Politecnico di Torino

Politecnico di Torino

Corso di Laurea Magistrale in Computer Engineering
Indirizzo Automation and Intelligent Cyber-Physical Systems
A.A. 2021/2022
Sessione di Laurea aprile 2022

An Innovative Strategy to Quickly Grade Functional Test Programs

Relatori:
Prof. Quer Stefano
Prof. Bernardi Paolo

Candidato:
Niccoletti Alessandro

Abstract

Testing implies the evaluation of a device or a software against requirements gathered from the system specifications or the users and it is conducted at different level during the design process. This project presents a novel technique to quickly provide a preliminary grade to functional test procedures of various nature, ranging from Software-Based Self-Test to Burn-In Functional Stress and System-level Test. The method is based on the analysis of the functional program execution trace obtained directly from the chip by using a debugger.

Starting from the basic theory of dependencies between instructions inside an assembly code, this work aims to exploit them on the execution trace to evaluate its connectivity.

When a piece is produced, it has to be tested to ensure its reliability. Testing, in the case that is considered, means letting execute a "gold" piece (a piece that has been already tested and it can be considered working), extracting from it a register sign depending on every register value at the moment in which the sign is computed and, finally, using the "gold" sign as comparison with all other signs belonging to the pieces to be tested. If the sign of a normal piece does not match the "gold" sign, something surely went wrong; but what if the two signs match? It cannot be said that the piece is working only considering this result.

Whether the piece is working properly or not, depends also on the program that has been used to perform testing. Here is where instruction dependencies become useful. They are exploited to understand how the data are propagated through a particular execution of the "gold" piece: if the data propagation is very high, we can say that, in a probabilistic way, if an error occurs it should be propagated until the end (or until the sign is computed); by the contrary, if the data propagation is low, in case of something goes wrong the source of the problem could be lost.

A graph representation of the data flow is created and visited to identify specific instructions that could impact the final coverage.

The experimental results are carried out on an Automotive device manufactured by STMicroelectronics to demonstrate the effectiveness of the approach.

Contents

I. Introcuction 5

II. Background 10

A. Software-Based Self-Test 10

B. Test during Burn-In 12

C. System Level Test 13

III. The Proposed Methodology 14

A. Basic Algorithm 18

B. Optimized Algorithm 23

C. Load and Store Instructions 25

D. Branch Instructions 28

E. Multiple Destination Instructions 31

IV. The Tool 35

A. The Pipeline 36

B. Software Structure 41

C. The Parser 48

D. Graph Analysis 56

E. Practical Use of the Tool 61

V. Experimental Results 70

A. Case Study 70

B. SBST Grading 71

C. Operating System Grading for System Level Test 73

VI. Conclusion 73
VII. Acknowledgement 76

I. Introduction

Complexity is absolutely characterizing the current testing scenario. Very large heterogeneous systems-on-Chip (SoCs) undergoes an extremely complex manufacturing test process and are demanded to be absolutely reliable all along their useful life.

To overcome the issues related to the huge size is a major objective of the industry. While the structural methods like Built-in Self-Test and other scan oriented techniques still looks appropriate to screen out devices affected by coarse defect in the early test stages, the adoption of functional approaches is becoming recognized as an effective solution to catch residual marginal behaviors.

Nevertheless, Functional test strategies have become very popular in the last decade to self-test the device while it is working in its mission field.

Despite the attractiveness of using functional methods, the industry is often reluctant to explore this kind of test solution. In fact, structural methods are well supported by EDA vendors. The test set creation often requires just push-button effort to launch parallel Automatic Test Pattern Generation (ATPG) processes that quickly return patterns and achieved coverage at the same.

Conversely, the functional technique development is based on fault simulation. For example, a test engineer who develops Software-Based Self-Test (SBST) first writes and run the

functional program to check if code works (i.e., not falling into exceptions and reaching to the end). Then, he grades the functional program by using fault simulation tools.

Notoriously, a fault simulation process of a functional procedure is extremely time consuming, therefore the results can take from hours to days to be computed. This is a very strong bottleneck. The test engineer in charge of creating functional test libraries can wait lot of time and get unsatisfactory results: this could happen, for instance, if they omitted to exercise some functionality or if they have not propagated test results correctly into signatures.

Automated generation methods based on constrained ATPG and randomization have been proposed to enlighten functional test program production. Such methods autonomously create functional programs which are often looking like a bunch of operations sequentially executed. In this way, the generation efforts are mitigated, but additional debug investigations are introduced to find blocking issues that impact the coverage.

The problematic of grading functional procedures becomes insurmountable when moving at application level. Many companies suggest to use real application as a final test before shipping the component to market. This technique is called System-level Test (SLT) and it relies on booting an Operating System (OS) that then schedules application tasks to mimics the in-field behavior. To grade a SLT functional program is simply unfeasible due to the execution length of about tens of

milliseconds or more. Therefore, SLT is currently considered an holistic technique, supported by an industrial evidence that running an OS and some application still screen out some components.

In this project I propose a novel methodology that aims at quickly feedback about the test quality of any functional procedure. In particular, the illustrated strategy permits to quickly grade any functional program and to provide indication how the functional procedure is promising or not. If not, it helps to identify weak instructions or blocks of code that may be impacting on final coverage.

The proposed method takes very little time to provide results as it is not based on simulation or fault simulation, but exploits the silicon implementation of the chip. More in details, the method is composed of two successive steps:

1) The functional program under evaluation is run on silicon and the complete trace of the gold execution is dumped on file (e.g., the unrolled list of the executed ASM instructions);
2) The execution trace saved on file is analyzed by an algorithm able to investigate on data flow characteristics and dependencies (e.g., discriminating how operands are used along the program execution).

I have defined a new metric called "connectivity" which is computed by the mentioned algorithm. This metric tells whether the operands processed by the functional program are effectively transported to diagnostic points or compressed into signature.

The connectivity metric is a strong approximation of lower level metrics like the fault and toggle coverage. My intentions is not to replace the fault simulation as the vehicle to compute very accurate figures. Conversely, I suggest to use the connectivity metric to quickly get an early indication about the overall quality of the functional program. A low connectivity value means that some flaws may affect the program (i.e., previously computed results are overwritten along the program flow without being read or memory locations are not included in the signature).

I have experimented the proposed technique on an automotive device manufactured by STMicroelectronics, a medium sized device belonging to the SPC58 family and used in critical parts of the vehicle, like ABS and AIRBAG.

Experiments encompasses three functional testing contexts:

1) Development and analysis of Software-Based Self-Test routines for in-field testing;
2) Generation of Functional programs able to properly stress/test the device during the Burn-In (BI) phase;
3) Evaluation of System-Level Test functional applications including OS boot and tasks distribution.

In all cases, the connectivity metric is very useful to early refine the functional routines avoiding repeatedly running extremely long fault simulation process. Examples reported in the experimental results demonstrate that a low connectivity is always correlated to a low fault coverage value. Reference [1].

II. Background

Reliability of Automotive Chips is based both on a excellent manufacturing test and a powerful error detection ability in-field. Traditionally, structural test methods have dominated the vast majority of the manufacturing test process. Nevertheless, the final test stages before market are currently more and more oriented to functional methods, especially in the automotive segment. The most desirable in-field self-test method to rely today is running software test libraries.

These routines can be preempted and are ideal to be run at regular times (i.e., scheduled by the operating system). In this background section, I briefly survey on:

1) Software-Based Self-Test (SBST) techniques;
2) Functional Stress/Test during Burn-In (BI);
3) System-Level Test (SLT) methods.

A. Software-Based Self-Test (SBST)

The principle of software-based self-test (SBST) is to run functional test patterns, based on the processor instruction set, exploiting processor resources to test the processor itself and the components around it [2][3]. SBST is one of the strategies firmly included in the manufacturing flow of microprocessors. Industrial experiences, such as [3] and [4] have confirmed the suitability of

the methodology. In [5] an interesting case targeting a multi-core processor is presented. In that experience functional patterns are loaded in cache and applied to each of the 8 Core Processing Units belonging to a 4 GHz multi-core server in order to perform partial-good device binning.

SBST is also an emerging alternative for identifying faults during normal operation of the product, by performing on-line testing. Several reasons push this choice: SBST does not require external test equipment, it has very little intrusiveness into system design and it minimizes power consumption and hardware cost with respect to other on-line techniques based on circuit redundancy. It also allows at-speed testing, a feature required to deal with some defects prompted by deep sub-micron technology advent.

Such procedures are designed to activate possible faults, then compress and store the self-test results in an available memory space, or raising a signal when the test has not ended correctly. A test engineer working on the development of SBST programs can follow guidelines given in [5] to reach high coverage and mitigate fault simulation effort. Anyway, the job of writing SBST programs is often considered a little boring due to the very long fault simulation time as well as a bottleneck. The disappointment is fueled by the total absence of tools helping identify SBST program weaknesses that may prevent you to reach a very high coverage.

B. Test during Burn-In

Burn-In (BI) is mainly a stress phase designed to remove the infant mortality of defective SoCs [6]. The BI provides external stress such as the thermal stress, generated by a climatic chamber or by socket-level local temperature forcing tools, aging the circuit material [7]. Furthermore BI perorms and internal stress such as the electrical one generated by scan-based approaches [8], Built-In Self-Test (BIST) modules [9], or functional test programs [10], driving circuit nodes to produce a high internal activity.

The creation of functional stress routines that produce the proper electrical activity is easier than the creation of SBST programs, which target the fault coverage. Furthermore, this creation process is often automated. Randomize tools or evolutionary engines are able to produce high quality stress patterns with limited human efforts [11].

Stressing a circuit by functional procedures is crucial and it is the first objective of the generation process. Anyway, also reaching a certain coverage is possible during BI and it is a desirable side effect. Test During Burn-In (TDBI) permits to retrieve information during the stress phase.

To maximize the coverage of TDBI stress/test procedures, the generation process is much more effective if the code templates or macros (i.e., ASM or C/C++ code) is granting propagation capabilities to check-points or signatures.

C. System Level Test

System-Level Test (SLT) has emerged as an important additional test insertion in today's semiconductor life-cycle. It is run by the circuit manufacturer in the final stage of production or by the buyer of the circuit, e.g., an automotive Tier-1 supplier who will integrate the circuit into a product, as part of incoming quality control. SLT can also be used during the post-silicon characterization phase where a circuit's extra-functional properties are measured on a population of several hundreds or thousands "first-silicon" circuits [12].

Quite often, SLT consists in running functional applications that would mimic the behavior of the device shows in field. Booting an Operating System and running benchmark is recognized SLT option. Despite the promising opportunity of anticipating the next integration levels, a major concern of SLT is how to measure the fault coverage it achieves. Very long simulation times and prohibitive fault simulation campaign are considerable blocking points when you want to grade a SLT program.

III. The Proposed Methodology

The proposed methodology addresses a common weakness of the SBST, TDBI and SLT test procedure generation. All of them share the weakness of very long evaluation times. Simulations and fault simulations are the major bottleneck to the creation of high-quality functional programs in short time. Therefore, any help to reducing the number of fault simulations is fundamental.

The technique proposed in this project looks in this direction. It permits to preliminary grade the test abilities of a functional procedure without any simulation. Instead, the silicon implementation of the device is used to extract a trace of the functional execution using a debugger. In this way, we produce a gold functional execution dump which can be further analyzed.

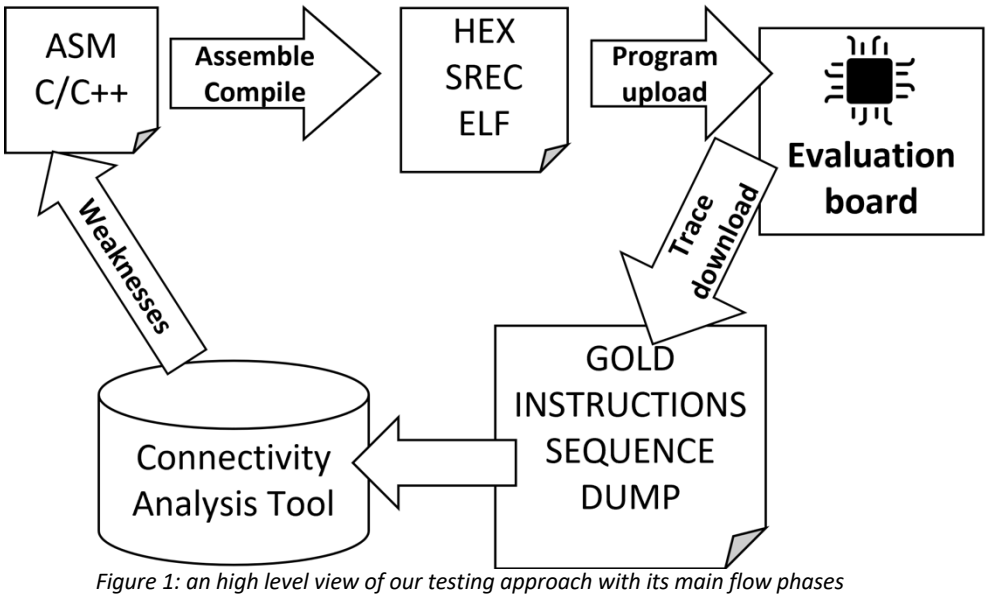

Figure 1: an high level view of our testing approach with its main flow phases

This dump phase takes very low time (up to seconds) compared with a logic simulation (up to hours) of the same functional behavior. The overall flow is illustrated in the following figure.

