Visualising CFG Differences Through Bisimulation

Céline Deknop[1,2], Johan Fabry[2], Kim Mens[1], Vadim Zaytsev[3]

[1]ICTEAM institute, UCLouvain, Belgium

[2]Raincode Labs, Brussels, Belgium

[3]Formal Methods & Tools, UTwente, The Netherlands

{celine.deknop, kim.mens}@uclouvain.be, johan@raincode.com, vadim@grammarware.net

I. Introduction

Refactoring is a common step in the process of modernising software. This task is often delegated to experts, e.g. when dealing with complex legacy software. An example of such experts is the company Raincode Labs, a company that provides services in the realm of legacy modernisation.

When working on code critical to a business, it is important to build up clients' trust in refactorings being truly behaviour preserving. One aid in building trust is formal proofs. Another is providing clear visualisations of the refactoring process. In this context, we are working on a comparison of Control-Flow Graphs (CFGs) generated from programs before and after the refactoring. In this extended abstract, we briefly cover our specific use case as well as the techniques used to generate our CFGs, and finally talk about how we envisage comparing and visualising them for differences through bisimulation.

II. Use case

One of the services provided by Raincode is *PACBASE migration* [8]. PACBASE is a fourth generation language [11] that generates COBOL code that is neither readable nor maintainable by humans and is now out of support [3]. To address this issue, PACBASE migration refactors this code to plain, human-readable COBOL code using a set of bespoke *refactoring rules* that are applied automatically to the code. An explanation of what PACBASE is as well as more details on the migration process can be found in our previous work [1].

Building trust in the beginning of such a migration project has been addressed in our earlier paper [2]. We now focus on the end of such projects: when the code is delivered and the client requires some kind of guarantee that the migrated code is indeed equivalent to the original PACBASE-generated one. To achieve this, we extract CFGs from the programs before and after the migration and compare them. Our goal is to show that, while the structure of the code has changed due to the refactoring, it is still semantically equivalent. For example, transforming two nested `IF`-statements into a single one containing both conditions linked by an `AND` will not change the behaviour of the program.

III. Comparing the graphs

Details on fuzzy parsing [10], the technique used to generate our CFGs, as well as our algorithm and its results can be found in our most recent unpublished paper, preprint made available.

This work is the CodeDiffNG AppliedPhD project funded by Innoviris.

Armed with our graphs, we now endeavour to find the best way to compare them and to provide a compact visualisation of the differences, if any. Both the techniques of bisimulation [9] and trace equivalence [6] could be good possible fits for our use case. We therefore started to explore existing tools, looking for one that could directly be applied.

We have not found such an off-the-shelf tool yet. First, most tools that perform bisimulation produce a binary answer: either the graphs are equivalent to each other, or they are not. This is the case for both the well-known CADP [4] as well as the more recent Keq [7] tools. To achieve our goal of building trust with the client, we need to have a more detailed and more nuanced answer. Hobbit [5] seemed promising since it gives a trace rather than a mere yes/no answer, but it will not be compatible with CFGs since it takes code as input while we need a higher level of abstraction.

While adapting one of those tools to provide more detailed answers or changing our input to better fit them is possible, we believe that it might be harder than expected since we already experienced a high complexity merely for installing them.

We therefore decided to implement our own bisimulation-inspired tool, accepting CFGs as input and providing a detailed explanation of which (parts of the) graphs are equivalent and which are not, returned in a format that can serve easily as input to a custom-made visualisation tool. Moreover, implementing our own tool will allow us to create our own "flavour" of bisimulation to best fit our use case.

IV. Visualising Graph Differences

Using our comparison tool, we then want to create a visualisation allowing clients to first see general statistics about the migration: a list of all the programs, colour-coded in green where our tool is sufficiently sure that the semantics is respected, in red the migrated programs which it suspects not to be behaviourally equivalent to their original version, and in orange those where it was not able to decide. Each of those programs would be clickable to open a more detailed visualisation in a second window. In this view both CFGs (pre- and post-migration) would appear side by side and the user could hover over parts of the graphs to see if there is a corresponding structure in the other graph. Such a visualisation would both increase the client's trust as well as allow them to have an easier time pinpointing where additional testing or verification needs to be performed in order to assure that the migration went smoothly.

REFERENCES

[1] C. Deknop, J. Fabry, K. Mens, and V. Zaytsev, "Improving Software Modernisation Process by Differencing Migration Logs," in *Proceedings of the 21st International Conference on Product-Focused Software Process Improvement (PROFES)*. Springer, 2020, pp. 270–286, DOI: 10.1007/978-3-030-64148-1_17.

[2] C. Deknop, K. Mens, A. Bergel, J. Fabry, and V. Zaytsev, "A Scalable Log Differencing Visualisation Applied to COBOL Refactoring," in *Proceedings of the Ninth Working Conference on Software Visualization (VISSOFT)*. IEEE, 2021, pp. 1–11.

[3] Hewlett-Packard Development Company, "Survival Guide to PACBASE™ end-of-life," https://www8.hp.com/uk/en/pdf/Survival_guide_tcm_183_1316432.pdf, Oct. 2012.

[4] INRIA, "CADP Homepage," https://cadp.inria.fr/, 2022.

[5] V. Koutavas, Y.-Y. Lin, and N. Tzevelekos, "Hobbit: A tool for contextual equivalence checking using bisimulation up-to techniques."

[6] P. B. Levy, "Infinite trace equivalence," *Electron. Notes Theor. Comput. Sci.*, vol. 155, p. 467–496, may 2006. [Online]. Available: https://doi.org/10.1016/j.entcs.2005.11.069

[7] D. Park, T. Kasampalis, V. S. Adve, and G. Rosu, "Cut-bisimulation and program equivalence," 2020.

[8] Raincode Labs, "PACBASE Migration: Flexible Process," https://www.raincodelabs.com/pacbase/, 2021.

[9] D. SANGIORGI, "On the bisimulation proof method," *Mathematical Structures in Computer Science*, vol. 8, no. 5, p. 447–479, 1998.

[10] V. Zaytsev, "Formal Foundations for Semi-parsing," in *Proceedings of the IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE 2014 ERA)*, S. Demeyer, D. Binkley, and F. Ricca, Eds. IEEE, Feb. 2014, pp. 313–317.

[11] V. Zaytsev and J. Fabry, "Fourth Generation Languages are Technical Debt," International Conference on Technical Debt, Tools Track (TD-TD), 2019. [Online]. Available: http://grammarware.net/text/2019/4gl-techdebt.pdf