ECE 553 Project Report — Standard

Zak Smith

December 1, 1997

Contents

Chapter 1

Problem Statement

1. Given a circuit description, generate a compact test set with high fault coverage.

2. Using test set from (1), determine whether three supplied circuits contain an injected fault.

3. Finally, determine the fault location in a fourth supplied circuit.

Chapter 2

Results

2.1 Test Set for Fault Detection

The test set used for fault detection was made up of 87 vectors. The test set was handed in electronically.

2.2 Fault Coverage

All detectable faults were covered. There were 8 undetectable faults. Using a fault list with reduction based on structural equivalence, there were a total of 1574 faults, which yields a fault coverage of 99.492%.

```
*** PPCPT fault simulation: Net - ../netlist ***

Passes    :   Sim. Time   :  Fault Coverage
pass    1 : time =   0.03 : coverage =  89.6
pass    2 : time =   0.03 : coverage =  95.2
Last pass :
pass    3 : time =   0.03 : coverage =  99.5

total patterns number = 87
total simulation time = 0.03
fault coverage = 1566/1574 = 99.492
```

2.3 Undetectable Faults

There were 8 undetectable faults:

```
    985    925    1
    983    912    1
    982    899    1
    984    886    1
    978    873    1
    979    860    1
    980    847    1
    981    834    1
```

2.4 Common Circuit Result

2.4.1 Circuit #1

Circuit #1 was faulty. It was detected by vector #13.

2.4.2 Circuit #2

Circuit #2 had either no fault or had an undetectable fault.

2.4.3 Circuit #3

Circuit #3 was faulty. It was detected by vector #1.

2.5 Fault Diagnosis Result

Two faults were in the final set:

```
826 654 1
827 782 1
```

These two faults are equivalent, and the injected fault is either one of these faults, or equivalent to these faults.

Chapter 3

Test Generation

3.1 Original Strategy

Our original strategy was to produce an initial set with very high coverage, invoke the fault simulator only once, and then "clean up" any remaining faults using podem.[1] Finally, the sortcover and accumulate tools are applied for further reduction of the test set.

To produce an initial high–coverage test set, we used scoap to select the faults with the highest 10% testability values. podem was then applied to these supposed "hard" faults.

While scoap is not a reliable measure of an individual fault's detection probability, it should do a better job of predicting which fault sets have lower probability of detection.[2] Ideally, we would use static compaction to target the unused inputs for these vectors, but podem did not leave any inputs unspecified.

Since the remaining faults should be easy to test, it was hoped that a simple heuristic could be developed to figure out how many random vectors are needed.

Pseudo–code for the original strategy s-r-c:

```
1.  Use listfaults to generate fault list, using structural equivalence
2.  Run scoap to find faults with top 10% testability
3.  Generate tests for faults from (2) using podem
4.  Try static compaction (vcompact), and randomized inputs (randvec -v)
5.  Generate 3*n random vectors (randvec -i)
6.  Find so-far undetected faults with ppcpt -u
7.  Generate tests for faults in (6) with podem
8.  Run sortcover and accumulate to compact vector set
```

The actual scripts are contained in Appendix B.

3.2 Actual Strategy

The actual strategy was a fast and simple two–pass method. First, we simulate blocks of n random vectors, running the fault simulator for each block. Simulation of random blocks ends when the returns diminish — that is, when, for n new vectors, we get coverage of less than n additional faults. At this point, it is just as expensive to switch to podem and generate tests for all the remaining faults. Podem generates one vector per fault.

To be sure that an added vector always covers at least one fault, n would have to be 1. This yields lots of overhead because the fault simulator must be invoked for every added fault. We arbitrarily chose n to be 1% of the total number of faults. This value, 16, worked well.

Finally, sortcover and accumulate are used to compact the final set.

Pseudo–code for the final strategy ir-c:

```
1.  Use listfaults to generate fault list, using structural equivalence
2.  Loop while additional faults detected >= n
    a. Use randvec to generate n random vectors
    b. Append list in (a) to vector list
    c. Run the fault simulator ppcpt on the vector list
3.  Run podem to clean--up any leftover faults
4.  Run sortcover and accumulate to compact vector set
```

3.3 Details

In generating the test vectors, 5 iterations of random vector set generation occurred before the termination criteria was met, resulting in 80 random vectors. At this point, the fault coverage was 87.42%, and podem created tests for the remaining 190 detectable faults.

Sortcover and accumulate then reduced this list of 270 vectors to the final number of 87.