The gold trace retrieved from chip by a hardware debugger contains the linear execution of the functional program, e.g., the list of executed instructions, with loops unrolled and real register/memory values dumped instruction by instruction from the chip. Once it is produced, the trace can be parsed for being analyzed.

In the proposed approach, the trace is organized as a graph where every executed instructions constitutes a node. The graph build phase and the following visits are oriented to feedback the test engineer about the quality of the functional procedure. How the quality definition is computed from the graph do stem from the following theorem.

Theorem: Write-after-Write (WAW) instruction sequences occurring during the instruction flow over a shared addressable location cause the previously computed values to be overwritten and most likely lead to a loss of fault coverage. Conversely, Read-after-Write (RAW) instruction sequences propagate values along the execution flow, as they can reach
to an observation point, possibly leading to an increase of fault coverage.

The example that is presented next, can clarify the situation that we could address analyzing the trace:

$$\begin{array}{lll} 1 & add & r19, r18, 192 \\ 2 & sub & r6, r18, 55 \\ 3 & add & r6, r19, 35 \end{array}$$

example 1: WAW and RAW instruction sequences

If we take a look at the previous code, it appears that the test contribution of instruction 2 is vanished by instruction 3, due to the WAW sequence over register r6. Vice-versa, test contribute of instruction 1 is preserved by the RAW of r19 by instruction 3. Of course, the real fault coverage effectiveness of instruction 1 depends on the computed values (patterns) stored in source registers and can be determined only by fault simulation. As well, it is certain that the instruction 2 does not bring any benefit to fault coverage and it could be purged
from program or the program need to be fixed to unblock it.

Such a undesirable blocking situations due to WAW code scenario should never appear when a test engineer works in ASM language or he compiles C/C++ functional programs, while RAW are desirable and should always exist. Anyway, there are some case when WAW may appear in any case. To know about their presence in advance would be crucial at least in the following contexts:

- Randomization and evolutionary methods to automatically generate functional program may introduce WAW quite naturally. Therefore it would be beneficial to tune the generation to minimize the WAW and to reach a high coverage faster;

- To error is human and even typos may introduce unwanted WAW sequences when low-level programming in ASM language. To discover a mistake after a fault simulation is a very frustrating situation and a quick check before would greatly help to avoid any waste of time;

- It is often unfeasible to grade the fault coverage achieved by the very long program, such as the execution of benchmarks directly compiled in C/C++ language or Operating System boot and tasks. In this case, a check about testing ability potential weaknesses could be the only feasible measurement.

The analysis performed on the gold trace dump aims at determining whether a functional program includes any of these blocking situations. As shown in the next subsection, every instruction node of the graph is classified as blocked (black) or not (green) if it propagates or not the result of its execution according

to theorem 1. I define the novel metric called *program connectivity* computed as the fraction of blocked instructions over the total number of investigated instructions. A program showing very high or full connectivity can be considered as promising under the point of view of the potential fault coverage it could reach. Conversely, it is not a very good sign if the functional program shows a lot of blocked instructions and a low connectivity; such program need to be revised. The tool computing the connectivity metric also pinpoints the blocked instructions and can guide the test engineer to fix the proper components of the development flow.

A. Basic Algorithm

The proposed analysis flow is based on an algorithm able to identify blocking situations from the executed gold instruction trace, i.e., identify WAW which prevents information to forwarded along the program data flow.

Given that, the proposed algorithm elaborates the complete sequence of instruction executed from the beginning to the program end. It strongly differs from static and dynamic code analysis techniques used for Formal Software Testing and security assessment of functional programs [13]-[19]. These techniques take the program source code as the input of their investigation. Conversely, the proposed approach statically elaborates the real sequence of instructions executed by the system CPU.

As an example, a short loop based program may return a very long trace if the loop is extensively repeated. By the trick based on using the real silicon and a hardware Debugger, the uncertainty of formal software testing approaches is overcome and the gold instruction trace dump is obtained in very short time compared to simulation.

The following table reports a short snapshot of a gold execution trace dump. The first column reports the order of execution or cycle. Second and third columns report address and mnemonic code of the relative instruction. Finally SRC and DST columns pinpoint sources and destinations of every instruction in the dump.

Cycles	*Address*	*Intruction*	*SRC*	*DST*
1	*0x00000000*	*subfme r19,r6*	*r19, r6*	*r19*
2	*0x00000004*	*e_add16i r6,r19,35*	*r19*	*r6*
3	*0x00000008*	*e_add2i r6,r18,192*	*r18*	*r6*
4	*0x0000000C*	*e_add2i r6,r6,r1*	*r6, r1*	*r6*
5	*0x00000010*	*subfme r19,r6*	*r19, r6*	*r19*
6	*0x00000014*	*subfze r6,r3*	*r6, r3*	*r6*
7	*0x00000018*	*e_add16i r6,r19,35*	*r19*	*r6*

Table 1: Basic Execution Trace Dump

Based on this information, a graph is first built and then visited to compute the connectivity level of the analyzed functional program. As well, it can extract a list of critical instructions out of the executed code.

The base algorithm version is composed of three steps, as also visualized in the *Figure 2*:

- A Build phase;
- A WAW oriented-visit;
- A RAW oriented-visit;

The build phase, represented in *Figure 3*, is aimed at adding two types of edge to each instruction in the dump. If we look at the absolute left side of the next figure, there are two types of edges in the graph.

On the left part the RAW edges, that represent a read action performed by the lower node to the upper one. On the opposite side, the WAW edges: they are created if an instruction destination register is overwritten by another one whose instruction is placed in the dump after it.

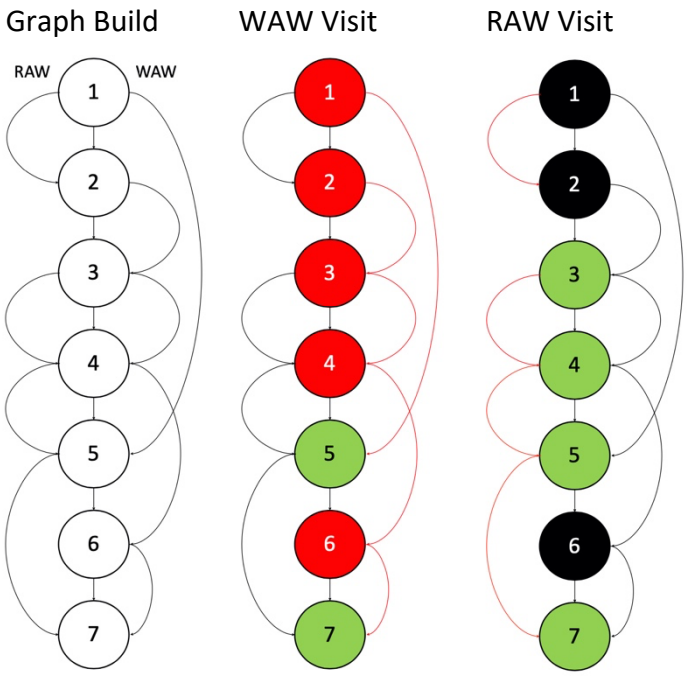

Figure 2: Basic Version of Graph Build and Visits

```
Build_Graph ()
 1: foreach (node in G) do
 2:     Search (G, node_index)
 3: end for
Search (G, node_index)
 4: foreach (n > node_index) do
 5:     if (source[n] == destination[node_index]) then
 6:         create RAW edge between node_index and n
 7:     end if
 8:     if (destination[n] == destination[node_index]) then
 9:         create WAW edge between node_index and n
10:         return
11:     end if
12: end for
```

Figure 3: Build graph function that elaborates the execution dump

Based on these two types of edge, the analysis continues with two successive visits. Both of them has linear complexity.

The first visit is not actually a visit: we are moving through the graph traversing the central edges corresponding to the sequential structure of the trace, but we call it visit for now to make better the idea. We will see in the next section that this function will be removed to optimize the process. During the visit, which pseudo-code is reported in *Figure 4*, the nodes are labeled red if they are WAW victim, green if not. Instruction 4 is red because it has an aggressor in instruction 6. Instruction 5 is green because the value it computes is never overwritten hence, it will reach the end of the program.

The second visit is based on traversing RAW edges to confirm or not the WAW blocking situation that is present at the moment on the red nodes. The pseudo-code of this visit is reported in *Figure 5*. Along the visit, a red node can turn into green if there is a RAW edge to connect it forward to a green instruction, also passing by other nodes not directly connected to

it. Conversely, the blocking suspect is confirmed and color updated to black. Instruction 4 is a case of red node evolving to green, while instruction 6 is confirmed blocked and becomes black.

Once the graph nodes are colored, the connectivity grade is measured. The illustrated scenario shows a 57.14\% of connectivity, because 3 out of 7 instructions are labeled as critical. In fact, the instruction 1, 2 and 6 will never contribute to fault coverage since their computational results are overwritten before being propagated anywhere else.

```
WAW_Visit ()
 1: foreach (node in G) do
 2:    if (node has WAW edge) then
 3:       node color = red;
 4:    else if (node does not have WAW edge) then
 5:       node color = green;
 6:    end if
 7: end for
```

Figure 4: WAW visit

```
Visit ()
 1: foreach (node in G) do
 2:    RAW_Visit (node_index)
 3: end for
RAW_Visit (node_index)
 4: if (node[node_index] color == green) then
 5:    return green
 6: else if (node[node_index] color == black) then
 7:    return black
 8: else if (node[node_index] color == red) then
 9:    foreach (RAW in RAW_edges[node_index]) do
10:       Returned color = RAW_Visit (RAW)
11:       if (Returned color == green) then
12:          node[node_index] color = green
13:          return green
14:       end if
15:    end for
16:    node[node_index] color = black
17:    return black
18: end if
```

Figure 5: RAW visit

B. Optimized Algorithm

As anticipated before, the WAW visit, in this phase, is removed to optimize the overall process of analysis. The work that was done, until now, by this visit, is now performed by the graph build process.

The reason of this change are multiple:

- The WAW edges were created only to verify their presence, hence they were never traversed by a visit. This was an unnecessary waste of memory;
- The set of nodes are scanned once by the graph build process, indeed the WAW visit could be incorporated inside the building process reducing the computation overload. Instead of linking the two nodes belonging to a WAW sequence the algorithm sets directly the node to red.

Looking at *Figure 2*, instruction 6 could be already set to black during graph build, because the destination is overwritten and none of the following nodes has a RAW dependency with it. Due to this fact, another role occupied by the build phase, besides the WAW visit inheritance, is to set black the nodes with the characteristics mentioned before. Thanks to this, the final RAW visit will have the probability to find more nodes that do not need a call to the visit function (the green and black nodes) and can be returned immediatly.

Figure 6 and 7 will better clarify the optimization performed.

```
Build_Graph ()
 1: foreach (node in G) do
 2:     Search (G, node_index)
 3: end for
Search (G, node_index)
 4: foreach (n > node_index) do
 5:     if (source[n] == destination[node_index]) then
 6:         create RAW edge between node_index and n
 7:     end if
 8:     if (destination[n] == destination[node_index]) then
 9:         if (RAW edge exists) then
10:             node[node_index] color = red
11:         else if (RAW edge does not exist) then
12:             node[node_index] color = black
13:         end if
14:         return
15:     end if
16:     node[node_index] color = green
17: end for
```

Figure 6: Optimized Graph Build

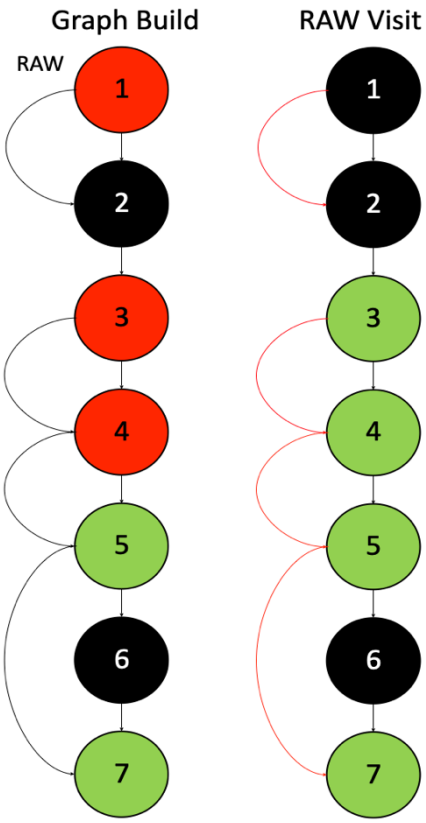

Figure 7: Optimized Version of Graph Build and Visits

In addition to what has been already said about instruction 6, also instruction 2 is set to black during the first phase.

