Reporting test programs connectivity in a human-readable format

Reporting Test Programs Connectivity In A Human-Readable Format

Lorenzo Cardone
Dip. automatica ed informatica, Politecnico di Torino
Turin, Italy
lorenzo.cardone@polito.it

***Abstract*—Because of the increasing complexity of systems on chips, our ability to test their proper functioning is decreasing. Since safety standards, such as ISO-26262 [1], propose very time-consuming metrics to ensure the safety of the users, new techniques to achieve high fault coverage are making their way to speed up this process. In this article, we take a look at the connectivity metrics proposed last year by Francesco Angione et al [2]. We will discuss software that we have developed in order to expose to the user in a more readable and intuitive way the data produced by the analysis carried out through the previously presented methodology. We will take functional tests on a chip developed by STMicroelectronics from the SP58 family as a case study and show what information can be exposed to the end-user and what value each of these can provide.**

***Index Terms*—Burn-in test, functional test, heuristics, EDA, Automotive.**

I. Introduction

The need to manage increasingly complex systems leads to the creation of chips consisting of an increasing number of components. As a result, meeting security standards is becoming an increasingly time-consuming and, consequently, more expensive process than ever before. Although several companies have adopted functional approaches to testing their devices [3], [4], the approach requires significant effort, both in writing code and in simulation time. In an effort to alleviate the simulation time of functional methods Francesco Angione et al. [2] have proposed a new metric, called connectivity, that allows the programmer to quickly assess program quality by reducing the need for repeated fault simulations. In this paper, we will discuss methods for exposing to the end user as much of the data produced during connectivity computation as possible in order to make this tool as immediate and intuitive as possible.

In section II we will explain the original connectivity metrics to show its uses. In section III we will shortly explain how our program works and what metrics we are able to extract. In section IV we will examine every exposed value and we will discuss how it can be used to obtain the best results. In section V we will propose improvements to our current program and which other values can still be computed and shown to the user.

Thanks to (in alphabetical order): Francesco Angione, Paolo Bernardi, Andrea Calabrese, Stefano Quer, and the STMicroelectronics' team.

II. Background

Functional testing is a methodology that has seen ample use in testing, since it can be used both during manufacturing and after the deployment of the device to check its operation over time. The idea behind this method is fairly simple: it implies running a program and checking whether its outputs match the expected ones, usually defined by the designers or obtained running the same code on a $golden$ device, an already tested device of which we assume correct functioning. Functional tests are usually hand-written or generated by automatic tools, like microGP [5]. In either case, the standard to check their ability to cover any given defect is fault coverage. Fault coverage, although an extremely powerful metric, is very time consuming to compute and, as such, any help to reduce the requirement for repeated fault coverage is a welcome improvement.

During the 2022 International Test Conference [2], the connectivity metric was proposed to address this problem. This methodology is not suitable to completely replace fault coverage, but it proposes a fast and efficient method to grade the quality of a test program before the need to compute any fault coverage. The connectivity metric takes in account the flow of each instruction of the program during a sample execution. It grades each instruction depending on whether or not its outputs are propagated to the end of the program or are overwritten by another instruction. Although this information was propagated back to the trace of the execution, the ordered list of executed instructions, its outputs were hard to read and slowed down any attempt to fix the original program with this information.

III. Proposed approach

The original connectivity measure was written on top of the execution trace. The execution trace is a file containing every information pertaining a sample execution of a given program. It contains the address of the instruction inside the compiled code and the instruction itself. Exploiting the information provided during the compilation step we are able to propagate the output further. Thanks to the address of any given instruction we can then use the ELF (Executable and Linkable Format) file of the compiled program to understand from which source file has been originated, thanks to the information stored by the linker. We then parsed the source file to extract every function and macro declared to then be able to

TABLE I
SIMPLIFIED SAMPLE OUTPUT PROGRAM.

Line	Instruction	Connectivity	Shadowing
1	li r0, 0x0000	100%	0
2	cmp r0, 0x0001	100%	0
3	beq jump	0%	0
4	li r0 0x1111	NaN	0
5	jump:		
6	li r1, 0xaaaa	0%	0
7	INCREASE r0		
8	// MACRO: INCREASE rD in file utils.s		
9	addi &rD, 1	100%	0
10	// END MACRO		
11	addi r1, r0, 0xbbbb	100%	1
12	add r2, r1, r0	100%	0
13	li r0, 0x7777	100%	0

reference them and move freely through the source code. To better follow the execution order of the instructions we opted to "unroll" macros in the original code, which means that we wrote out the content of the macro alongside the code where it was used to better demonstrate its effects on the surrounding code.

Let's show a sample output to better explain how the results will look.

In this section we will introduce only the information that this program is able to show alongside the source code, we will leave the explanation in detail of the usefulness of this information to the next section and indicate how it can be exploited in order to speed up the process of improving a program.

Table I shows a simplified code fragment. Instead of a sequence of executed instructions, the code is formatted in the same way that the programmer used. As previously stated, each macro is unrolled within the code to mirror the behavior of the compiled code.