[1] I use lowercase "podem" and "scoap" to avoid confusion with the ATG tools

[2] Agrawal & Mercer: "Testability Measures – What Do They Tell Us?"

3.4 Discussion

The initial s-r-c method seemed good, but it had a few disadvantages. The first problem was scoap, which did not give a good indication of testability. After podem, only easy faults should have been left for the random vectors, but we still had a significant number of faults left in the clean–up stage. No obvious heuristic became apparent to estimate the number of random vectors needed. Another disadvantage of this method is that podem is slow–running, and we run it twice.

After discovering that the initial scoap/podem method did not work well, we moved to a r-c method: choose some large number of random vectors, and do cleanup on whatever is left. Finally, we compact the final set with sortcover and accumulate. This method produced a compact final test set, and had low run–time, but we had no good way to justify our choice of the number of random vectors to try.

Next, we tried what ended up being our final method, ir-c: incremental random vectors, iterate until the completion criteria, and then do cleanup and compaction. This method was fast because podem was only run once.

The next method we tried was ir-id-c: iterate on random, then iterate on scoap and podem, and finally do cleanup and compaction. Running podem on all remaining faults generates one vector per fault. This method iterates running podem on the hard faults so that multiple vectors are not produced for two faults when an earlier podem–generated vector covers the later fault. This method suffers because scoap does not work very well, and it has very slow run–time because podem is invoked many times.

Deciding to investigate further, we tried method ird-c: iterate on random and hard vectors from scoap with podem, and finally cleanup and compaction. For the ir-c method, the number of faults which successive random vectors covered decreased as the iterations went on — the slope of the fault/vector curve decreased. The idea for this method was to keep the slope of that curve steep by catching the hard faults (scoap) after each random iteration. This method suffered because scoap is not a good measure, and it was slow because podem was run so often.

In the end, we decided on ir-c: incremental random, and cleanup and compaction. All of the methods had final vector counts within 5% of each other, and the final vector count often just depended on the random numbers we happened to get. We chose ir-c as the final method because: it was a simple algorithm; it was fast; and on average, it generated a few less vectors than the other methods.

Chapter 4

Fault Diagnosis

4.1 Original Strategy

The originally–planned strategy was to run a set of test vectors against the to–be–diagnosed circuit. For each vector, v_i, which excited the injected fault, save the list of faults which v_i detects. Once finished, for each fault, f_j, count how many vectors (ex: v_i) had f_j in their fault list. The fault with the most contributing vectors should be the injected fault.

Another way of saying it is this: each vector has an associated set of faults which it can detect. If a vector excites the fault, we know the injected fault is one of the faults in its fault list. The injected fault must be in the fault list of every vector which excited the fault. Thus, the intersection of all of the exciting–vectors' fault lists will contain the injected fault.

4.2 Actual Strategy

The original strategy is a good start, but it does not use all the information we have. Besides using the "inclusion" idea to know which faults are most likely to be the injected fault, we can also use an "exclusion" idea: for each vector which did not excite the fault, we know the injected fault is not in any of those fault lists. Thus, we can subtract the set of all non–exciting–vector faults from the final list from the original strategy. This reduces the final "candidate list" by removing the faults we know cannot be the injected fault.

4.3 Details

Here is pseudo–code for the final method:

```
Input: a vector list V
Output: a sorted list of faults

1. Make sure the fault list for the golden circuit exists
2. Run logic simulator on golden circuit with vector list V, save as GoldenOV

3. Run logic simulator on faulty circuit with vector list V, save as FaultyOV

4. Find the output vectors which differ between GoldenOV and FaultyOV, save as DifferV

5. Find the output vectors which are the same between GoldenOV and FaultyOV, save as SameV

6. For each vector v in DifferV, do the following
   a. figure out which faults v detects, by
      i.  use ppcpt to find out which faults v does not detect
      ii. subtract (i) from the total fault list
   b. append this fault list to the CandidateList

7. For each vector v in SameV, do the following
   a. figure out which faults v detects, by
      i.  use ppcpt to find out which faults v does not detect
      ii. subtract (i) from the total fault list
   b. append this fault list to the ExcludeList

8. Subtract the set ExcludeList from the set CandidateList
9. Sort the result from (8), and count the number of occurrences of each fault

10. The fault with the most number of occurrences is the injected fault
```

The actual scripts are contained in Appendix A.

4.4 Discussion

Initially, I ran our “small” test vector set (87) used for diagnosis. After the “inclusion” phase, 23 faults were in the candidate set. After exclusion, only 2 remained.