First of all, comparing *Figure 7* with *Figure 2*, we will notice that the optimized one has to call at least 3 times the RAW visit function on the three red nodes. Conversely, the basic version has to call the visit at least 5 times, consuming unnecessary resources and time to find out a blocking situation easily findable before.

Then we can also see from the examples, that the results of the two proposed version correspond.

The RAW visit does not change, but could be also decided to include the WAW visit inside it, instead of in the graph build.

C. Load and Store Instructions

When a load or a store instruction is found in the code, the algorithm needs to consider them in a alternative manner with respect to the straightforward instructions that elaborate the data from register to register.

Therefore we propose slight a variation of the algorithm, where the location in memory is treated as a virtual register. To effectively implement this modification, the gold trace dump needs to be modified and extended to get register values to be used to compute the virtual register address in memory.

Cycles	*Intruction*	*SRC*	*DST*
1	*e_stb r0, 0(r1)*	*r0, 0, r1*	*mem(0+r1)*
2	*e_add16i r2,r0,35*	*r0*	*r2*
3	*e_ lbz r2, 0(r1)*	*mem(0+r1), 0, r1*	*r2*
4	*e_add2i r2, r1, r2*	*r1, r2*	*r2*
5	*e_stb r2, 0(r1)*	*r2, 0 , r1*	*mem(0+r1)*
6	*subfze r1,r3*	*r1, r3*	*r1*
7	*e_add16i r1,r0,35*	*r0*	*r1*

Table 2: Execution Trace Dump with Load and Store Instructions

The code snippet reported in *Table 2* includes load and store instructions from and to memory locations, in addition to arithmetic operations. For the sake of simplicity, all memory locations have a size of 1 Byte.

Figure 8 reports the corresponding graph build phase (left-hand side) and the final result as returned by the RAW visit (right). This example shows as memory addresses are handled the same way as registers are. To be more specific, instruction 1 stores a value in the location pointed by register r1. The same location is accessed by a the load instruction at position 3, therefore store instruction 1 is finally labelled as green. Also instruction 5 is marked green, due to the fact that the value written in memory is propagated until the end of the execution.

The previous can be said if we decided to compute the signature including also memory virtual registers. However, given that is not an easy task to include them inside the signature, if we decided to consider only real registers, the node 5 would be

colored as black. This is given by the fact that the information about memory locations does no longer gives contribute to the final coverage.

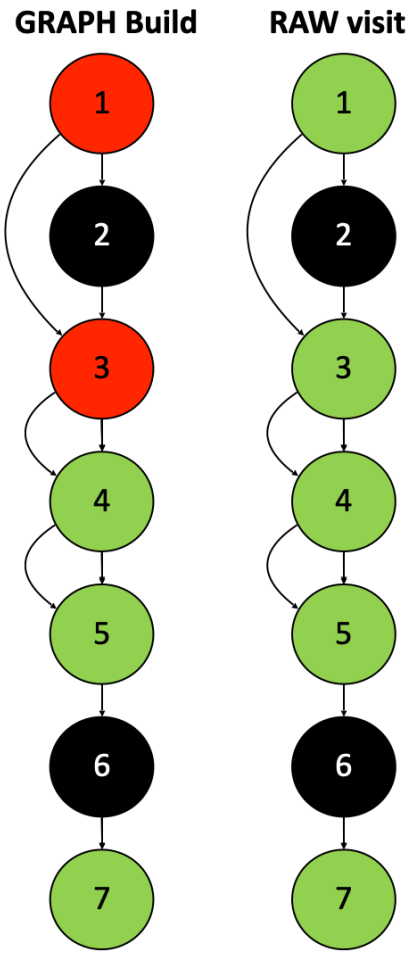

Figure 8: Optimized Version of Graph Build and Visit considering also Load and Store Instructions

D. Branch Instructions

Another category of instructions that is not considered in the basic version of the proposed algorithm is branch instructions. These are quite difficult to elaborate, given that an issue modifying a branch decision leads to a strong modification of the gold flow.

In order to classify branches as green or black, we modified the algorithm in the following directions:

- During the regular visits, conditional branches are left undecided and temporary colored orange;
- An additional visit is performed to possibly classify them as green or black graph nodes.

In principle, the additional visit tries to understand if the incorrect execution of a branch is leading to a different signature. Anyway, just some cases can be resolved by exclusively looking at the gold trace dump. The algorithm can color the branches in green or black color only in the cases where the alternative address to the one reached by correct execution of the branch is found among the instructions that follow to the investigated branch in the code.

The code reported in *Table 3* includes a branch generated by a simple if--then-else construct, while in *Figure 9* is reported graphically the same example.

Cycles	Address	Intruction	SRC	DST
1	0x00000000	e_li r0,0	/	r0
2	0x00000004	cmpl r0,1	r0, 1	CR
3	0x00000008	e_ beq jump	CR	/
4	0x0000000C	cmpl r0,0	r0, 0	CR
5	0x00000010	e_add16i r1,r2,r3	r2, r3	r1
6	0x00000014	jump: e_add6i r1,r0,r1	r0, r1	r1

Table 3: Execution Trace Dump with Branch Instructions

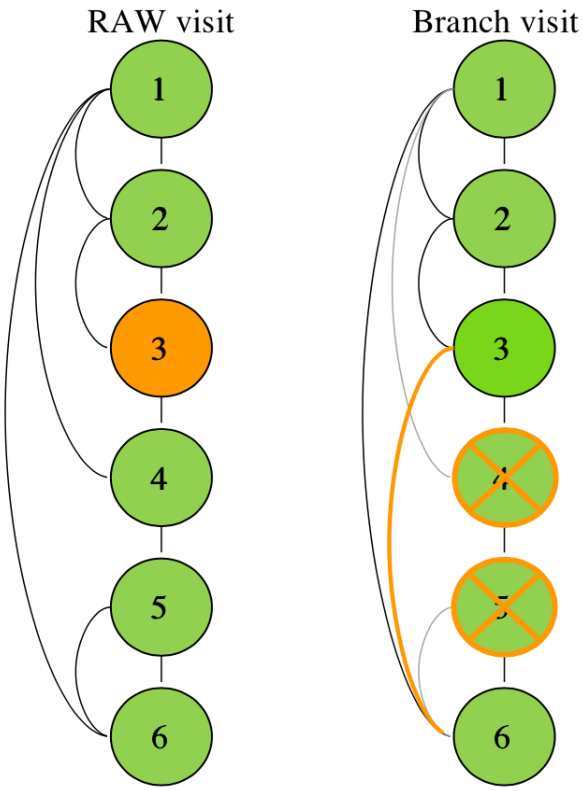

Figura 9: Example containing Branch Instructions Analysis

The previous figure reports the analysis in which our basic visit, reported on the left-hand side, leaves the branch node undecided. Once done that, the second visit evaluates the signature under the hypothesis of taking the wrong branch path.

In this specific case, the instruction that would be reached if the branch decision is wrong is included within the set of instructions that follow the branch itself. A new orange colored edge is added in the figure to highlight this situation and it clearly appears that some instructions will not be executed if the branch is wrongly done (taken if it was not taken, and vice-versa).

In the current scenario, the instructions that are skipped are green, therefore it results that the final registers value may change and the signature value potentially compromised. Indeed, the branch node is labeled with green color.

Conversely, if all skipped instruction were black, meaning they are not contributing to the signature value, the branch would be colored black as illustrated in *Table 4* and *Figure 10*.

Cycles	*Address*	*Intruction*	*SRC*	*DST*
1	*0x00000000*	*e_li r0,0*	*/*	*r0*
2	*0x00000004*	*cmpl r0,1*	*r0, 1*	*CR*
3	*0x00000008*	*e_ beq jump*	*CR*	*/*
4	*0x0000000C*	*e_add16i r1,r2,1*	*r2*	*r1*
5	*0x00000010*	*e_add16i r1,r0,1*	*r0*	*r1*
6	*0x00000014*	*jump: e_add6i r1,r0,r2*	*r0, r2*	*r1*

Table 4: Execution Trace Dump with Branch Instructions Blocking Version

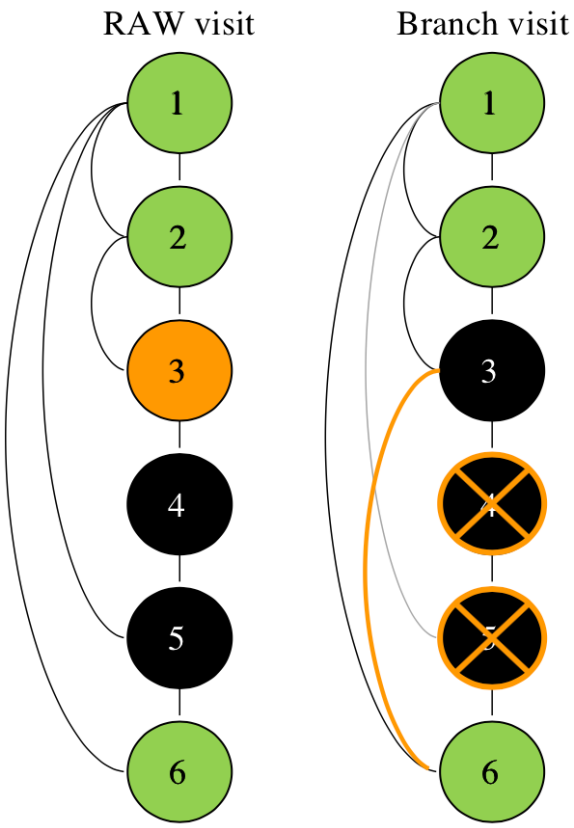

Figure 10: Example containing Branch Instructions Analysis Blocking Version

E. Multiple Destination Instructions

In this section, is presented a new version of the algorithm that take into account also instructions that could have more than only one destination. Now the main difference is that the color is not anymore assigned to the node, but to the single destination. The reason is that every destination belonging to a node could have different colors, so they could bring different contributes to the final coverage.

In the next *Table 5* and *Figure 11* is reported an example taking into account multiple destinations. To better approach for the first time this new method of analysis, it is represented without the optimization of the WAW visit.

Cycles	*Address*	*Intruction*	*SRC*	*DST*
1	*0x00000000*	*subf r0,r1*	*r0, r1*	*CR, r0*
2	*0x00000004*	*e_add16i r1,r0,35*	*r0*	*r1*
3	*0x00000008*	*e_add2i r1,r0,192*	*r0*	*CR, r1*
4	*0x0000000C*	*e_add2i r1,r1,r1*	*r1*	*CR, r1*
5	*0x00000010*	*subf r0,r1*	*r0, r1*	*CR, r0*
6	*0x00000014*	*subf r1,r2*	*r1, r2*	*r1*
7	*0x00000018*	*e_add16i r1,r0,35*	*r0*	*r1*

