BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2024
http://excel.fit.vutbr.cz

Applying formal methods to analysis of semantic differences between versions of software

František Nečas*

Abstract
The goal of this work is to propose an integration of formal methods into DiffKemp, a static analysis tool for analyzing semantic differences of large-scale C projects. The aim of this extension is facilitating analysis of more complex code changes to arithmetic and logic expressions. To achieve this, whenever a possible semantic change is found, the equivalence of the ensuing code blocks is encoded into an SMT problem instance and the difference is either confirmed or refuted using an SMT solver. The proposed solution has been implemented in DiffKemp and our experiments show that it extends the capabilities of the tool.

*xnecas27@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

In the vast world of software development, there are some projects where maintaining semantic stability between versions is crucial, because even a tiny change could negatively impact the users of the said projects. An example of such projects are various system libraries, e.g., implementations of the standard C library, where the semantics of the exported functions should remain unchanged between versions so that other programs can safely use these functions. In order to simplify and automate identifying potential changes in semantics, various static analyzers of semantic differences have been developed.

There are several tools focusing on finding semantic differences, their approaches vary significantly. On one hand, there are tools based on formal methods which can verify semantic equivalence with high precision but do not scale well (e.g., LLRêve [1]). On the other side of the spectrum, there are light-weight static analyzers that are able to analyze large chunks of code very fast but produce numerous false warnings or errors (on the extreme end, the diff Unix utility could be considered an example of this approach). One of the tools, DiffKemp [2], tries to find a middle ground between these two extremes.

DiffKemp builds on the idea that refactoring is typically done on a small part of the code, while the rest stays intact. Therefore, the analysis algorithm in DiffKemp compares the program mostly per-instruction in the LLVM Intermediate Representation (IR). When such comparison fails, DiffKemp tries to apply one of its patterns that are known to preserve semantics. While there are numerous patterns already implemented in DiffKemp, there are still some cases where the tool reports inequality despite the versions being semantically equal. A common example of this are changes to arithmetic and logic expressions, e.g., using distributive laws and other algebraic properties. There are a lot of such possible refactorings, therefore it is not feasible to implement them all manually as patterns in the tool itself.

To overcome this, our work proposes an integration of formal methods into the DiffKemp's analysis algorithm. Whenever a possibly differing instructions are found and no pattern is applicable, the equivalence of the ensuing code blocks is encoded into an SMT problem instance and the difference is either confirmed or refuted using an SMT solver.

2. Integrating Formal Methods into DiffKemp's Analysis Algorithm

The overall approach can be summarized as follows. Whenever a possible semantic change is found and no pattern is applicable, we try to perform the following steps:

1. Find the nearest pair of instructions, after which the code can be synchronized again, e.g., the instructions are the same. The code blocks

between the differing instructions and this pair of instructions needs to be checked for semantic equality. We focus only on sequential blocks that do not manipulate the memory and have no side effects.

2. If such a pair of code blocks was detected, encode the problem of checking their semantic equality into the SMT problem. Let $InVar_1$ be the set of input variables of the first block, $varmap$ the bijection mapping semantically equivalent variables, $OutVar_1$ the set of output variables of the first block, $outmap$ the bijection mapping output variables that are supposed to be equal and $Block_1$ and $Block_2$ the encoding of operations in (i.e., the formula representing the semantics of) the first and second block, respectively. We construct the formula:

$$\bigwedge_{v_1 \in InVar_1} v_1 = varmap(v_1) \wedge$$
$$Block_1 \wedge Block_2 \wedge$$
$$\neg \bigwedge_{out_1 \in OutVar_1} out_1 = outmap(out_1)$$

and use an SMT solver to check its satisfiability. The blocks are semantically equal, iff the formula is unsatisfiable. Intuitively, if we give both blocks the same input, they need to produce the same output in order for the blocks to be equivalent. If the formula has a model, the model corresponds to the inputs under which the outputs differ. The formula on the poster gives an example of encoding of semantic equality of the highlighted blocks in the section with experiments.

3. If the SMT solver verified equality of the blocks in the provided amount of time, continue the analysis using DiffKemp's main algorithm. Otherwise, report semantic inequality of the programs.

3. Results and Experiments

We have implemented the proposed solution inside DiffKemp using the Z3 [3] SMT solver. This particular solver was determined to be the best fit due to its extensive support of various theories, a mature API, active community and performance being on-par with other state-of-the-art solvers.

To evaluate the impact of our extension on DiffKemp, we have performed a series of experiments. Firstly, we checked our solution on a small set of simple hand-crafted examples in order to verify the basic functionality. More importantly, we experimented with the EqBench [4] benchmark – a collection of 147 equivalent and 125 non-equivalent program pairs. Some of the cases come from existing benchmarks that were used to evaluate other tools for checking semantic equivalence, such as Rêve and Clever, i.e., they may contain very complicated refactorings, since the tools are based on formal methods.

The table on the poster compares the results of DiffKemp with and without our extension. We can see that thanks to our extension, DiffKemp was able to correctly analyze 4 more programs as semantically equal in this particular configuration. One example of such a program can be seen at the top of the poster. While DiffKemp contains a pattern for inverse branch condition, it was not applied in this case, since the condition is not inverted syntactically (the condition would need to be $x \leq 100$). However, since the variable is an integer, $x < 101$ and $x \leq 101$ are semantically equivalent – such nuances can nicely be checked using an SMT solver.

We also experimented with our solution on projects of larger scale. For example, we tried to check semantic equality of functions that are part of the RHEL Kernel Application Binary Interface[1]. While in this experiment, the use of an SMT solver did not improve the results, it showed that DiffKemp's performance is not degraded by our extension. Finally, we have tried checking semantic equality of Linux commits that contain the words `optimize` or `refactor`. We managed to identify one commit, where our extension facilitated correct analysis of the functions as semantically equal. The said functions can be seen at the bottom of the poster – unnecessary bitwise shifts have been removed to reduce the number of instructions needed to perform the operation.

Acknowledgements

I would like to thank my supervisor Viktor Malík for his help. This work was supported by Red Hat Research.

References

[1] Moritz Kiefer, Vladimir Klebanov, and Mattias Ulbrich. Relational program reasoning using compiler ir. In Sandrine Blazy and Marsha Chechik, editors, *Verified Software. Theories, Tools, and Experiments*, pages 149–165, Cham, 2016. Springer International Publishing.

[2] Viktor Malík and Tomáš Vojnar. Automatically checking semantic equivalence between versions of

[1] A list of functions whose semantics should remain unchanged between minor releases of RHEL.

large-scale c projects. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 329–339, 2021.

[3] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[4] Sahar Badihi, Yi Li, and Julia Rubin. Eqbench: A dataset of equivalent and non-equivalent program pairs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 610–614, 2021.