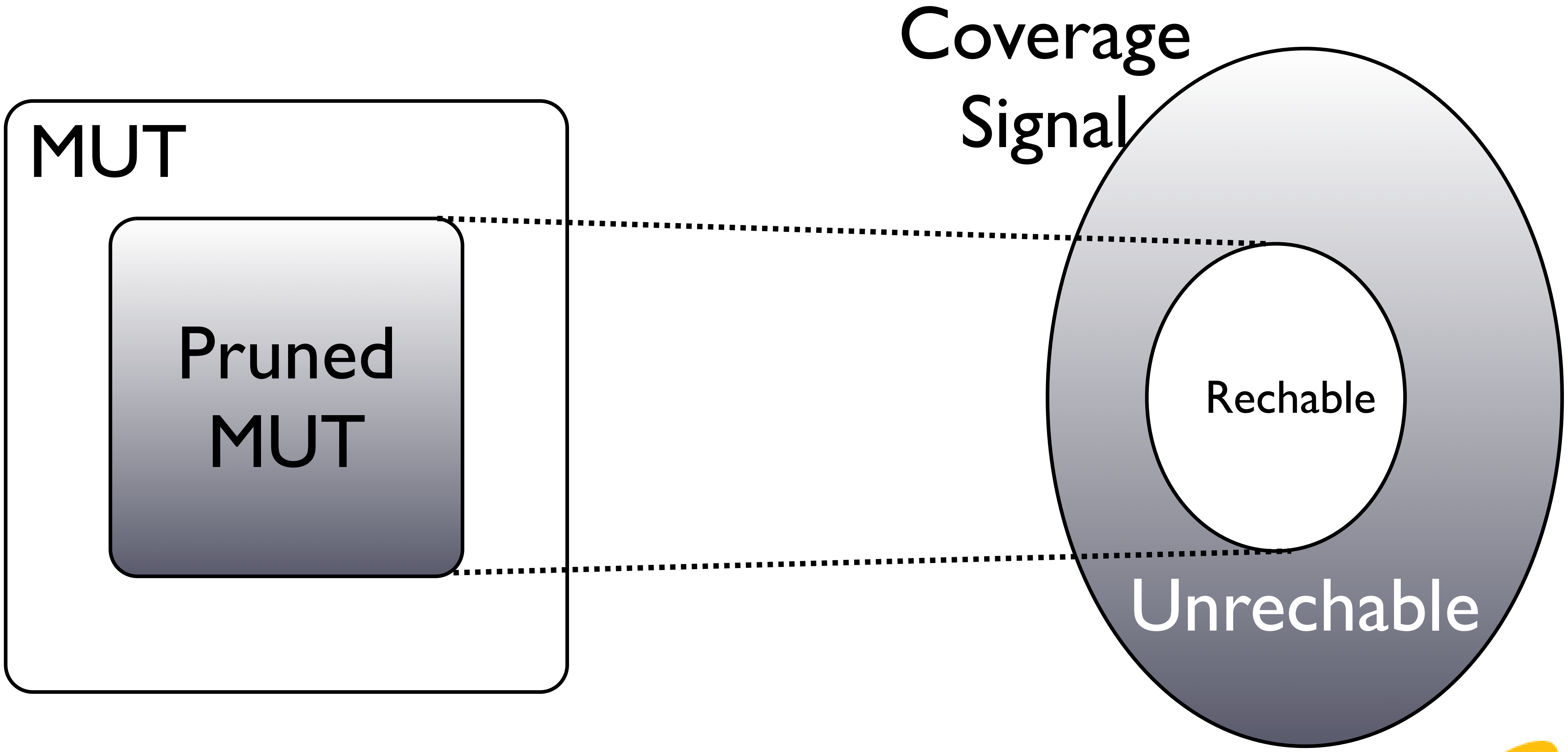
Unreachability - goal

- 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 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