Table 5: Execution Trace Dump with Multiple Destination Analysis

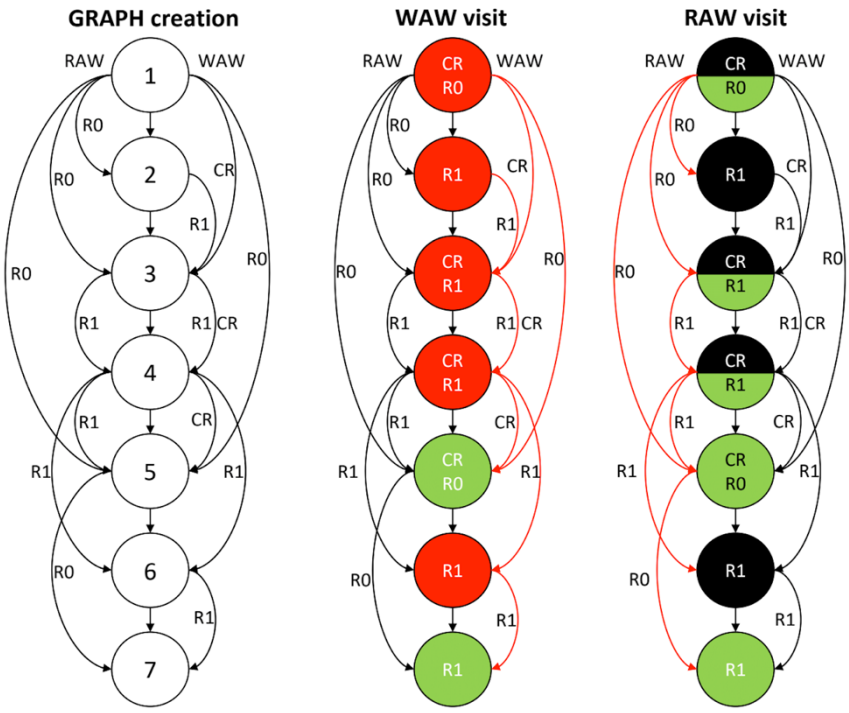

Figure 11: Example containing Multiple Destination Analysis

In the practice, also the analysis with multiple destination is performed optimized without WAW visit and WAW edges. In the next *Figure 12* and *Figure 13, indeed,* are reported the two pseudo algorithm of graph build and RAW visit, in the case of multiple destination.

```
Build_Graph ()
 1: foreach (node in G) do
 2:     foreach (destination in node) do
 3:         Search (G, node, destination)
 4:     end for
 5: end for
Search (G, node, destination)
 6: foreach (n > node in G) do
 7:     if (source[n] == destination) then
 8:         link destination with source[n]
 9:     end if
10:     if (destination[n] == destination) then
11:         if (destination is never read) then
12:             destination color = black
13:         else if (destination is read) then
14:             destination color = red
15:         end if
16:         return
17:     end if
18: end for
19: if (destination never found) then
20:     destination color = green
21: end if
```

Figure 12: Multiple Destination Graph Build Optimized

```
Visit (G)
 1: foreach node ∈ G) do
 2:    RAW_visit (G, node)
 3: end for
RAW_visit(G, node)
 4: foreach (Destination in Node) do
 5:    if (Destination color == green) then
 6:       return_color = green
 7:    else if Destination color == black) then
 8:       return_color = (return_color == black) ? black : green
 9:    else if (Destination color == red) then
10:       foreach (RAW edge i) do
11:          Returned color = RAW_visit (G, RAW edge[i])
12:          if (Returned color == green) then
13:             Destination color = green
14:             return_color = green
15:          else if Returned color == black) then
16:             Destination color = (Destination color != green) ? black : green
17:             return_color = (return_color == black) ? black : green;
18:          end if
19:       end for
20:    end if
21: end for
22: return (return_color)
```

Figure 13: Multiple Destination RAW Visit

Multiple destination analysis could be considered the last of the consecutive steps we have made until now. It is not an isolated case of analysis, indeed most of the instructions encountered in a real-life situation has more than one destination, including some of the branch instructions and load/store ones.

This themes will be discussed better in the section completely dedicated to the algorithms and software structure.

IV. The Tool

Until now the focus has been put on the general method of analysis, to have a first contact with this techniques, without entering too much into details. This chapter is dedicated more on technical aspects of the overall organization of the work, from the extraction of the trace by the chip, using the debugger, passing through the implementation and the algorithms, up to finally the computation of connectivity.

In this section we will deal with the following main point:

- The pipeline, representing the overall work-flow from the beginning to the end of the analysis;
- The software structure, in which the main class are presented and described, with all their attributes and methods;
- The parser: this is the first macro part of the project. It takes as input some file extracted, by debugging, from the gold execution and produce a graph file;
- The second macro part of the project: the graph analysis. It receives from the parser the graph file just created and performs graph build, all the visits and the final analysis to compute the connectivity;
- The algorithms. In this section the algoritms presented in the chapter III "The Proposed Methodology" will be seen more closely.

A. The Pipeline

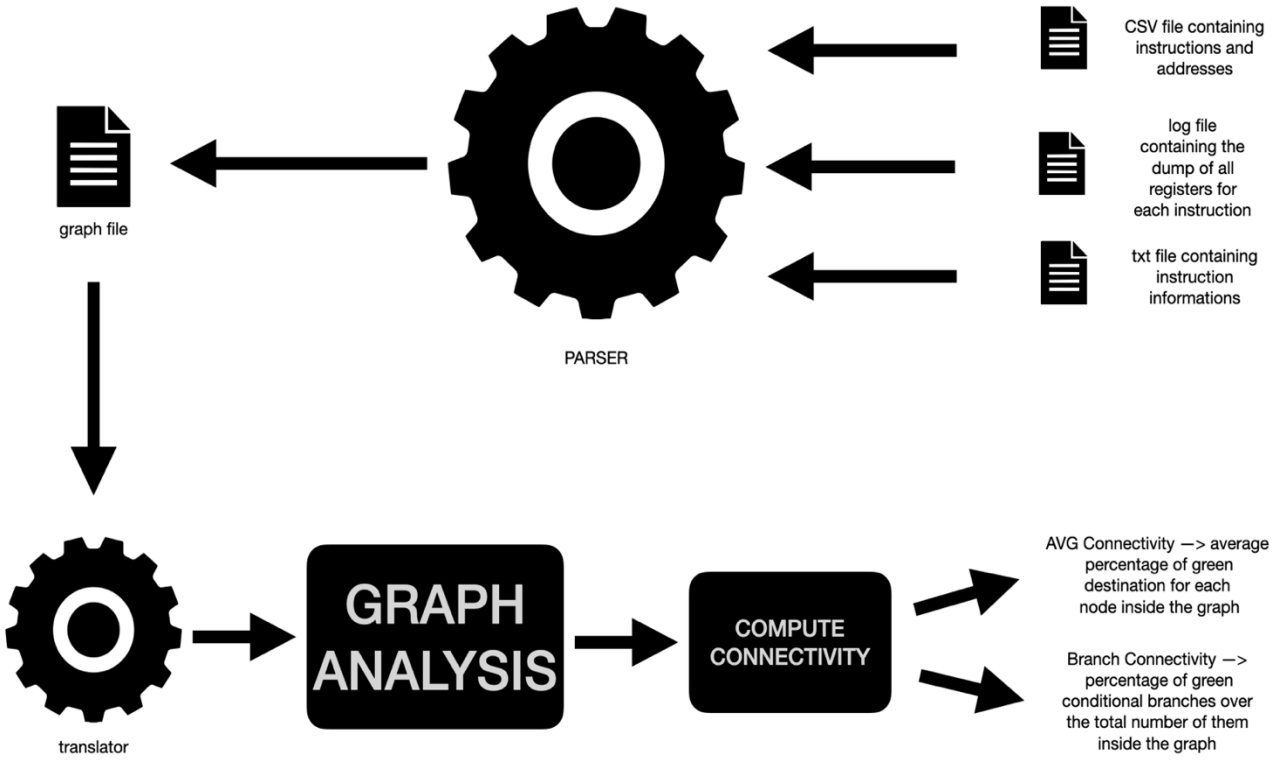

Figure 14: Overall Organization of the Work-Flow

In the previous *Figure 14* is represented the overall organization of the work-flow. At the top right corner we can see that the starting process, called *parser*, receives as input three different files.

Let's analyze them more closely:

- *instructionInfo.txt*: this file contains the lists of all instructions needed by the analysis. For each line, associated with one instruction, are reported, in the following order and separated by a white space, some essential information:

1. A string representing the instruction type in the set {A, B, BC, L, S, SP}, where A stands for Arithmetical, B for Branch, BC for Branch Conditional, L for Load, S for Store and SP for Special;
2. A string representing the name of the instruction;
3. A string composed by two part: the destinations part and the sources part. Each of them is separated by a comma and contains, at the beginning, the number of destinations/sources and then a list of string of three possible types: a number if the instruction has an explicit parameter in that position, a string if the instruction has an implicit parameter corresponding to that string, a string, starting with "mem" if the instruction performs an access in memory, containing the indications to compute the base address and the number of location to be accessed.

- *regFile.log*: this files has been created during the debug of the chip. It contains, for each line that is associated with an instruction, the list of all registers followed by their actual value in hexadecimal base. It is predominantly exploited by Load and Store

instructions to compute the address of memory locations to be read or written. This file could have very large dimensions, depending on how many instruction the gold device has executed.

- *traceFile.csv*: it is the execution trace dump exctracted directly from the chip in a csv format. At the very beginning it is given in a very rough form, with a lot of unnecessary information inside to be filtered and a structure that does not follow a very straight shape. This is why when we have to deal with trace of thousands or millions of instructions, this file could be very large and tough to manage. Therefore traceFile.csv has to be filtered and the information extracted from it are the instruction address, the instruction name and the explicit operands, if they are present.

In the next three figures (*Figure 15, figure 16* and *figure 17)* is given an example of all the three input files of the parser phase.

```
A se_srawi 2 XER 1 , 2 1 2
A se_srw 1 1 , 2 1 2
A se_srwi 1 1 , 2 1 2
S se_stb 1 mem(2+3) , 3 1 2 3
S se_sth 1 mem(2+3)*$2 , 3 1 2 3
S se_stw 1 mem(2+3)*$4 , 3 1 2 3
```

Figure 15: Piece of File instructionInfo.txt

```
R0=000000FF R1=40077C80 R2=C3C3C3C3 R3=F0F0F0F0 R4=B83FC74D R5=00000000
R6=40000000 R7=FFFFFFFF R8=FFFFFFFF R9=FFFFFFFF R10=FFFFFFFF R11=FFFFFFA4
R12=E21C6B51 R13=FFFFFFFF R14=FFFFFFFF R15=FFFFFFFF R16=FFFFFFFF R17=FA690000
R18=FFFFFFFF R19=00000104 R20=FFFFFFFF R21=00000033 R22=FFFFFFFF R23=FFFFFFFF
R24=FFFFFFFF R25=FFFFFFFF R26=FFFFFFFF R27=00000000 R28=F34203D7 R29=FFFFFFFF
R30=00FEE210 R31=26F54387 PP=00FE60A4 IP=00FE60A4 XER=20000000 CR=80000000
CTR=00000000 LR=00FE5E92 SPRG0=00FEE210 SPRG1=E2A7AF3B SPRG2=60E6A2BE
SPRG3=7EA5CE47 USPRG0=B357E8B9 SRR0=00FE5FA0 SRR1=02061000 CSRR0=F74FF7F6
CSRR1=D78D9BDA MCSRR0=00FC8A90 MCSRR1=00021000 MCSR=00000000 MCAR=00000000
IVPR=00FC5300 ESR=00000000 DEAR=3F962B98 ECR=08000010 MSR=02061000 PVR=815FA000
PID=00000000 PIR=00000002
```

Figure 16: One Line of regFile.log

```
"B::ART.List"
"record","run","address","cycle","data","symbol","ti.back"
"T000000","SV:00FE5C04","asmstep",4803,"\\uCOSIII_app\main\main+0x24"
"se_li","r3,0x0",";","r3,0"
"B::ART.List"
"record","run","address","cycle","data","symbol","ti.back"
"T000000","SV:00FE5C06","asmstep",4826,"\\uCOSIII_app\main\main+0x26"
291,"OSIdleTaskCtr=2;"
"se_li","r6,0x2",";","r6,2"
"B::ART.List"
"record","run","address","cycle","data","symbol","ti.back"
"T000000","SV:00FE5C08","asmstep",70E8E008,"\\uCOSIII_app\main\main+0x28"
"e_lis","r7,0x40080000",";","r7,1074266112"
"B::ART.List"
"record","run","address","cycle","data","symbol","ti.back"
"T000000","SV:00FE5C0C","asmstep",54C7044C,"\\uCOSIII_app\main\main+0x2C"
291,"OSIdleTaskCtr=2;"
"..."
"*"
"*","*/"
"OSStart(&err);","/*","Start","the","multitasking","*/"
"while","(DEF_TRUE)"
";"
"return","err;"
"#endif"

352,"}"
"e_stw","r6,0x44C(r7)",";","r6,1100(r7)"
```

Figure 17: Piece of traceFile.csv

Looking more in details to *Figure 15,* the first line, for example, tells to the parser that it could encounter, during its execution, an arithmetical instruction named *se_srawi.* This kind of instruction has two destinations, register XER and the first explicit operand that appear in the instruction call, and two sources corresponding to the first two operands. If, instead, we

take a look at the last line of the example, it is a store instruction called *se_stw* and it has a memory destination, computed considering as base address the sum of the content of the explicit operands in position 2 and 3. Furthermore we have to consider how many cells of memory are addressed: in this case the instruction is storing a word, so it writes four location of memory, as reported after the closed bracket. We notice also that there are three sources, and they are the first three explicit operands.

In *Figure 17* it can be seen that the structure of the initial trace file is not very linear. However, analyzing it better, can be found very useful informations, like the SV field corresponding to the instruction address. We moreover find the necessary informations about the name and the operands of every instruction inside the gold execution dump.

