Test Pattern Generation (TPG)

1 Introduction

In the final phase of the project, you will complete your simulator by designing a *test pattern generation* (TPG) system. Many of the necessary modules have already been implemented in earlier phases. Our focus will now shift to designing heuristics that optimize the process of generating a **"test"** for a digital circuit, specifically finding a set of test patterns that can detect all possible single stuck-at faults in a given circuit.

As discussed in class, the general objectives for generating a test are:

- **Runtime**: With a fixed amount of computational resources, our aim is to minimize this.
- **Fault Coverage (FC)**: A constraint. You must achieve the highest possible FC.
- **Test Volume**: This should be minimized.
- **Memory**: This is not a major concern in this project.
- **Cost**: Also not a major concern in this project.

1.1 Development of Heuristics

Heuristics are practical shortcuts or *rules of thumb* used in problem-solving and decision-making. They are not perfect or guaranteed to find the "optimal solution", but they are effective for quickly finding a reasonable solution when it's too time-consuming or complex to achieve the perfect answer. You should follow this four-step process to develop new heuristics:

1. **Intuition**: Start with why you believe your idea is beneficial. There should be a meaningful rationale behind your intuition.
2. **Design**: Avoid rushing into coding. First, design your heuristic. This step can be executed by writing the algorithm in pseudo-code or by creating a flowchart. Focus on the crucial elements and leave the details for the coding phase.
3. **Implementation**: Begin coding.
4. **Experiment and Analysis**: Evaluate how your intuition fares by comparing the new algorithm's results with previous outcomes, using the objectives mentioned above. You should be able to justify your results through an analysis based on different benchmarks.

Remember, there is no guarantee that a heuristic will improve results across all objectives or on different circuits. Sometimes experimental results may challenge our initial intuition or hypothesis. Do not be discouraged; there are still valuable lessons to be learned from these scenarios.

2 TPG Heuristics

We have identified heuristics across various categories. Many of these heuristics are similar, and implementing one can simplify the development of others. The categories are:

- Using random test patterns (RTPs)
- Selecting from the D-Frontier in ATPG
- Justification in ATPG
- Fault order
- Test volume compression
- Choice of fault simulation method

2.1 Baseline

The baseline, or as sometimes called the vanilla version, is the simplest form of the algorithm where no heuristics are applied, i.e. in our case not using any random test patterns, no fault order, using the default ATPG algorithm, etc. Therefore, the algorithm would be very simple.

Note: We provided an example of a pseudo-code for the vanilla TPG algorithm below. This is just an example, it is not completely correct and requires adding more details and further modifications.

Algorithm 1: Vanilla test pattern generate (baseline)

Input: circuit, choice of ATPG (DALG, PODEM)
Output: TPs: A set of test patterns
$FL \leftarrow$ all single stuck at faults
$TPs \leftarrow$ empty list
while *FL is not empty or max FC is achieved* **do**
    f $\leftarrow$ select random from FL
    tp $\leftarrow$ ATPG(f)
    tp is ternary, you may need to modify it here
    $TPs \leftarrow TPs + tp$
    $DFs \leftarrow PFS(FL, tp)$ (DFs: list of detected faults)
    $FL \leftarrow FL - DFs$
return TPs

2.2 Random test patterns (RTPs)

The baseline does not generate any random TPs. However, using random patterns is very effective to reduce the runtime.

☑ -rtp v1 Start with RTPG, run PFS for each new test pattern until the fault coverage reaches a given value (hyperparameter).
Intuition: Consider how to select this hyperparameter. What are the implications of stopping RTPG early if random TPs could further improve FC? What if this threshold is set too high, making ATPG a better option? *This should lead you to the next heuristics.*

☑ -rtp v2 Begin with ATPG and continue until the improvement in fault coverage (delta FC) is below a specified value.
Intuition: This approach is sensitive to the quality of the random test pattern. Consider the impact of a single ineffective random test pattern on the decision to stop. *This should lead you to the next heuristics.*

☑ -rtp v3 Evaluate the improvement over the last K test patterns. If the average improvement does not meet a specified threshold, stop RTPG.
Intuition: If there are ineffective test patterns, why are we event adding them to our final results? *This should lead you to the next heuristics.*

☑ -rtp v4 Generate multiple random TPs and select the best one. We provided an example for the pseudo-code of such an algorithm.

Algorithm 2: RTPG v4: Selecting the best random test pattern

```
...
while not done with RTPG do
    FL: list of currently remaining faults
    ...
    newTPs ← empty lists
    detFaults ← empty lists
    for i ← 1 to Q do
        newTPs[i] ← RTPG
        detFaults[i] ← PFS(FL, newTPs[i])    # do not update FL
    best-idx ← argmax(detFaults)
    best-tp ← newTPs[best-idx]
    # add best-tp to TPs, run PFS, update FL
    # maybe calculating ΔFC to decide termination?
    ...
return TPs
```

2.3 ATPG – D-Frontier

The default method (baseline) is to select a node on the D-frontier without any specific order.