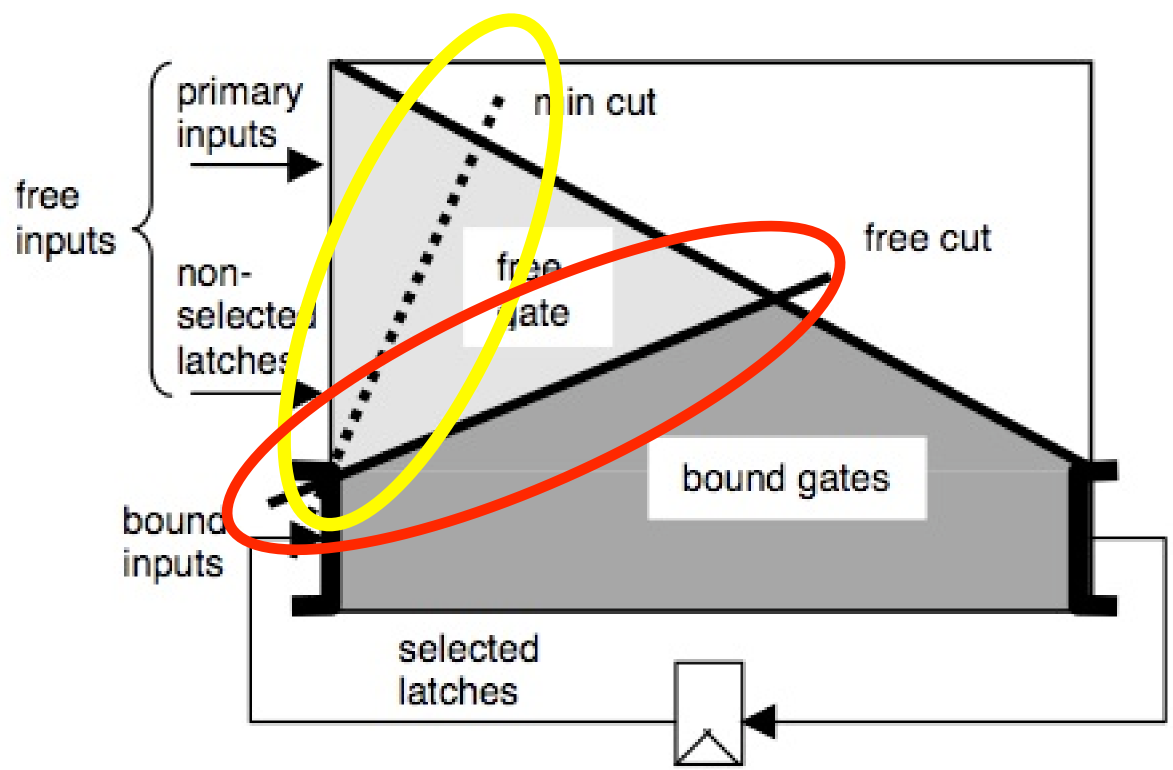
Pruned MUT projection



Latch selection

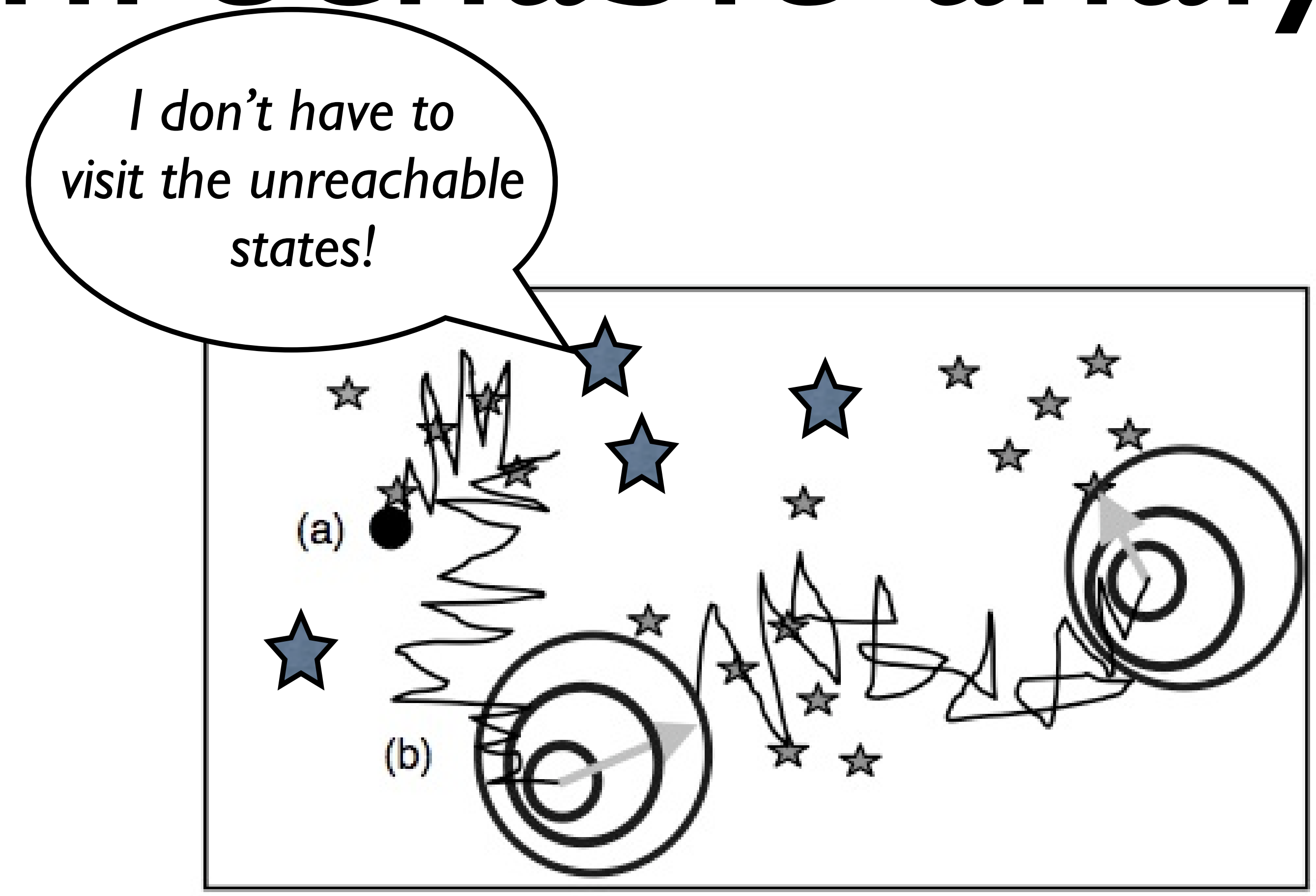
- Using BFS, one can find the latch dependancy to add the result
- After the selection of subset of latches, cutting algorithm to reduce the number of variables in the support of the transitive fan in.
- All the other latches are considered as PI

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 unrechable analysis



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

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

DUT	Ltch	Cov sig.	Cov state aftr Kchm unreach	Reach cover states Rndm	Reach cover states Kchm	Imp (%)
IU	4558	17	230 *445sec*	9 *24hr*	40 *24hr*	344
8051	784	11	896 *299sec*	317 *24hr*	597 *24hr*	88
DCU	385	25	111 *259sec*	109 *24hr*	111 ***2min***	2
SMU	217	16	132 *1423sec*	104 *24hr*	132 ***45min***	30
Bus	155	16	342 *60sec*	44 *24hr*	342 ***75min***	677

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

Results

Unreachability idea applied

		Cov sig.	Cov state aftr Kchm unreach	Reach cover states Rndm	Reach cover states Kchm	Imp (%)
IO	4558	17	230 *445sec*	9 *24hr*	40 *24hr*	344
8051	784	11	896 *299sec*	317 *24hr*	597 *24hr*	88
DCU	385	25	111 *259sec*	109 *24hr*	111 ***2min***	2
SMU	217	16	132 *1423sec*	104 *24hr*	132 ***45min***	30
Bus	155	16	342 *60sec*	44 *24hr*	342 ***75min***	677