After the parsing phase a file graph is produced, containing informations to build properly the graph structure. At the very beginning of this file, is reported the number of total nodes that the final built graph will have. Immediatly after, is written one line for each instruction inside the trace. The layout of every line is not fixed: it can be different depending on the type of instruction. In general, every line of graph file contains informations about destinations and sources: they are the most crucial and important informations of the overall analysis.

At this point of the pipeline, the graph file is given as input to a translator, that interpretes its content and store the informations about every node in an array, that mantains the sequentially order of instructions inside the trace.

Finally, the array is passed to the graph analysis process that builds the graph, perform RAW visit and, at the end, the Branch visit. Te overall process is completed by the computation of the connectivity metric in two different types:

- The AVG Connectivity: it is the average percentage of green destinations inside a node.
- The Branc Connecticity: this is, instead, a connecticity that take into account only the information about conditional branches. Indeed, it is the percentage of green conditional branches inside all the graph.

B. Software Structure

This section deals with the software structure organization and, more in details, the main classes shape, their attributes and their methods. The overall project is developed in C++ so, the following declarations are reported in a C++ code style.

I have defined three main classes to implement the pipeline introduced in the previous section:

- Class Parser: this class represents the parsing process, the first in the pipeline. It contains attributes and methods with the purpose of performing the parsing from traceCSV file to graph file.
- Class Graph: this second class is the abstraction of the graph analysis phase, including also the connectivity computation. Graph class methods can access both private method of Parser class and Node class, presented in a while.
- Class Node: this class wants to be the abstraction of every single node inside a graph. It stores important informations about destinations and node colors.

In the next *Figure 18* these classes are presented in a diagram form, with all their attributes and methods. The connection between different classes means that, following the arrow direction, a class can accede the private member of the pointed one. In the diagram are not only reported classes, in a continue-line box, but also the structures needed by some of them, inside a fragmented-line box.

In the discussion that follows, for sake of simplicity, every getter and setter methods are not reported. Furthermore, the considerations done are based on the hypothesis that the code version is the one considering every possible situation: basic version, Load and Store, Branches and multiple destinations.

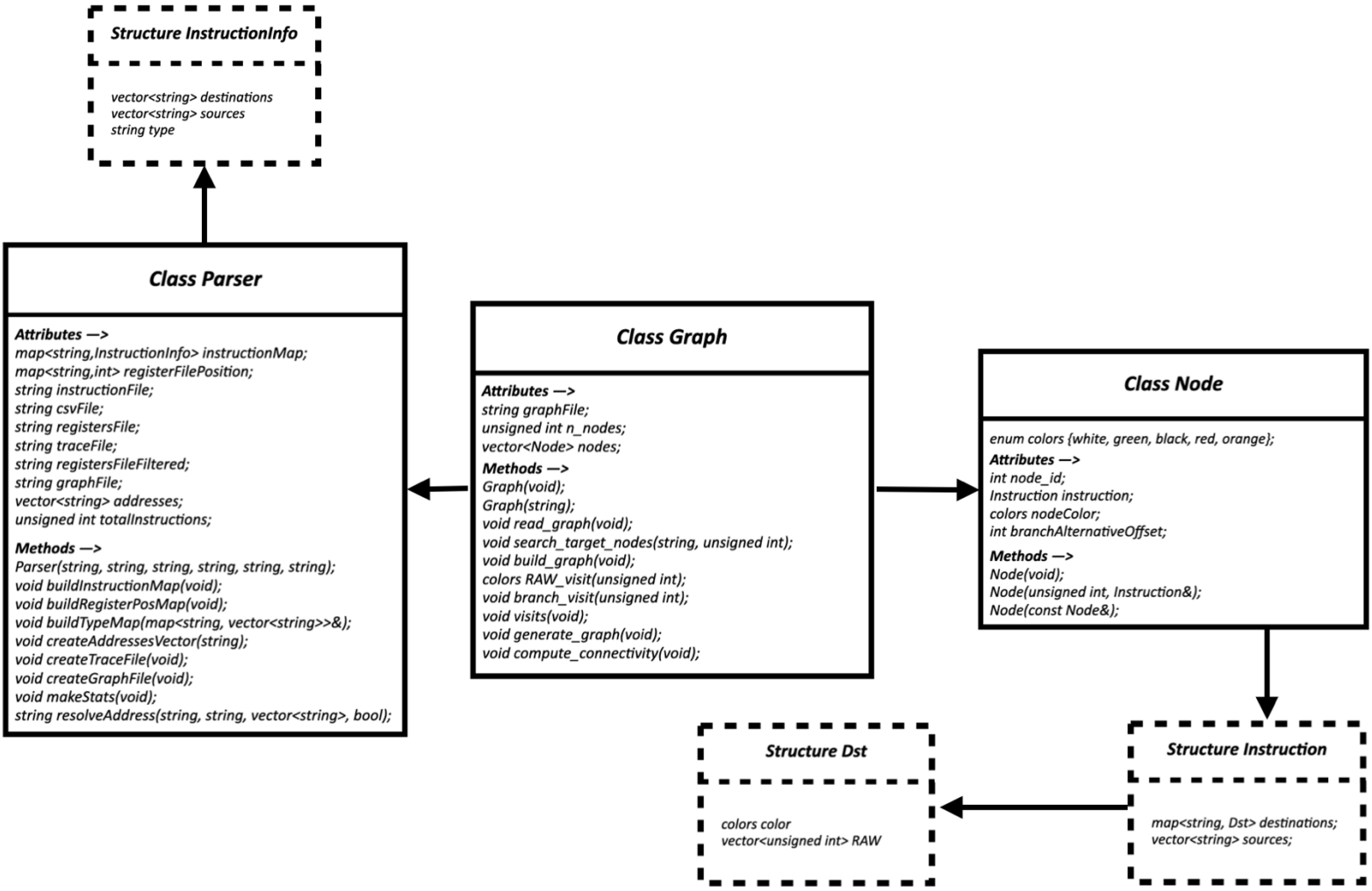

Figure18: Classes with their Attributes and Methods

Graph class has three attributes: a string containing the location of graph file to be read to build the graph, the number of nodes inside the graph and a vector of Node object. Its methods include two constructors, one of them a default one, a *read_graph* method implementing the *translator* process (*Figure 14)*, a *serach_target_node* and *build_graph* that together forms the graph build process, a *RAW_visit, branch_visit* and *visits* that make up the graph analysis phase and, finally, a

compute_connectivity method that produce the two connectivity index mentioned before. There is also a last method that is the *generate_graph*, useful to generate automatically a trace file, register file and graph file to test the functionalities of the tool, but keeping in mind that being generated automatically we will not have to take into account the connectivity evaluation.

Graph class is connected to both Node class and Parser class with an outgoing arrow, meaning that it can access private member of them. This is why Graph class stores a vector of Node object and it is very convenient to do it, to have more control. Another reason is that to generate automatically a graph, it is useful to easy access the parser methods and faster create trace, reg and graph files.

Node class give life to the most internal part of the graph. It is a very important container of informations, due to the fact that it stores the colors and conditional branches offset to perform branch visit. It contains a node ID, an instance of the structure representing every single instruction, a color representing the general coloration of the node (if at least one destination is green, this field is set to green) and an integer in which is saved, in correspondence with a conditional branch, the offset needed to jump to the alternative address and to peform branch visit. Regarding the methods, this class implements only one default constructor, one normal one and a copy contsructor. As we can see, always referring to *Figure 18*, structure *Instruction* is linked to the structure *Dst.* This is given by the the fact that

every destination inside an instruction, has its own color and its own list of RAW edges.

The last class to analyze is the Parser class. As it has been already said, parser class implements the parsing process. This brings a parser object to save, during its life, some information useful to help performing this task. The first attribute is a map that represents the *instructionInfo* file presented before. It stores the informations about destinations and sources of all possible instruction, avoiding that the reading of the file was done too much times. This map receive its fields from the *InstructionInfo* structure. Another attribute of the parser class is the *registerFilePosition* map: it stores the position of each physical register inside a single line of the register file, so that every access to the file is direct and it does not need any research.

The fields of the class that follow represent, in the order, the names of the locations of the files tath are needed by the parser: *instructionFile, csvFile, registersFile, traceFile, graphFile, registerFileFiltered.* The first three locations contain files that already exist while the others are locations that are going to be written by the parser. *Figure 14* shows that the parser has only one output, that is the graph file. This is actually true, but internally, parsing process, creates also other two files corresponding to *traceFile* and *registerFileFiltered*. The last two attributes are a vector containing the instruction addresses and the number of instruction inside the trace of the gold execution.

The methods of class Parser, are only listed next. However,

in the next section they will be discussed more in details. They are: a constructor, *buildInstructionMap, buildRegisterPosMap, buildTypeMap, createTraceFile, createGraphFile, makeStats, resolveAddress* and *createAddressesVector*.

Are represented below three pictures describing better the structure, in C++, of the classes presented before. These images include also methods and atrributes that do not appear in *Figure 18*, as setters, getters and printing functions, as well as could be possible that some name has been changed, to better describe *Figure 18*.

```
class Graph {

    string graphFile;
    unsigned int n_nodes;
    vector<Node> nodes;

    //hidden copy constructor
    Graph(const Graph&);

    //visit
    colors RAW_visit(unsigned int);
    void branch_visit(unsigned int);

public:
    //constructors
    Graph(void);
    Graph(string);

    void read_graph(void);
    void search_target_nodes(string, unsigned int);
    void build_graph(void);

    //visit call interface
    void visits(void);

    //graph generator
    void generate_graph(void);

    //getters
    string get_graph_file(void);

    //setters
    void set_graph_file(string);

    //connectivity
    void compute_connectivity(void);

    //overloading operator <<
    friend ostream& operator<<(ostream&, const Graph&);

};
```

Figure 19: Graph Header File

```
enum colors {white, green, black, red, orange};

struct Dst {
    colors color;
    vector <unsigned int> RAW;
};

struct Instruction {
    map<string, Dst> destinations;
    vector<string> sources;
};

class Node {

    int node_id;
    Instruction instruction;
    colors nodeColor;
    int branchAlternativeOffset;

    friend class Graph;

public:
    //constructors
    Node(void);
    Node(unsigned int, Instruction&);

    //copy constructor
    Node(const Node&);

    //getters
    int get_id(void) const;
    Instruction get_instruction(void) const;
    string get_color(colors) const;
    string get_nodeColor(void) const;
    int get_branchAlt(void) const;
};
```

Figure 20: Node Header File

```
struct InstructionInfo {
    vector<string> dst;
    vector<string> src;
    string type;
};

class Parser {

    map<string,InstructionInfo> instructionMap;
    map<string,int> registerFilePosition;
    string instructionFile;
    string csvFile;
    string registersFile;
    string traceFile;
    string registersFileFiltered;
    string graphFile;
    vector<string> addresses;
    unsigned int totalInstructions;

    friend class Graph;

public:
    Parser(string, string, string, string, string, string);
    void buildInstructionMap(void);
    void buildRegisterPosMap(void);
    void buildTypeMap(map<string, vector<string>>&);
    void createAddressesVector(string);
    void createTraceFile(void);
    void createGraphFile(void);
    void makeStats(void);

    //setters
    void setInstructionFile(string);
    void setCsvFile(string);
    void setRegistersFile(string);
    void setTraceFile(string);
    void setRegistersFileFiltered(string);
    void setGraphFile(string);

    //getter
    string getGraphFile(void);

    //utils
    string resolveAddress(string, string, vector<string>, bool);

    void printInstructionMap(void);
};
```

Figure 21: Parser Header File

C. The Parser

This section is completely dedicated to the parsing process description.

The files, directly extraced by tracing the gold execution and whose structures are shown in *Figure 15, Figure 16* and *Figure 17,* are directly fed to the parser. The first two operations to be performed, are constructing the two maps needed for the process : *instructionMap* and *registerFilePosition.* They are built calling methods *buildInstructionMap* and *buildRegisterPosMap.*

At this point, we are ready to switch on the parser and analyze the csv file with the aim of creating, at the end, the graph file. As it has been said before, this phase creates also two internal files: *traceFile* and *registersFileFiltered.* They hold the filtered version of csv file and registers file, initially given in a very rough shape.

The parser phase could be split into two main parts: the csv filtering, that produces a trace file given in input to the second part and the trace management, which in turn creates the final graph file.