To make sure that these two faults could not be further distinguished, I iteratively ran the algorithm with 50 additional random test vectors each time. After running this for some time with no distinction between the two faults, I decided they were equivalent to each other.

It is interesting to note that even using a small test set which was designed to maximize the fault coverage while minimizing set size, it was possible to diagnose the fault by using all the information available.

Chapter 5

Conclusion

The fundamental problem of test–set generation is that is is difficult to generate small, high–coverage sets. This is difficult because it is another example of the "minimum common set" problem, which is NP–complete. While still difficult to generate absolute minimum test sets, it is easy to generate reasonably small test sets using random and heuristic methods.

Given basic tools — a fault simulator, a logic simulator, a directed test generator (like podem), and some heuristic (maybe something better than podem) — it is easy to generate good test sets.

Further, with a little thought and some glue code, it is also fairly easy to write a general fault–diagnosis algorithm. Fault diagnosis depends on the ideas of "set inclusion" and "set exclusion." Vectors which excite the fault have their faults included, and vectors which do not excite the fault have their faults subtracted from this candidate list. If the small test designed for fault detection does not offer enough information for fault diagnosis, other vectors with other fault–sets can be added to help the process.

Chapter 6

Appendix A - Fault Diagnosis

This appendix contains all the scripts written to facilitate automation of the fault diagnosis process.

6.1 run

The main algorithm is contained in `run`. It is a shell script which drives the process.

```
#!/bin/sh
#
# code copyright (c) 1997 by Zak Smith, all rights reserved
#
# Pseudo-code:
#
# For some vector set V, do the following:
#
# 1. Do logic simulation on both the good and faulty circuits
# 2. For each vector, which excites the error:
#      a. figure out which faults this vector detects
#      b. save this list in the output directory
#
# *IF* there is a single fault which occurred more than any others, it is our
# fault!  Explanation: The "count" is incremented once for every vector
# which detects this fault (and we did observe that this vector produced
# different outputs in the good circuit vs. the faulty circuit), so there
# is at least one vector which detects this error, and no other
# vector detects it.
#
# Another way to think of it is this:
#     Each of the per-vector-fault-lists is the set of
#     vectors which that test detects, so one of them *is* the
#     real fault.
#
#     We want to do more test vectors until the set intersection of
#     all of these sets is 1 fault, which means that one of our
#     fault "counters" will be higher than all the others -- ie,
#     more tests intersect it than any other fault
#
# But..  we can do better.  For all the test vectors which did *not*
# excite the fault, we know that *none* of the faults in its set are
# the actual fault, so we can exclude all of those faults from
# the final list.
#
# After this script finishes, the sorted fault list is in:
#       results/sorted-results
#
#
##########################

export PATH=~/work/ece/553/proj/zak/bin:$PATH:~testcad/public/bin

A=`echo $1 | sed 's/\.v//'`

FL=faults.f
V=$A.v
```

```
C=zak.n
CF=zak.f
G=good.n
GF=good.f

#
# First, make sure they gave us a vector-set name
#
if [ ! -f $V ]; then
     echo Vector list "$V" does not exist.
     echo usage: $0 \<vectorlist\>
     exit
fi

#
# If the fault-list for the good circuit doesn't exist, create it
#
if [ ! -f $GF ]; then
     echo Creating fault list for good circuit
     listfaults -n $G -ae > $GF
fi

#
# If the fault-list for the bad circuit doesn't exist, create it
#
if [ ! -f $CF ]; then
     echo Creating fault list
     listfaults -n $C -ae > $CF
fi

#
# Now we need the outputs for our vector-set, for the good circuit
#
echo Generating Good Output $V.out.good
lsim -n $G -v $V | OV-1line > results/$V.out.good

#
# Now we need the outputs for our vector-set, for the faulty circuit
#
echo Generating $V.out.zak
lsim -n $C -v $V | OV-1line > results/$V.out.zak

#
# We just compare and see if we actually detected any errors
#
goodhash=`hash results/$V.out.good`
zakhash=`hash results/$V.out.zak`
if [ $goodhash != $zakhash ]; then
     echo Files differ: $V.out.good $V.out.zak
else
     echo Files the same..
fi

#
```

```
# figure out which test vectors detect our error
#
# This selects the vector numbers (ie, OV 13) which excite the error, in
# our current test-set
#
comm results/$V.out.good results/$V.out.zak -13 | \
     cut -d: -f1 | sed 's/OV *//' > results/$V.useful-vectors

#
# figure out which test vectors do NOT detect our error
#
# This selects the vector numbers (ie, OV 13) which do not excite the error, in
# our current test-set
#
comm results/$V.out.good results/$V.out.zak -12 | \
     cut -d: -f1 | sed 's/OV *//' > results/$V.nofault-vectors

#
#
# Now, for each of these exciting vectors, we want to get a list of all the
# faults it stimulates
#
for a in `cat results/$V.useful-vectors`; do
     echo $a

     #
     # extract the current vector ($a) from the vector set
     #
     get-vector $V $a > /tmp/curvec.v

     #
     # to figure out what faults we *do* detect, we figure out which
     # ones we don't and then subtract the sets
     #
     ppcpt -n $G -f $GF -v /tmp/curvec.v -u /tmp/undet.f

     #
     # .. so we need to fix the fault list format because the -u format
     # and the output of listfaults are not exactly the same
     #
     cat $GF | fix_fl > /tmp/my_fl
     cat /tmp/undet.f | fix_fl > /tmp/un_fl

     #
     # now take the faults which are *not* repeated in the full and
     # undetected lists, ie, the subtraction of the sets
     #
     cat /tmp/my_fl /tmp/un_fl | sort | uniq -u > results/$V.myfaults.$a

     #
     # print out the number, because we are curious
     #
     wc -l results/$V.myfaults.$a
done
```

```
#
# Now, we want to build a list of faults to *exclude*, in other words:
#
# For each of the non-exciting vectors, we want to get a list of all
# the faults they stimulate
#
for a in `cat results/$V.nofault-vectors`; do
     echo $a

     #
     # extract the current vector ($a) from the vector set
     #
     get-vector $V $a > /tmp/curvec.v

     #
     # to figure out what faults we *do* detect, we figure out which
     # ones we don't and then subtract the sets
     #
     ppcpt -n $G -f $GF -v /tmp/curvec.v -u /tmp/undet.f

     #
     # .. so we need to fix the fault list format because the -u format
     # and the output of listfaults are not exactly the same
     #
     cat $GF | fix_fl > /tmp/my_fl
     cat /tmp/undet.f | fix_fl > /tmp/un_fl

     #
     # now take the faults which are *not* repeated in the full and
     # undetected lists, ie, the subtraction of the sets
     #
     cat /tmp/my_fl /tmp/un_fl | sort | uniq -u > results/$V.my_nof.$a

     #
     # print out the number, because we are curious
     #
     wc -l results/$V.my_nof.$a
done

#
# do the subtraction of sets -- we know that none of the faults in *.my_nof.*
# is our fault, so we just take the lines with are in *.myfaults.* and
# not in *.my_nof.*
#

( find results -name '*myfaults*' | xargs cat ) > results/ALL-MYFAULTS
( find results -name '*my_nof*'   | xargs cat ) > results/ALL-MYNOF-intermediate

# make the file smaller by removing duplicates
cat results/ALL-MYNOF-intermediate | sort | uniq > results/ALL-MYNOF

comm -23 results/ALL-MYFAULTS results/ALL-MYNOF > results/FAULT-CANDIDATES
```

```
cat results/FAULT-CANDIDATES | sort | uniq -c | sort -n > results/sorted-results
```