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

Results

Unreachability idea applied

Automatic test generation idea applied

		Cov sig.	Cov state aftr Kchm unreach	Reac cover states Rndr	Kch	
IU	4558	17	230 *445sec*	9 *24hr*	40 *24hr*	344
8051	784	11	896 *299sec*	317 *24hr*	597 *24hr*	88
DCU	385	25	111 *259sec*	109 *24hr*	111 ***2min***	2
SMU	217	16	132 *1423sec*	104 *24hr*	132 ***45min***	30
Bus	155	16	342 *60sec*	44 *24hr*	342 ***75min***	677

- Improvement is from 2% to 677%
- The # of coverage state after unrechable 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.

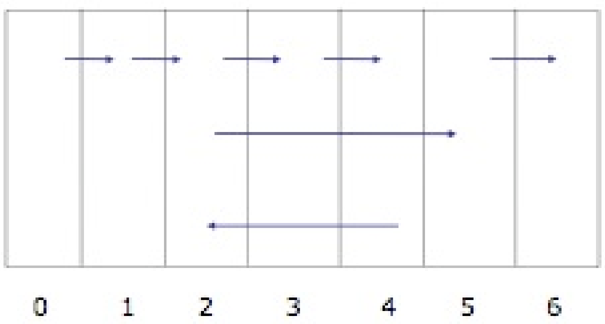
Synergy

- 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

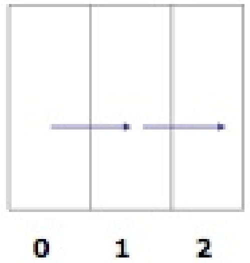
```
    void foo(int a)
    {
      int i, c;
0:    i = 0;
1:    c = 0;
2:    while (i < 1000) {
3:      c = c + i;
4:      i = i + 1;
      }
5:    assume(a <= 0);
6:    error();
    }
```



DART - Directed Testing

- Exhaustively generates input vectors
- Normally, not work well with many branches

```
     void foo(int x, int y)
     {
0:     if (x != y)
1:       if (2*x = x + 10)
2:         error();
     }
```




```
        void foo()
        {
0:        lock.state = L;
1:        if (*) {
2:          x0 = x0 + 1;
          }
3:        else {
4:          x0 = x0 - 1;
          }
5:        if (*) {
6:          x1 = x1 + 1;
          }
7:        else {
8:          x1 = x1 - 1;
          }

          ...

m:        if (*) {
m+1:        xn = xn + 1;
          }
m+2:      else {
m+3:        xn = xn - 1;
          }
m+4:      if (lock.state != L)
m+5:        error();
        }
```

Lee-Yannakakis algorithm

- T := initial state
- S_ := {Initial, Error, S_\(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

LEE-YANNAKAKIS($P = \langle \Sigma, \sigma^I, \rightarrow \rangle$, ψ)
Assumes: $\sigma^I \cap \psi = \emptyset$.
Returns:
("fail", t), where t is an error trace of P reaching ψ; or
("pass", $\Sigma_{\simeq}$), where $\Sigma_{\simeq}$ is a proof that P cannot reach ψ.