The csv filtering takes as input *csvFile* and *registersFile.* Then the method *createTraceFile* is called to produce the final files *traceFile* and *registersFileFiltered*. In *Figure 22* is reported a chunk of *traceFile.* It is the *linear* version of the csv file, without unnecessary fields and, for each line is reported an instruction of

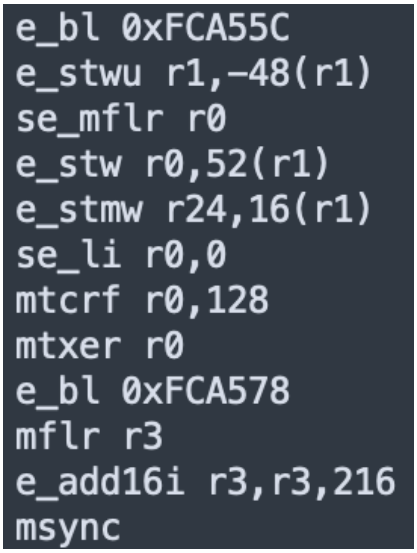

Figure 22: Chunk of traceFile after csv filtering phase

the execution dump with all its explicit operands. Instead, *registersFileFiltered* has the same structure of the initial *registersFile* but contains precisely one line for each instruction in *traceFile*. This filtering on *registersFile* is performed because during the phase of debugging, some line of the register dump is copied multiple times inside the initial file: if this fact is not correctly managed, could be cause of some bad behavior of the tool.

Trace management phase, conversely, take as input the just generated *traceFile* and *registersFileFiltered* and give them to the *createGraphFile* method that is called with the aim of creating the final *graphFile*. The next figure shows graphically the parser phase, underlining the split into the two main parts of the process.

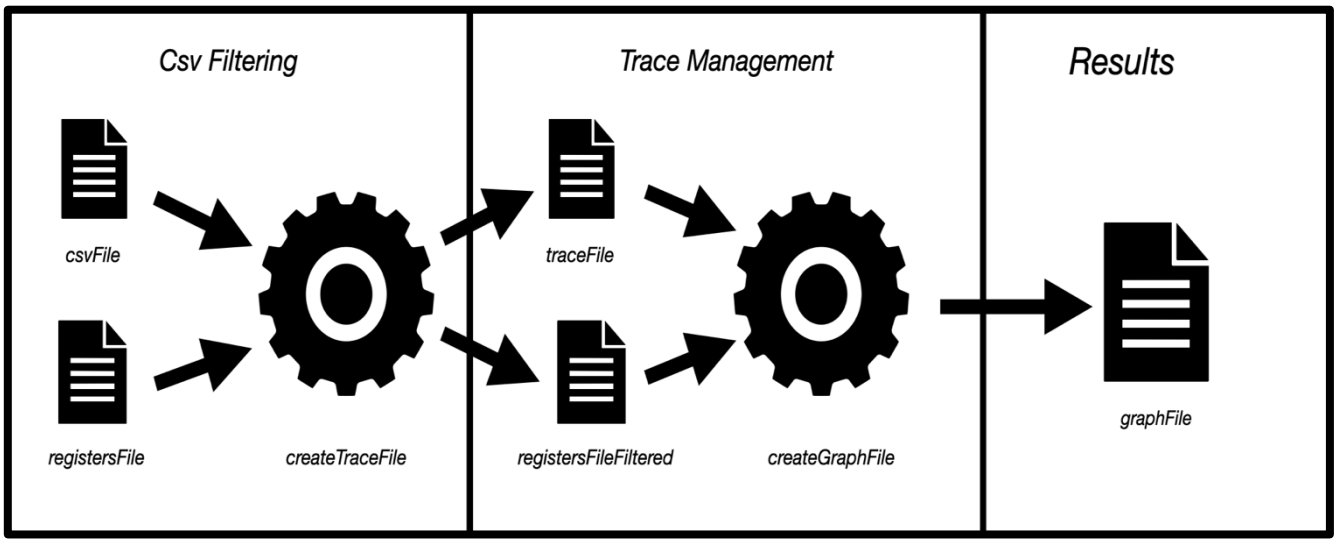

Figure23: Parser Phase with its Main Division

We have talked about the two methods that together form almost the overall process of parsing: *createTraceFile* and *createGraphFile.* Let us analyze them more closely:

1. *createTraceFile:* this method has two main objectives. The first one is to fill the *addresses* vector every time a valid instruction is found in the csv file with the proper instruction address. This work will result useful in the *createGraph* method to set the conditional branch offset. The second main objective of the method that we are analyzing, is to filter csv file and register file to write the useful informations into the trace file and the register fitered one.

2. *createGraphFile:* this procedure is definitely the most crucial of the overall process of parsing. It completely separates the trace dump from its initial structure and create ad hoc file, already presented as graph file, that is the only one needed to build completely the graph and analyze the gold trace execution. At the beginning of the file, the method writes the number of total instructions inside the trace. For each instruction in *traceFile* this method controls if it appears in the instruction map. Then, if the instruction is a conditional branch, is performed a control on the existence of both possible addresses to jump and, consequently to its result, there are two possibilities: if it is positive, is computed the distance from the current instruction to the alternative address to jump and it is stored in the graph file, otherwise the method writes a string *DANGER*, to signal that the alternative does not exist. At this point, the process continue writing on graph file a capital letter D, meaning that the list of destination is going to begin. The destinations are computed based on the informations stored in the instruction map, so every destination present in the map is stored in the graph file, from explicit operands to implicit ones. The memory operands are treated specially by a method of the class Parser that

will be analyzed next: *resolveAddress.* Currently let us consider that the memory operand will be reported in the graph file in the form *mem(base_address)num_location.* Now a comma followed by a capital letter S, signals that the list of sources in taking place. The proceedings used is the same as for destinations, since instruction map contains both informations about destinations and sources. All the informations stored in the graph file are separated by a white space.

At this point *graphFile* has been generated. As said before, it has a structure that completely separates the trace from its initial condition. For example, from now on, the instruction name is an information that is not anymore useful, reason for which it is not included in the file. This separation is wanted also in the sense of tool execution: the number of instructions at the beginning of graph file, let a user in possession of a graph file to execute the tool without calling the Parser methods and going directly to the analysis phase.

As we can see in the next *Figure 24* the graph file carries on only indispensable informations about analysis purposes: branch offset (if needed), destinations and sources. This is all the necessary to build the graph structure, perform the analysis and compute the connectivity to reach the gol of grading the gold execution trace dump.

```
D r5 , S 0 r3 mem(16669732)4
D CR r3 , S r4 r5
3 D , S CR 0xFE606E
D r3 , S 16842752
D r3 , S r3 257
D r2 , S 8 r1 mem(1074232456)4
D r13 , S 12 r1 mem(1074232460)4
D r7 , S 16 r1 mem(1074232464)4
D r8 , S 20 r1 mem(1074232468)4
D MSR , S r7
D , S
```

Figure24: Chunk of Graph File

The first line describe an instruction with one destination, corresponding to register r5, and three sources: an immediate with value 0, register r3 and 4 consecutive locations of memory starting at address 16669732. The third line is associated with a conditional branch, due to the fact that it starts with a branch alternative offset of value 3. This branch has no destinations and two sources corresponding to Condition Register CR and the immediate that represents the address jump of the instruction.

To better understand the offset mechanism, the number 3 means that in case of an error in the third instruction, the execution would jump directly to instruction 6. This, following the reasoning of section *III D. Branch Instructions,* is useful to perform branch visit, that controls if there is at least one green node between the branch instruction and the target offset intruction. For completeness, looking at the last line, this means that the instruction linked to it does not have any operand.

It has been mentioned before a method named *resolveAddress.* This method receives as input a memory expression directly from the instruction map and compute the base address and the number of consecutive location to be considered. This method is essential for Load and Store instructions, because it resolves all possible intruction involving accesses in memory, from a single Byte to a multiple word instruction.

The last three methods remained to analyze are not useful for the final computation of grading results. *buildTypeMap* method is called during the automatic generation process and it creates a map containing, for each type in the set mentioned before, the list of all instruction of that type. In this way it is easier to control the generation at different type percentage.

createAddressesVector is a method used only in a very particular condition: in the case in which the csv file is not given and in some way it is available a trace file , if we want to call the *createGraphFile* method we have to build the *addresses* vector that, usually, is created by the *createTraceFile* method. In this way the process can be continued also in the absence of one of the files inside the chain.

The last method is called *makeStats* and it performs a statistic on the type of instruction inside the *traceFile,* returning a percentage for each type on instruction. It also writes on a file, for each distinct instruction found in the trace, how many times they appear.

The next two *Figure 25* and *Figure 26* contain the pseudo code of the two main methods mentioned before: *createTraceFile* and *createGraphFile.*

```
createTraceFile (csvFile, registersFile)
1:  foreach (line in csvFile) do
2:      if (instruction address is found) then
3:          push_back address in addresses vector
4:      end if
5:      if (valid instruction is found) then
6:          set operands
7:          write instruction name and operands on traceFile
8:          write one line of registersFileFiltered
9:      end if
10: end for
```

Figure 25: Pseudo Code of createTraceFile Method

```
createGraphFile (traceFile, registersFileFiltered)
1:  foreach (line in traceFile) do
2:      if (instruction is not present in the instructionMap) then
3:          return error: "instruction not found"
4:      end if
5:      if (instruction is a conditional branch) then
6:          if (both addresses related to the branch exist) then
7:              write branch alternative offset at the beginning of graph file line
8:          else
9:              write DANGER at the beginning of graph file line
10:         end if
11:     end if
12:     set Destinations and write them in the graph file
13:     set Sources and write them in the graph file
14: end for
```

Figure 26: Pseudo Code of createGraphFile Method

D. Graph Analysis

After having performed the Parsing phase, we are in possession of a very precious file: the *graphFile.* As previously said, this file alone is sufficient to build the graph structure and to do analysis to compute the connectivity metric needed to grade the gold execution trace dump.

This phase is diveded into three main parts:

- The translation part, in which graph file is taken as input by the *read_graph* method. This process stores in memory the informations about every Node in the vector *nodes* that is an attribute of the Graph class. In this phase the most important passes are basically two: the first pass coincides with the management of the attribute *branchAlternativeOffset,* set by the first part of the method, in that the value needed is situated at the very beginning of each graph file line (if corresponding to a conditional branch instruction). The second one, is the set up of all destinations and sources iside the destinations map of the class Node. Next, in *Figure 27*, is reported a pseudo code of the *read_graph* method, the one that could be considered as the main performer of the overall translation part.

- The build part: in this second section of the graph analysis phase, the main protagonist is the *build_graph* method. This procedure has already been analyzed in *Figure 3* with a basic representation of it and in *Figure 12*, in which the algorithm has its final shape, considering also the multiple destination mechanism.
- The visits part: this last segment of analysis is the real part in which the graph is visited and the informations elaborated to color every destination and every node inside the graph. The first visit (RAW one) performed, has been already commented in *Figure 5* (basic version) and in *Figure 13* in which also multiple destination are taken into account. The second visit is represented by *branch_visit* method: this method aims at visiting the graph, starting from the end, and finding out if, in case of a conditional branch instruction, the nodes bewteen it and the target node have at least one green node. If this happens the branch node will be colored as green because, in case of something went wrong, the final computed sign would be changed. If these nodes analyzed are all completely black, the branch node will be sign as black. Both RAW visit and branch visit are called by the interface method *visits.* In *Figure 28* is reporthed the *branch_visit* method.

```
read_graph (graphFile)
1:  foreach (line in graphFile) do
2:      create a Node object named current_node
3:      if (there is a string at the beginning of the line) then
4:          if (the string represents a number) then
5:              set branchAlternativeOffset to that number
6:          else
7:              set branchAlternativeOffset to -1
8:          end if
9:      else
10:         set branchAlternativeOffset to 0
11:     end if
12:     read destinations and sources and set them into current_node
13:     set vector nodes[i] = current_node
14: end for
```

Figure 27: Pseudo Code of read_graph Method

```
branch_visit (node_index)
1:  foreach (Node in Graph starting from the end) do
2:      if (Node color == orange) then
3:          foreach (Node bewtween node_index and (node_index+offset)) do
4:              if (Node color == green) then
5:                  Node[node_index] = green
6:              else if (Node color == black) then
7:                  Node[node_index] = (Node color != green) ? black : green
8:              end if
9:          end for
10:     end if
11: end for
```

Figure 28: Pseudo Code of branch_visit Method

The last two methods of the Graph class to be analyzed, are the *generate_graph* and *compute_connectivity.* The first one is a procedure that, as sais before, create a trace file and a registers file in a completely random way. It receives, by the user, the total number of instructions that he wants to put in the trace and the percentage, for each type of instruction. Then the method *createGraphFile* is called and a new graph file i created to let the tool perform the final analysis.

The *compute_connectivity* method, instead, is the final procedure to be called at the end of the pipeline. It take as input the vector *nodes* of the Graph class and computes the two, mentioned before, metrics of the connectivity: the AVG connectivity and the branch connectivity.