6.2 4_fields

This script removes all spaces, and prints the 4 space–separated fields in a consistent way.

```
#!/usr/bin/awk -f
{
     printf("%s %s %s %s\n", $1, $2, $3, $4);
}
```

6.3 get-vector

The useful script returns the n^{th} vector from a vector–list.

```
#!/bin/sh
file=$1
want=$2

cat $file | v-1line |grep ^Test| head -$2l | tail -1l
```

6.4 OV-1line

Modify the `lsim` output file so that there is only one line per vector.

```
#!/usr/local/bin/perl
while (<>) {
     chop;
     if (/^OV/) {
          print "$a\n";
          $a = $_;
     } else {
          $a = $a . $_;
     }
}
```

6.5 fix_fl

This is to make sure that fault–lists have the exact same format. It works similar to `4_fields`.

```
#!/usr/bin/awk -f
{
     printf("%s %s %s\n", $1, $2, $3);
}
```

6.6 hash

This returns the md5 hash of a file, useful for comparisons.

```
#!/bin/sh
md5sum $1 | cut -d' ' -f1
```

6.7 v-1line

Modify a vector–list file so that there is only one line per vector.

```
#!/usr/local/bin/perl
while (<>) {
     chop;
     if (/^Test/) {
          print "$a\n";
          $a = $_;
     } else {
          s/^Cont://;
          $a = $a . $_;
     }
}
```