$T := \sigma^I$
$\Sigma_{\simeq} := \{\sigma^I, \psi, \Sigma \setminus (\sigma^I \cup \psi)\}$
loop
  for all $S \in \Sigma_{\simeq}$ **do**
    if $S \cap T \neq \emptyset$ and $S \subseteq \psi$ **then**
      choose $s \in S \cap T$
      $t :=$ TestFromWitness(s)
      return ("fail", t)
    end if
  end for
  choose $S \in \Sigma_{\simeq}$ such that $S \cap T = \emptyset$ and
      there exist $s \in S$ and $t \in T$ with $t \rightarrow s$
  if such $S \in \Sigma_{\simeq}$ and $s, t \in \Sigma$ exist **then**
    $T := T \cup \{s\}$
    $parent(s) := t$
  else
    choose $P, Q \in \Sigma_{\simeq}$ such that $P \cap T \neq \emptyset$ and
        $\mathrm{Pre}(Q) \cap P \neq \emptyset$ and $P \not\subseteq \mathrm{Pre}(Q)$
    if such $P, Q \in \Sigma_{\simeq}$ exist **then**
      $\Sigma_{\simeq} := (\Sigma_{\simeq} \setminus \{P\}) \cup \{P \cap \mathrm{Pre}(Q), P \setminus \mathrm{Pre}(Q)\}$
    else
      return ("pass", $\Sigma_{\simeq}$)
    end if
  end if
end loop

LY vs. Synergy

- Synergy is based on the LY algorithm
  - Loop structure, fail test, refinement
  - The idea of stability(bisimilation) 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
    - $Pre(S_k) = \{s \in \Sigma \mid \exists s' \in S_k, s \rightarrow s'\}$