☑ -df nl/nh Choose the node from the D-frontier with the lowest/highest number.

☑ -df lh Select the node from the D-frontier that is at the highest level.

☑ -df cc Choose the node from the D-frontier with the lowest SCOAP observability.

2.4 ATPG – Justification

There is no justification process in PODEM, but in the presence of XOR/XNOR gates, the same heuristic can be used for selecting a likely value within Backtrace.

☑ -jf v0 Select the line with the lowest SCOAP controllability.

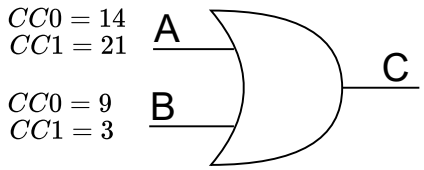

Figure 1: The heuristic utilizes the SCOAP metrics and suggests to use $B = 1$ to justify $C = 1$.

2.5 Fault order

You are required to develop two new heuristics in this category. Here are some suggestions:

☑ Starting with RFL, then targeting the remaining faults.

☑ Starting with the easier or harder faults. The definition of easy/hard faults is up to your discretion.

☑ Using level information may also be beneficial.

2.6 Test volume compression

Design a heuristic aimed at minimizing the number of test patterns within the ATPG part (not RTPG). One approach could involve transforming the X values in the output of ATPG algorithms into binary ones.

☑ While implementation is not included in the minimum requirements, you should provide intuition and design (either an algorithm or flowchart).

2.7 Fault simulation

Discuss the differences between using DFS and PFS in relation to our objectives, and provide experimental results for scenarios with low, high, and medium fault coverage values. If you had to choose one method for the entire process, which would it be?

☑ Low FC values (let's say below 20%)

☑ High FC values (higher than 95%)

☑ Middle FC values

3 Simulator and Commands

In this phase you will implement the TPG and slightly modify the TPFC command. All commands from the previous phases should be still functional in this phase.

3.1 DTPFC

You had already implemented the TPFC command, i.e. randomly generating test patterns, running fault simulation and calculating the fault coverage to produce a tpfc report. The DTPFC command is very similar, but reads the test patterns from a file instead.

```
[cmd$] TPFC <tp-fname> <freq> <tpfc-report-fname>
```

3.2 TPG

The general format of the TPG command is given below.

```
[cmd$] TPG <list of arguments> <alg> <output-fname>
```

While in the previous phases we specified the exact format of the commands (e.g. LSIM, PFS, DFS, DALG, etc.), you will have the freedom of designing your format based on the implemented heuristics. The commands should be completely explained in the README file of your GitHub repository.

```
[cmd$] TPG -rtpg v0 -df vL -jf m3 -fl v3 32 64 DALG res1.tp
```

In this hypothetical example, the TPG command is instructed to:

- Run the "v0" version (heuristic) of RTPG,
- Select from D-Frontier based on "vL" method
- Select from J-Frontier
- Fault list selection is based on v3 with two hyper-parameters set to 32 and 64

Once again, you can and should design your own structure of the command arguments. Make your own standard. One important note is to set default values, i.e. when no arguments are provided. The default value typically offers the baseline version of the ATPG heuristic.

Important note: For the final evaluation of your simulator, we need to run the TPG command with different arguments based on the README file documentation. Make sure the description is without error and self explanatory.

3.3 Simulator

Your simulator should accept a seed for "random generation functions" as an argument, defaulting to 658. This ensures that your outputs are consistent and reproducible, even when using RTPG.

```
$ ./simulator <optional:  int as seed for random generator>
```

4 Report Format

A Google Slide is provided for each group. Each heuristics should include these items:

- Intuition: explained earlier.
- Design: explained earlier, either in pseudo-code or by creating a flowchart using draw.io.
- Command: a simple example of the TPG command to run
- Results: Use various large circuits (e.g. c1355, c1908, c3540, c5315, and c6288). Don't forget to include or make reference to the results of the version you are comparing with, and our comparison is to use our objectives, which are: runtime, fault coverage, and the number of test patterns.
- Analysis: State if the results match your expectation or not, and what can be understood from the results. In some cases providing the TP-FC figures can be helpful.

5 Submission

Here is a summary of what needs to be submitted:

- Code on GitHub repository
- REAME file with the complete description of the arguments
- Report on the shared Google Slides

Good luck and see you on the demo day!