Chapter 7

Appendix B - Test Generation

7.1 r-c

```
#!/bin/sh
export PATH=$PATH:~testcad/public/bin
FL=fault.f
NL=netlist

RN=1000

listfaults -n $NL -ae > $FL
randvec -n $NL -i $RN > rand.v
ppcpt -n $NL -f $FL -v rand.v  -u undet.f

podem -n $NL -o undet.v -f undet.f

cat undet.v rand.v > final.v
sortcover -n $NL -v final.v -f $FL -o sorted.v
accumulate -n $NL -v sorted.v  -f $FL -o acc.v
ppcpt -n $NL -f $FL -v acc.v
```

7.2 ir-id-c

```
#!/bin/sh
export PATH=$PATH:~testcad/public/bin
FL=fault.f
NL=netlist

RN=20
SN=2

listfaults -n $NL -ae > $FL

r=-
true > all_rand.v

while [ $r != y ]; do
     randvec -n $NL -i $RN >> all_rand.v
     ppcpt -n $NL -f $FL -v all_rand.v -u undet.f
     echo done\?
     read r
done
r=-
while [ $r != y ]; do
     scoap -n $NL -o highest.s -f undet.f -F $SN
     cat highest.s | cut -d: -f1 > highest.f

     podem -n $NL -o inc_pod.v -f highest.f

     cat all_rand.v inc_pod.v >> all_rand.v2
     mv all_rand.v2 all_rand.v
     ppcpt -n $NL -f $FL -v all_rand.v -u undet.f

     echo done\?
     read r
```

```
done

podem -n $NL -o undet.v -f undet.f

cat undet.v all_rand.v > final.v
sortcover -n $NL -v final.v -f $FL -o sorted.v
accumulate -n $NL -v sorted.v  -f $FL -o acc.v
ppcpt -n $NL -f $FL -v acc.v
```

7.3 ird-c

```
#!/bin/sh
export PATH=$PATH:~testcad/public/bin
FL=fault.f
NL=netlist

RN=20
SN=2

listfaults -n $NL -ae > $FL

r=-
true > all_rand.v

while [ $r != y ]; do
     randvec -n $NL -i $RN >> all_rand.v
     ppcpt -n $NL -f $FL -v all_rand.v -u undet.f

     scoap -n $NL -o highest.s -f undet.f -F $SN
     cat highest.s | cut -d: -f1 > highest.f

     podem -n $NL -o inc_pod.v -f highest.f

     cat all_rand.v inc_pod.v >> all_rand.v2
     mv all_rand.v2 all_rand.v
     ppcpt -n $NL -f $FL -v all_rand.v -u undet.f

     echo done\?
     read r
done

podem -n $NL -o undet.v -f undet.f

cat undet.v all_rand.v > final.v
sortcover -n $NL -v final.v -f $FL -o sorted.v
accumulate -n $NL -v sorted.v  -f $FL -o acc.v
ppcpt -n $NL -f $FL -v acc.v
```

7.4 r-c

```
#!/bin/sh
export PATH=$PATH:~testcad/public/bin
FL=fault.f
```

```
NL=netlist

RN=1000

listfaults -n $NL -ae > $FL
randvec -n $NL -i $RN > rand.v
ppcpt -n $NL -f $FL -v rand.v  -u undet.f

podem -n $NL -o undet.v -f undet.f

cat undet.v rand.v > final.v
sortcover -n $NL -v final.v -f $FL -o sorted.v
accumulate -n $NL -v sorted.v  -f $FL -o acc.v
ppcpt -n $NL -f $FL -v acc.v
```

7.5 s-r-c

```
#!/bin/sh
export PATH=$PATH:~testcad/public/bin
FL=fault.f
NL=netlist

SN=150  # scoap number
RN=400

listfaults -n $NL -ae > $FL
scoap -n $NL -o highest.s -f $FL -F $SN
cat highest.s | cut -d: -f1 > highest.f

podem -n $NL -o highest.v -f highest.f
vcompact -v highest.v > s_highest.v

ppcpt -n netlist -f fault.f -v s_highest.v

randvec -n $NL -i $RN > rand.v
ppcpt -n $NL -f $FL -v rand.v

cat s_highest.v rand.v > z.v
ppcpt -n $NL -f $FL -v z.v -u undet.f

podem -n $NL -o undet.v -f undet.f

cat z.v undet.v > final.v
ppcpt -n $NL -f $FL -v final.v
sortcover -n $NL -v final.v -f $FL -o sorted.v
accumulate -n $NL -v sorted.v  -f $FL -o acc.v
ppcpt -n $NL -f $FL -v acc.v
```