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

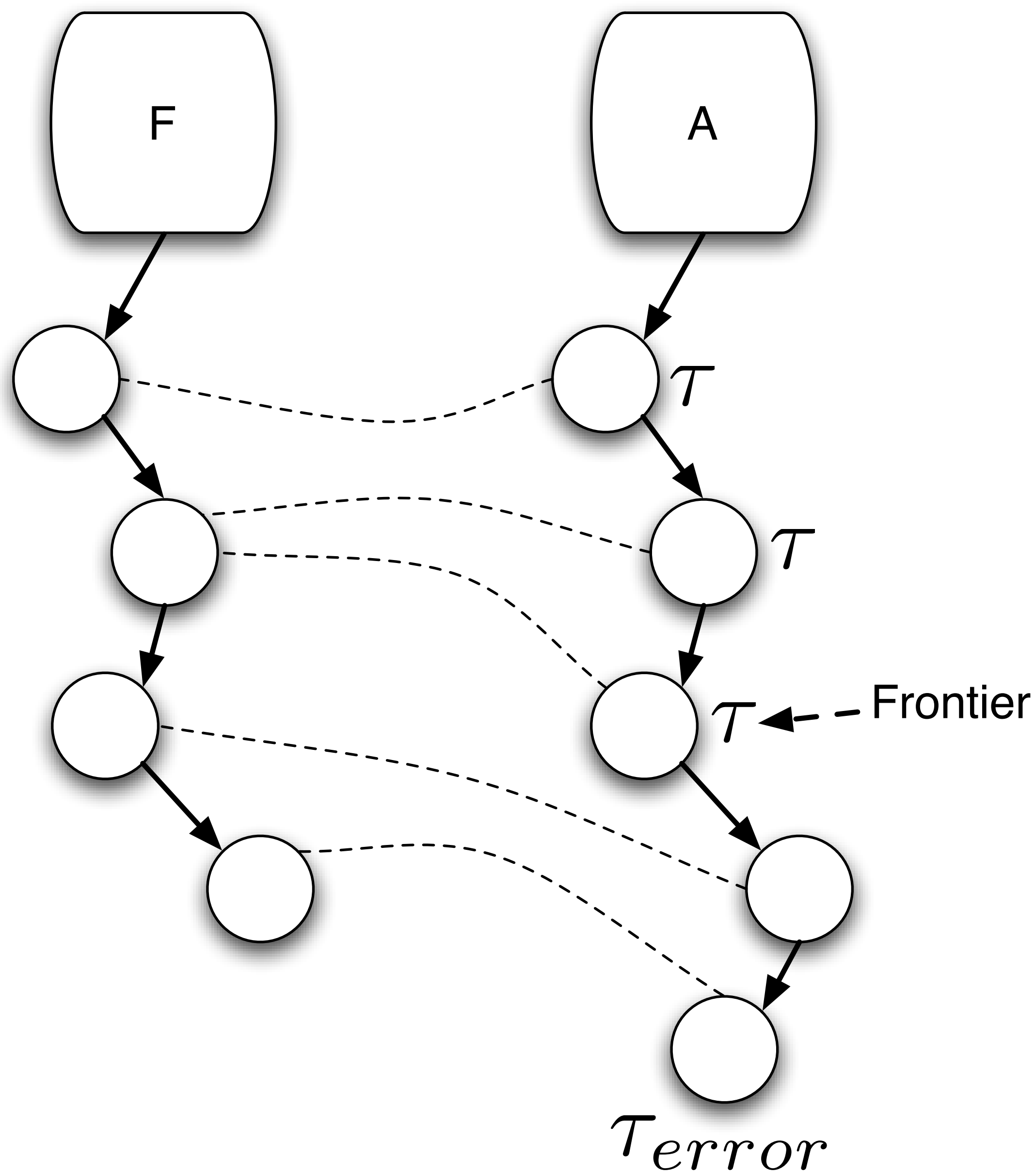
Synergy data structure

- 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

Frontier in Synergy

- There exists a frontier($S_0, S_1, ..., S_n$) such that (a) $0 <= k <= n$, and (b) S_i and F = 0 for all $k <= i <= n$ and (c) S_j and F not 0 for all $0 <= j < k$
- 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

$$< \Sigma_{\simeq}, \sigma^{I}_{\simeq}, \rightarrow_{\simeq} >$$

- GetAbstractTrace
  - Searches for abstract error trace
- Frontier
- TestFromWitness
- GetOrderedAbstractTrace
  - Given trace, returns <Terr, k> k=frontier
- RefineWithGeneralization

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 algorithm

SYNERGY($P = \langle \Sigma, \sigma^I, \rightarrow \rangle$, ψ)
Assumes: $\sigma^I \cap \psi = \emptyset$.
Returns:
("fail", t), where t is an error trace of P reaching ψ; or
("pass", $\Sigma_{\simeq}$), where $\Sigma_{\simeq}$ is a proof that P cannot reach ψ.

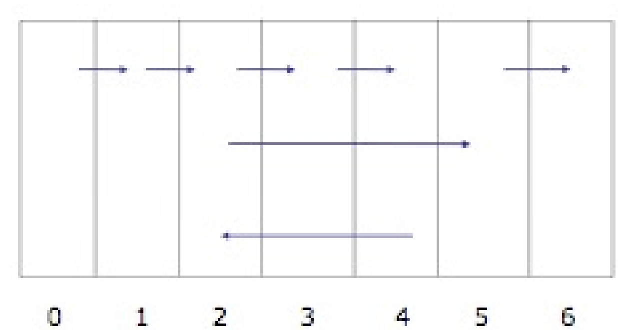
$F := \emptyset$
$\Sigma_{\simeq} := \{\sigma^I, \psi, \Sigma \setminus (\sigma^I \cup \psi)\}$
loop
  for all $S \in \Sigma_{\simeq}$ **do**
    if $S \cap F \neq \emptyset$ and $S \subseteq \psi$ **then**
      choose $s \in S \cap F$
      $t :=$ TestFromWitness(s)
      return ("fail", t)
    end if
  end for
  $\langle \Sigma_{\simeq}, {\sigma^I}_{\simeq}, \rightarrow_{\simeq} \rangle :=$ CreateAbstractProgram($P, \Sigma_{\simeq}$)
  $\tau =$ GetAbstractTrace($\langle \Sigma_{\simeq}, {\sigma^I}_{\simeq}, \rightarrow_{\simeq} \rangle, \psi$)
  if $\tau = \epsilon$ **then**
    return ("pass", $\Sigma_{\simeq}$)
  else
    $\langle \tau_{err}, k \rangle :=$ GetOrderedAbstractTrace(τ, F)
    $t :=$ GenSuitableTest(τ_{err}, F)
    let $S_0, S_1, \ldots, S_n = \tau_{err}$ **in**
    if $t = \epsilon$ **then**
      $\Sigma_{\simeq} := (\Sigma_{\simeq} \setminus \{S_{k-1}\}) \cup$
        $\{S_{k-1} \cap \mathrm{Pre}(S_k), S_{k-1} \setminus \mathrm{Pre}(S_k)\}$
    else
      let $s_0, s_1, \ldots, s_m = t$ **in**
      for $i = 0$ **to** m **do**
        if $s_i \notin F$ **then**
          $F := F \cup \{s_i\}$
          $parent(s_i) :=$ **if** $i = 0$ **then** ϵ **else** s_{i-1}
        end if
      end for
    end if
  end if
  /*
  The following code is commented out,
  and is explained in Section 5:
  $\Sigma_{\simeq} :=$ RefineWithGeneralization($\Sigma_{\simeq}, tt$)
  */
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 <= 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*