The AVG metric is computed, for every node, as the percentage of green destinations over the total number of them. Then, is performed the average of the percentages just calculated as the final result. This metric give us an idea of the total connectivity of the gold execution trace: the higher is the average obtained, the more is the goodness of the program associated with the analyzed trace.

The branch connectivity, conversely, is computed taking into account only nodes that contain informations about conditional branches. The final result is the percentage of green conditional branches over the total number of them. The result just obtained is an indication of how the branch mechanism is

working inside the specific execution of the gold device. The higher is the percentage obtained at the end, the more the process is spreading its informations, even in the case a possible error on a conditional branch occurs.

Next is reported a picture in which is shown the overall process of graph analysis.

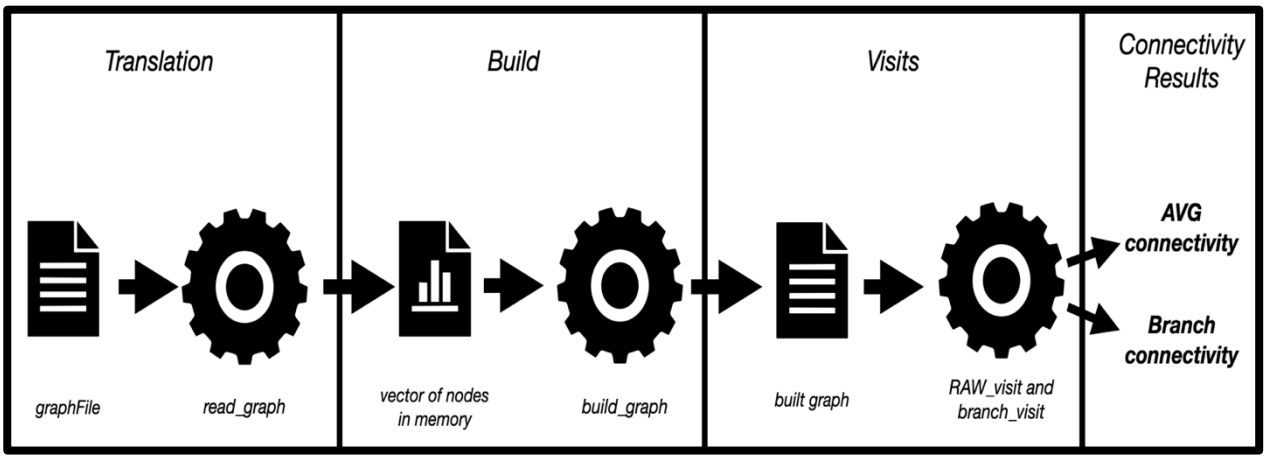

Figure 29: Graph Analysis Phase

E. Practical Use of the Tool

From a user point of view, this section could be useful as a guideline to understand better the overall process needed to start the tool, from the building to the execution.

First of all we have to build the executable file. I have defined some *make lables* that are handy to quickly compile and execute the program:

1. *make* or *make build:* they equally build the overall project, from parsing to analysis. This commands create also two directories: *build* and *stats.* The first could be useful to store the results of parsing analysis, passing the right path to the tool, and contains by default the executable program. The second one is used by Parser *makeStats* method to store informations about the number of distinct instructions inside the trace.
2. *make parser:* it is correspondent to the previous commands but it only consider the parsing part of the process. This could be useful to only test this part without compilink and linking the whole project. This time the directories created are *parser* and *stats,* with the same goal of the previous described.
3. *make debug:* this command sets up a debug session with lldb (can be changed modifying the type of

debugger in the makefile) of the whole project. *debug* and *stats_debug* are two directories automatically created.

4. *make debug_parser:* this is the corresponding version of the previous command but related to the parsing part of the tool. *debug_parser* and *stats_debug_parser* directories are created.
5. *make clean, clea_ parser, clean_debug, clean_debug_parser:* these are the commands to cleanup the directories created by the corresponding build lables.
6. *make run, run_parser:* these two commands start the program inside the *build* directory and the *parser* directory, respectively.

Let us consider we are calling the *make build* command. Once the tool is built, the *make run* command starts the tool. The main file of this project is organized as a choice menu with eleven different alternatives. The next *Figure 30* will better clarify this graphically. Subsequently, will be presented, command by command, a description of the tool functionalities.

```
<--MENU-->
0 --> Create a new instance of trace analysis tool
1 --> Build instruction map and register map
2 --> Filter csv file and create trace file
3 --> Translate trace file and create graph file
4 --> Build graph
5 --> Perform analysis on graph
6 --> Modify files (remember then to do the right action based on which file/s you have changed)
7 --> Print graph
8 --> Make statistics on trace
9 --> Compute connectivity
10 --> Generate a trace
11 --> Exit
TO SELECT FROM MENU, TYPE THE CORRESPONDING NUMBER
-->
```

Figure 30: Main Tool Menu

0. This command, as written in the previous figure, create a new instance of the tool. More in detail, it asks the user to insert the six file positions needed to create Parser and Graph objects, as it can be seen in the next *Figure 31.* The first three positions contain files already existing, while last three locations correspond to the files that the parsing process will generate: *traceFile, registersFileFiltered* and *graphFile*.

```
TO SELECT FROM MENU, TYPE THE CORRESPONDING NUMBER
--> 0
Insert the name and extension of instruction file --> /Users/alessandro/Desktop/trace_analysis/instructionInfo.txt
Insert the name and extension of csv file --> /Users/alessandro/Desktop/trace_analysis/traces/trace_stl_v2/trace.csv
Insert the name and extension of registers file --> /Users/alessandro/Desktop/trace_analysis/traces/trace_stl_v2/regfile.log
Insert the name and extension of trace file --> build/trace.txt
Insert the name and extension of registers file filtered --> build/reg.txt
Insert the name and extension of graph file --> build/graph.txt
```

Figure 31: Option 0 Execution

1. This second command, performs the building of the two main maps inside the Parser class: instruction map and register file position map. This execution has sense ony if called after an

instance of the tool has been generated but, even if it was called before, nothing bad will happen.

2. This option will tell the tool that we want to analyze the csv file and, consequently, the *createTraceFile* method will be executed. The first two file positions that was empty before, looking at *Figure 31* the positions *build/trace.txt* and *build/reg.txt*, are now occupied by the contents of *traceFile* and *registersFileFiltered.* This option can be selected only if the user is in possession of the csv file and the registers file. Furthermore it has sense to select this choice only after having created an instance of the tool and the two maps.
3. If this option is chosen, the toll will analyze trace file and registers file filtered with the aim of creating the final graph file: everything is done by the *createGraphFile* method. At the end of the execution even the file position *build/graph.txt* is not anymore empty: the content of *graphFile* has been copied there. This command can be selected even if we are not in possession of the csv file: only the trace file and the registers file filtered are needed. As it has been said in the section *C. Parser*, this particular case has the overload of the *createAddressesVector* method that has to set up the necessary structure to create the graph file. The only necessary condition to run properly this option, is that exists an instance of the tool and the maps are built.
4. At this point, there is an important change of role: the Parser object has finished its job and the overall work is passed to the

Graph instance. This option can be selected even if none of the previous commands has been already called before: the only thing necessary, in this situation, is to have a valid graph file. If we are in a situation in which we have used the Parser before this moment, the command will call directly *read_graph* and *build_graph* methods. Conversely, the tool will ask to insert the position of the graph file to be used. Then, first calls a constructor of the Graph class to setup a Graph object and then calls the same methods mentioned a little while ago.

5. Now in memory is saved the built graph with all the RAW edges set. If this command is selected, the *visits* interface methods is called. It, in turn, calls the RAW visit for each node in the graph and the branch visit for each contidional branch that is colored has orange. This option can be chosen only if the option 4 has been called before. This is given by the fact that it has no sense to perform a RAW visit or a branch visit without having built a graph before.
6. To this option is related an useful procedure to quickly change the path of one of the six file positions stored in the Parser and Graph instances. The tool will ask to insert one of the following string to identify which file is going to be changed: *instr, csv, reg, trace, regFilt, graph.* Then we have only to insert the new file position and everything is set up.

7. This command prints on the terminal a list of informations related to each node inside the graph structure. For each node is reported the node color, the branch alternative offset, every destinations with their colors and RAW edges, and every sources. The layout of the printing is reported in the next *Figure 32.*

```
Instruction number (383):
Node Color --> green
Branch Alternative Offset --> 0
Destination/s -->
Name --> R31 Color --> green RAW --> 392
Name --> XER Color --> green RAW --> 392
Sources -->
Name --> XER
Name --> R31
Name --> R5
```

Figure 32: Option 7 Execution: Printing Graph

8. This command is useful if we want to know more about the types distribution of the instructions inside the trace file. The method called is the *makeStats* belonging to the Parser class. The next two *Figure 33* and *Figure 34* report the two output of the process.

```
***STATISTIC RESULTS***

Arithmeticals --> 60.1415%
Unconditional Branches --> 11.7925%
Conditional Branches --> 0.235849%
Load --> 4.95283%
Store --> 2.12264%
Specials --> 20.7547%
Total Number of Instructions --> 424
```

Figure 33: Terminal Output of makeStats Method

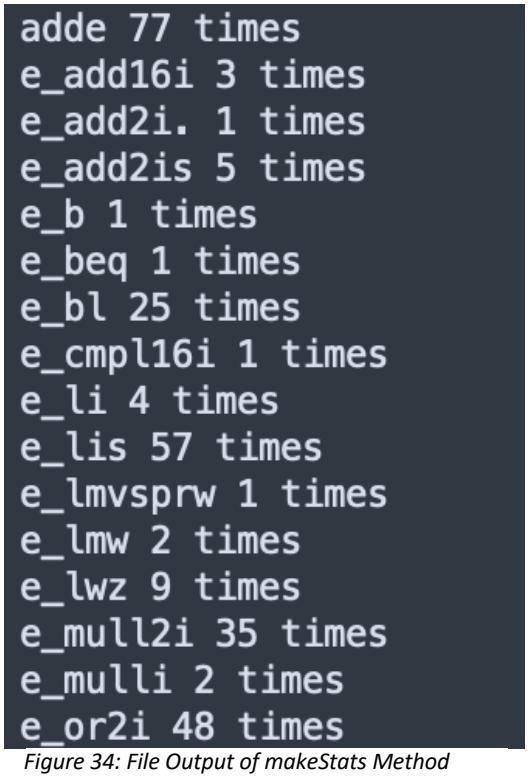

Figure 34: File Output of makeStats Method

9. This option, if chosen, computes the two connectivity metrics discussed before. Once the method *compute_connecticity* is called, the two expected output are printed on the terminal window. An expanded analysis of the graph is reported in a file created by the method: *connectivity.txt*. It contains, for each node inside the graph, the percentage of green destinations over the total of them. These information will be then compressed to create the AVG connectivity metric. Furthermore, inside the new created file, are stored more informations about the graph analysis just performed: for each distinct destination that appear in the graph, is computed the percentage of times that the single destination has resulted green over the total number of time that the

destination is encountered. In the next two figures are reported a chunk of terminal window in which AVG and branch connectivities are shown, and a piece of connectivity file that shows some registers that has been found as destinations inside the trace.

```
Average percentage of green destinations for each node --> 82.3113%
percentage of green conditional branches --> 100%
```

Figure 35: AVG Connectivity and Branch Connecticity Printed on Terminal Window

```
Register CR has 11.1111% of green destination inside the trace
Register CTR has 100% of green destination inside the trace
Register ESR has 100% of green destination inside the trace
Register LR has 10.7143% of green destination inside the trace
Register MSR has 33.3333% of green destination inside the trace
Register R0 has 36.3636% of green destination inside the trace
Register R1 has 100% of green destination inside the trace
Register R10 has 100% of green destination inside the trace
Register R11 has 100% of green destination inside the trace
Register R12 has 100% of green destination inside the trace
Register R13 has 100% of green destination inside the trace
Register R14 has 100% of green destination inside the trace
```

Figure 36: Piece of Connectivity File in which Some Destination Register is Reported

10. This command is selected to generate automatically a trace file, a registers file and a graph file. The only things that the tool asks to insert are: the file positions of instruction file, trace file to be generated, registers file to be generated and the graph file to be generated. Then the program wants to know if the generation should be automatic or not: the reason is that, at this moment, only the automatic generator has been implemented but there is the possibility to write the code for a guided version of the generator. Anyway, if we do not select the automatic version, the execution will return to the main

menu. Now the tool asks how many instruction will be in the trace and, for each type of instruction, what will be the presence percentage.

