Native binary mutation analysis for embedded software and virtual prototypes in SystemC

Christoph Kuznik, Wolfgang Mueller
Faculty of Electrical Engineering, Computer Science and Mathematics
University of Paderborn/C-LAB
D-33102 Paderborn, Germany
christoph.kuznik@c-lab.de
wolfgang@acm.org

Abstract—Mutation analysis is a powerful tool for white-box testing of the verification environment in order to produce dependable and higher quality software products. However, due to high computational costs and the focus on high-level software languages such as Java mutation analysis is not yet widely used in commercial design flows targeting embedded (software) systems. Here the industry is modeling both hardware and related software parts at higher levels of abstraction, called virtual prototypes, to accelerate parallel development and shorten time-to-market. In this paper we propose a mutation testing verification flow for SystemC based virtual prototypes that may not rely on source code only but on annotated basic blocks and enables mutant creation at assembler level to heavily reduce execution costs and equivalence mutants likelihood.

I. FAULT INJECTION AND MUTATION ANALYSIS

Mutation analysis is a useful technique to test for weaknesses in the verification environment. However, most approaches focus on software flows only, especially Java. On the other hand, in the embedded software domain there is a tendency for native development kits in languages such as C++ rather then Java due to performance requirements. Moreover, these embedded systems themselves are incorporated in more and more safety related domains, such as automotive, and need to be highly dependable and fault tolerant.

For modeling and early development of these systems languages such as SystemC are ideal. SystemC is a versatile C++ based design and verification language, offering various mechanisms and constructs required for embedded systems modeling. With help of transaction-level modeling SystemC is also highly effective for virtual prototype definition and later reuse of the same verification environment with the actual IPs.

So to fully leverage mutation analysis in this area we have to optimize its existing shortcomings. For example, mutation analysis generates a huge amount of mutants which need to be compiled, resulting in enormous execution costs. Moreover, equivalent mutant detection is a tedious task. Besides that, traditional mutation analysis approaches perform mutant generation on source-code level or intermediate object code, such as bytecode in Java. However, as SystemC is C++ based and we want to enable the usage of mutation analysis in a variety of compiler flows as well as for off-the-shelf IP we propose a technique for (instruction-set specific) mutation analysis on assembler level. This approach will allow much greater generation and execution speed of the mutants themselves as well as more accuracy as we support various compiler optimizations and test the actual native binary code that runs later on in the final product.

The rest of the paper will be structured as follows. In section II we will summarize the proposed flow for native binary mutation analysis. In section III we will shortly discuss related work before we conclude in section IV.

II. INSTRUCTION-SET MUTATION ANALYSIS

To target SystemC for mutation analysis we propose an approach based on annotating basic blocks in the source code by means of a SystemC grammar parser build with ANTLR [1]. ANTLR is a language tool that provides a framework for constructing interpreters, compilers, and translators from grammatical descriptions. The steps are depicted in figure 1.

Based on the defined grammar, the parser inserts marks in the source code that do not alter the functionality but can be found later on in the native compilation, even after compiler optimization (as long as marks are not multiplied and the basic blocks are reachable code). Based on the complexity of the parser we can mark constants, variables and so on besides control flow conditions and loops. However, we first focus to identify and annotate the available control flow statements for SystemC
in our grammar definition file. Therefore, in comparison to mutation analysis for high-level software languages the amount of mutation operators is limited. Anyhow, altering the control flow conditions still allow to mimic typical program errors and do lead to improvement of testcases and the overall verification environment. Once the DUV is compiled it can be analyzed based on the marks in the native code and fault injection can be undertaken as can be seen in figure 2. First, a CFG is constructed for the control flow part and possible mutants are generated. Based on this a number of patches is generated that may alter the DUV compilation to a mutant. This compiler-integrated technique for run-time optimization does not require the mutant to be compiled but to just patched in the original compilation based on the CFG analysis, reducing computational effort. Next the test cases are applied to each mutant and the output is checked in the verification environment (strong mutation analysis). When a mutant is not killed, so both the testcase and verification environment were not able to detect the mutant, feedback is provided which basic block is affected and which mutation operator was used. Despite the CFG approach would also allow weak mutation analysis, we do not consider it here. Once a standard for the SystemC grammar parser is defined the source code does not require the mutant to be compiled but to just patched in the original compilation based on the CFG analysis, reducing computational effort. Moreover, the mutation analysis is performed on native binary level, allowing to evaluate the actual native assembler code that is run later on in the embedded system while reducing computational effort. As next step we aim to refine the SystemC grammar parser to allow a broader coverage of traditional mutation operators as well as conducting an in-depth case study assessing performance and usability of our approach in comparison to existing frameworks. Apart from that, we plan to analyse the impact of desired mutation operators, e.g. for ARMv7. Especially, a combination with dynamic binary translation, e.g. by means of QEMU, is a promising approach to further enhance mutant generation and execution speed [5]. Besides, we plan to combine this approach with previous work conducted in [6], [7] on functional coverage collection and evaluation for the SystemC ecosystem. Finally, we plan to establish a representation of the mutation analysis verification scheme in evolving multi-languages verification methodologies such as the Unified Verification Methodology (UVM) [8].

ACKNOWLEDGMENT

This work was partly funded by the DFG Collaborative Research Centre 614 and by the German Ministry of Education and Research (BMBF) through the BMBF project SANITAS (01M3088I). We greatly appreciate the cooperation with the project partners [9].

REFERENCES