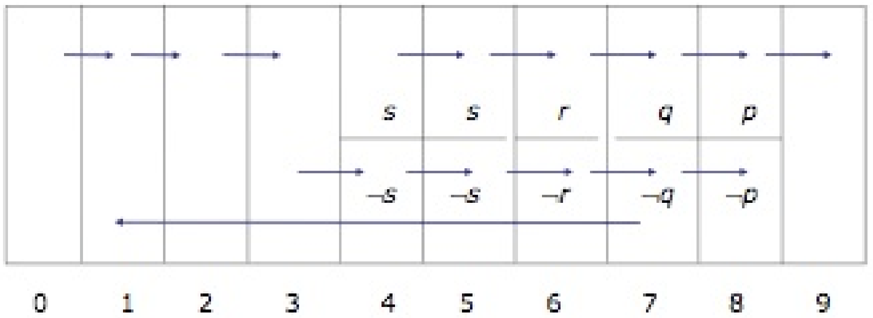
```
   void foo(int a)
   {
     int i, c;
0:   i = 0;
1:   c = 0;
2:   while (i < 1000) {
3:     c = c + i;
4:     i = i + 1;
     }
5:   assume(a <= 0);
6:   error();
   }
```



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*

```
    void foo(int y)
    {
0:    lock.state = U;
1:    do {
2:      lock.state = L;
3:      x = y;
4:      if (*) {
5:        lock.state = U;
6:        y++;
        }
7:    } while (x != y)
8:    if (lock.state != L)
9:      error();
    }
```



p: (lock.state != L)

q: (lock.state != L) && (x == y)

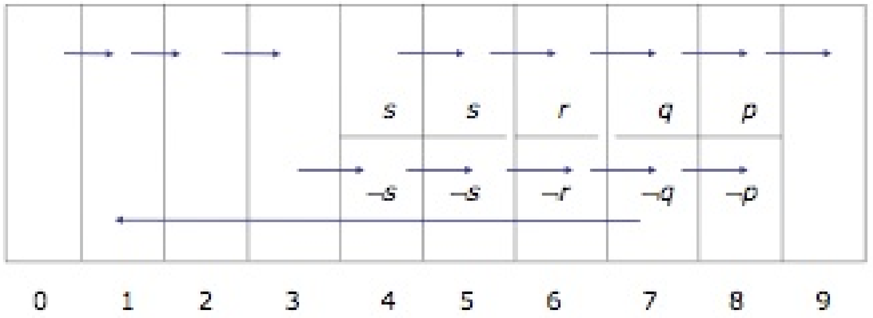
r: (lock.state != L) && (x == y+1)

s: (x == y+1)

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"*

```
    void foo(int y)
    {
0:    lock.state = U;
1:    do {
2:      lock.state = L;
3:      x = y;
4:      if (*) {
5:        lock.state = U;
6:        y++;
        }
7:    } while (x != y)
8:    if (lock.state != L)
9:      error();
    }
```




```
p: (lock.state != L)
q: (lock.state != L) && (x == y)
r: (lock.state != L) && (x == y+1)
s: (x == y+1)
```

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*

```
    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

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

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.