11. This option closes the tool and free all the memory allocated for the Graph, Parser and Node objects.

V. Experimental Results

The experimental results section includes many scenarios sharing the same automotive device. The common case of study is a SPC58 family micro-controller manufactured by STMicroelectronics. The chip characteristics are detailed in the next section *V. A.*

Section *V. B* illustrates how the methodology was used to assess SBST programs belonging to a Core Selt-Test library.

Section *V. C* reports about preliminary experiments for grading a Real-Time Operating system bootstrap and workload including SBST tasks.

A. Case Study

The case study is an automotive microprocessor belonging to the STMicroelectronics SPC58 family. The selected processor features multiple cores, many modules such as timers and communication modules. This device is used in critical parts of vehicles. The circuit counts around 20 millions gates and each of the CPU counts 1.5 million Stuck-at faults. *Figure 37* shows the experimental setup used to validate the proposed approach.

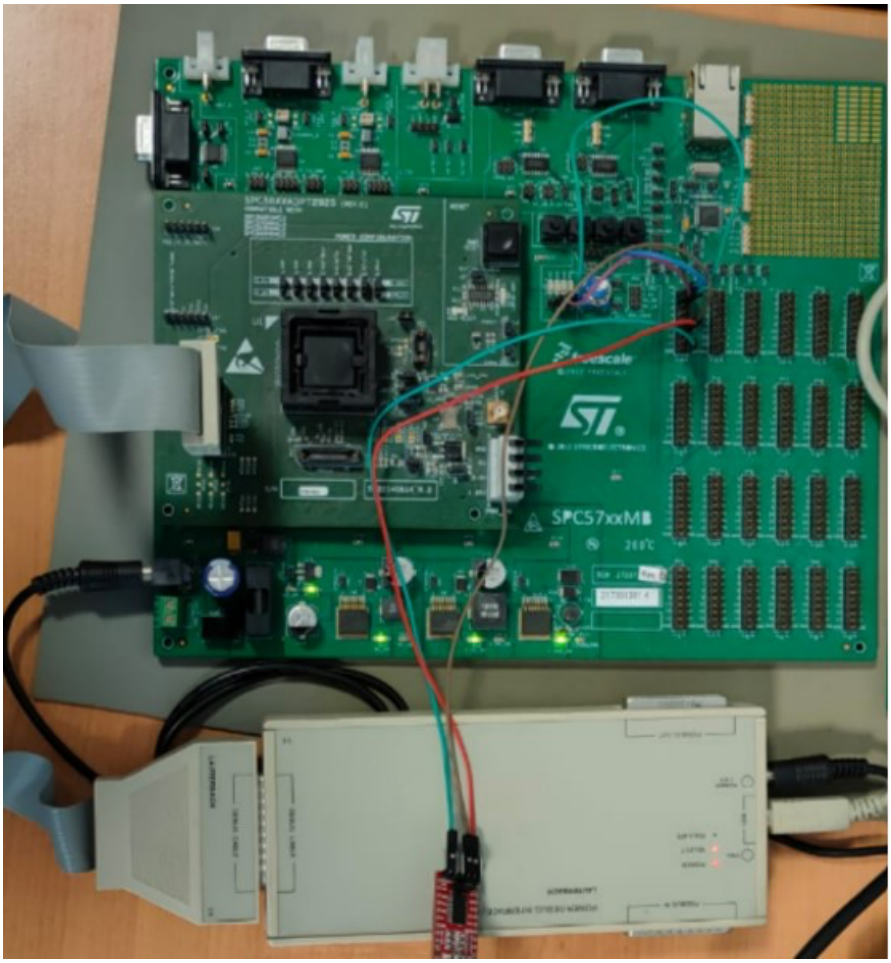

Figure 37: Experimental Setup including a Development Board and a Hardware Debugger

The debugger in the previous figure is used to extract the gold execution trace dump. Then, as we have widely said, the dump is passed to the tool to compute the connectivity metrics.

B. SBST Grading

The adoption of the proposed strategy is encouraged by the results that we achieved when grading Software-based Self-Test programs belonging to a Core Self Test library written by very

skilled test engineers. In particular we analyzed SBST programs targeting many components of the CPU core.

Table 6 reports about the investigated SBST programs, putting in comparison the fault analysis times and results, with our new method of preliminar grading.

Program Name	*Program size [Bytes]*	*Connectivity [%]*	*Grading Time [s]*	*Stack-at fault #*	*Stack-at fault coverage [%]*	*Fault simulation CPU time [m]*
Automatic trace 1M	*/*	*38.5*	*50*	*/*	*/*	*/*
Automatic trace 10M	*/*	*76*	*229*	*/*	*/*	*/*
Arithmetical adder	*1,635*	*58.5*	*40*	*19,760*	*92.6*	*1,463*
Count-zeros unit	*1,160*	*63.3*	*19*	*3,069*	*86.8*	*512*
Bit-wise logical	*680*	*97.9*	*11*	*2,828*	*95*	*339*
Branch target buffer	*4,476*	*40.6*	*/*	*19,990*	*71.2*	*4,009*

Table 6: Results from the Execution of the Tool

As we could expect, the first automatic generated trace has a lower connectivity with respect to the others written by qualified test engineers. The second has an high connectivity, but it is a matter of probabilities: being randomly generated is possibile that instruction has an actual high connectivity between them only by chance.

The previou table shows that, on average, our method takes seconds to perform the analysis. Comparing this time with the average fault simulation time, we obtain a very important result. I want to remember that this method is not a substutution of the fault analysis process: it is, instead, a quick way to decide

if performing fault analysis could have sense or not. But, it is also an indicator of which instructions and which destinations are giving more problems. For this reason, the toolt becomes useful to remove or correct them to better perform the testing phase.

C. Operating System Grading for System Level Test

Now we are going to try to grade some trace that are exctracted during the execution of the gold device during some task performed by operating system.

Program Name	*AVG Connectivity [%]*
Real Time Operating System (RTOS)	*79.8*
RTOS with enhanced stack frame	*100*

Table 7: Results from the Execution of the Tool with Real Time Operating System Trace

Table 7 shows some result of the analysis performed by the tool, this time with real time operating system trace. Also this kind of analysis took some milliseconds that, compared with fault simulation average times, is a very good results.

VI. Conclusion

Hardware testing needs to be thorough and precise because if a bug is missed, the cost of fixing it later is huge. Unfortunately, not even the world's brightest hardware manufacturers get their designs right the first time around. Moreover, testing may require costly and domain-specific testing devices and testing may be extremely time-consuming.

Dynamic taint analysis and forward symbolic execution are quickly becoming staple techniques in security analyses, such as malware analysis and vulnerability discovery. Nevertheless, there has been little effort to apply similar techniques to other domains.

In our work, we first run our functional program, we collect its complete trace, and we dump it on file. Then, we visit such a file and we build the data flow graph of our code. Finally, we visit such a graph to understand how operands are manipulated by the instruction flow. We present several possible versions of our application, starting from the trivial implementation and moving toward the most optimized one. We show how all these versions have reasonable memory and time cost. Moreover, we prove how the connectivity figures to tool return can indeed be used to appropriately grade the coverage on the chip.

This work demonstrates that, with the new metric we have introduced, is possible to give a preliminar grade to a test

program without performing any king of fault simulation that, as we have seen, is the strongest bottleneck for a testing process.

I think that putting beside this technique with a fault injection and other testing methods this work will result very useful and fast to use, but, above all, it will bring very relevant innovations.

Especially, being this method a novel technique, i think that a lot of improvements can be done, both in the performances and in the reasoning that lies behind it, especially about all the part related to the conditional branches and *branch_visit.*

VII. Acknowledgements

In this section i want to thank all the people that have worked with me during these months and are still involved in this project: my supervisors Stefano Quer and Paolo Bernardi to have chosen me to do this job and let me undesrstand better the testing world; Francesco Angione, who has created for me traces to be analyzed, extracted directly from the chip using the debugger; i want to thank Francesco, also for the experimental results related to fault analysis process; finally, Andrea Calabrese and Lorenzo Cardone for the support and the useful advice given in this time together.

References

[1] D. Appello, P. Bernardi, A. Calabrese, S. Littardi, G. Pollaccia, S. Quer, V. Tancorre, and R. Ugioli, “Accelerated analysis of simulation dumps through parallelization on multicore architectures,” in 2021 24th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS), 2021, pp. 69–74.

[2] A. Paschalis and D. Gizopoulos, “Effective software-based self-test strategies for on-line periodic testing of embedded processors,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 24, no. 1, pp. 88–99, Jan 2005.

[3] P. K. Parvathala, K. Maneparambil, and W. C. Lindsay, “Functional random instruction testing (frits) method for complex devices such as microprocessors,” United States Patent 6948096, 2005.

[4] I. Bayraktaroglu, J. Hunt, and D. Watkins, “Cache resident functional microprocessor testing: Avoiding high speed io issues,” in 2006 IEEE International Test Conference, Oct 2006, pp. 1–7.

[5] P. Bernardi, R. Cantoro, S. De Luca, E. S´anchez, and A. Sansonetti, “Development flow for on-line core self-test of automotive microcontrollers,” IEEE Transactions on Computers, vol. 65, no. 3, pp. 744–754, March 2016.

[6] T. Mak, “Infant mortality–the lesser known reliability issue,” in 13th IEEE International On-Line Testing Symposium (IOLTS 2007), 2007, pp. 122–122.

[7] M. Zakaria, Z. Kassim, M.-L. Ooi, and S. Demidenko, “Reducing burn-in time through high-voltage stress test and weibull statistical analysis,” IEEE Design Test of Computers, vol. 23, no. 2, pp. 88–98, 2006.

[8] A. Benso, A. Bosio, S. D. Carlo, G. D. Natale, and P. Prinetto, “Atpg for dynamic burn-in test in full-scan circuits,” in 2006 15th Asian Test Symposium, 2006, pp. 75–82.

[9] D. Appello, C. Bugeja, G. Pollaccia, P. Bernardi, R. Cantoro, M. Restifo, E. Sanchez, and F. Venini, “An optimized test during burn-in for automotive soc,” IEEE Design Test, vol. 35, no. 3, pp. 46–53, 2018.

[10] F. Almeida et al., “Effective screening of automotive socs by combining burn-in and system level test,” in IEEE International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS), 2019.

[11] P. Bernardi, A. Bosio, G. Di Natale, A. Guerriero, E. Sanchez, and F. Venini, “Improving Stress Quality for SoC Using Faster-than-At-Speed Execution of Functional Programs,” in VLSI-SoC: System-on-Chip in the Nanoscale Era – Design, Verification and Reliability, ser. IFIP Advances in Information and Communication Technology, T. Hollstein, J. Raik, S. Kostin, A. Tˇsertov, I. O’Connor, and R. Reis, Eds., vol. AICT-508. Tallinn, Estonia: Springer International Publishing, Sep. 2016, pp. 130–151. [Online]. Available: https://hal.inria.fr/hal-01675205.

[12] I. Polian, J. Anders, S. Becker, P. Bernardi, K. Chakrabarty, N. El-Hamawy, M. Sauer, A. Singh, M. S. Reorda, and S. Wagner, “Exploring the mysteries of system-level test,” in 2020 IEEE 29th Asian Test Symposium (ATS), 2020, pp. 1–6.

[13] P. Subramanyan, S. Malik, H. Khattri, A. Maiti, and J. Fung, “Verifying information flow properties of firmware using symbolic execution,” in Design, Automation Test in Europe Conference Exhibition (DATE), 2016, pp. 337–342.

[14] S. Malik and P. Subramanyan, “Specification and modeling for systems-on-chip security verification,” in Proceedings of the 53rd Annual Design Automation Conference, 2016, pp. 1–6.

[15] M. Hassan, V. Herdt, H. M. Le, D. Große, and R. Drechsler, “Early soc security validation by vp-based static information flow analysis,” in 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). IEEE, 2017, pp. 400–407.

[16] R. Drechlser and D. Große, “Ensuring correctness of next generation devices: From reconfigurable to self-learning systems,” in 2019 IEEE 28th Asian Test Symposium (ATS). IEEE, 2019, pp. 159–1595.

[17] W. Hu, C.-H. Chang, A. Sengupta, S. Bhunia, R. Kastner, and H. Li, “An overview of hardware security and trust: Threats, countermeasures, and design tools,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 40, no. 6, pp. 1010–1038, 2020.

[18] K. M. Alatoun, S. M. Achyutha, and R. Vemuri, “Efficient methods for soc trust validation using information flow verification,” in 2021 IEEE 39th International Conference on Computer Design (ICCD). IEEE, 2021, pp. 608–616.

[19] T. Alam, Z. Yang, B. Chen, N. Armour, and S. Ray, “Firver: Concolic testing for systematic validation of firmware binaries,” in 27th Asian and South Pacific Design Automation Conference (ASP-DAC), 2022, pp. 352–357.