We can distinguish 3 columns:

1) The first contains the instruction as written by the programmer himself, so that it can be easily traced back to the original code.
2) The second contains a percentage between 0 and 100, or NaN. An instruction can contain intermediate values only if it writes more than one register or if it is traversed more than once during execution. The instruction will be marked with a NaN if it was never executed during the execution under consideration.
3) The third column contains an integer indicating how many disconnected instructions have been overwritten by the current one. It should be noted that a single instruction could be disconnected due to more than one other instruction.

Not shown in the table, we have an additional piece of information that we can add to the code. Next to each instruction we can indicate its instruction pointer (IP) and, separately, the instruction pointer of each other instruction that contributes to the loss of connectivity of the current one.

IV. EXPERIMENTAL RESULTS

We wrote all of our code in C++ and we tested it on functional tests written for the SPC58 automotive autocontroller provided by STMicroelectronics.

In this section we will discuss how to exploit each of the information that the previous step overlaid alongside the source code.

Let us start with the original metric, connectivity. This metric, although simple, has a rather important function. In particular, this shows us which instructions do not contribute to the final state of our execution. In fact, taking the table from earlier, we can see that whatever value the R1 register possessed after line 6, the final result would not be varied since it is overwritten at line 11. This allows us to develop the code in two directions, either we remove instruction 1 reducing the code size and execution time, or, we could modify the code so that that instruction contributes to the final state of the program.

The third column in the above table, on the other hand, shows us a value representing the number of instructions we could make connected by modifying the current instruction. This, especially in very long code, allows the programmer to understand how best to modify their code while minimizing the number of changes they need to make to their work.

The last feature we propose is to associate each institution with its own instruction pointer and report the IPs of which other institutions are going to override the result of the current one. This gives us an immediate understanding of the relationship between the current instruction and those to follow. By showing us where in the code the overwriting occurred, we can then easily follow the execution flow that led us to this loss of information and tweak it in a timely manner so as to correct the behavior of a specific instruction.

V. CONCLUSIONS

The tool that we propose improves the feedback to the programmer, allowing for faster corrections without the need to cross-reference the execution trace and the original code to understand where the instructions are overwriting the output of one another. Although the current method is only applicable to assembly code, we are working in order to extend the current results to any program by expoiting tools like Ghidra to trace single assembly instructions to the original code.

REFERENCES

[1] "Iso 26262-[1-10], road vehicles – functional safety," 2011.

[2] F. Angione, P. Bernardi, A. Calabrese, L. Cardone, A. Niccoletti, D. Piumatti, S. Quer, D. Appello, V. Tancorre, and R. Ugioli, "An innovative strategy to quickly grade functional test programs," in *2022 IEEE International Test Conference (ITC)*, 2022, pp. 355–364.

[3] S. M. Thatte *et al.*, "Test generation for microprocessors," *IEEE Transactions on Computers*, vol. C-29, no. 6, pp. 429–441, 1980.

[4] D. Brahme *et al.*, "Functional testing of microprocessors," *IEEE Transactions on Computers*, vol. C-33, no. 6, pp. 475–485, 1984.

[5] G. Squillero, "MicroGP—An Evolutionary Assembly Program Generator," *Genetic Programming and Evolvable Machines)*, 2005.

COVER LETTER for PhD FORUM

First name	Lorenzo
Last name	Cardone
Affiliation	Politecnico di Torino
E-mail	lorenzo.cardone@polito.it
Phone number	3917154269
Proposed PhD thesis title	GPU-based Parallel Techniques for the Evaluation of Reliability Measures of Large System-on-Chip devices
Start PhD date	November 2022
Expected End PhD date	February 2024
Type of PhD topic[1]	Applied research
Collaborators[2]	Politecnico di Torino
PhD motivation[3]	My interest in computer science started during my high school years due to my passion for video games and logic problems. During my studies I became passionate about the field of algorithmic optimization and, later, about the parallelization of my code over both multi-core and many-core systems. This passion of mine led me to the development of my master's thesis and later prompted me to explore the world of research in order to challenge my knowledge and expand my horizons.
Scientific results[4]	F. Angione et al., "An innovative Strategy to Quickly Grade Functional Test Programs," 2022 IEEE International Test Conference (ITC), Anaheim, CA, USA, 2022, pp. 355-364, doi: 10.1109/ITC50671.2022.00044 Lorenzo Cardone, Stefano Quer, “The Multi-Maximum and quasi-Maximum Common Subgraph Problem”, submitted to 2023 MDPI Computation, Special Issue Graph Theory and Its Applications in Computing

[1] e.g., Applied research/ R&D, fundamental research, etc.
[2] e.g., university, industry, research centre, etc.
[3] explain why you have decided to pursue a PhD and why in this topic.
[4] provide a list of scientific outputs and innovations (accepted, submitted, expected) including: Patents, Books/ book chapters, Journal papers, Conference papers, Workshop papers etc.