Log-based behavioural differencing applied to an industrial case

Céline Deknop

ICTEAM institute, UCLouvain, Belgium
celine.deknop@uclouvain.be

Abstract. When working with a complex process, it is difficult to get a clear idea on how exactly changes to the input can impact the output. Visualising how the steps of such process evolves with input can help understanding and/or boost confidence in the produced result.
We took a suitable existing industrial process, and created a visualisation for it using an adapted log-based behavioural differencing algorithm that highlights the parts of the process that really changed. We show that our visualisation scales well, propose that our solution could be applied to other use cases, and provide a replication package to allow others to try for themselves.

Keywords: Differencing, Visualisation, Refactoring, Legacy, Migration, Industrial

1 Introduction

This paper, presented as a poster at Informatics Europe's first *Early Career Researchers Workshop* during its annual ECSS2021 conference, reports on the first 2 years of my PhD research. This research was conducted in the context of an Innoviris AppliedPhD project, in collaboration between UCLouvain university and the Raincode Labs company in Belgium. The goal of such projects is to stimulate research while keeping in touch with the reality of the industry. In this particular project, we explore the discipline of code differencing in an industrial context.

We start by introducing the chosen discipline of differencing and present its limitations (Section 2). Next, we introduce Raincode Labs and the use case we will focus on Section 3. Section 4 then presents the technique that we used along with the limitations that we encountered when applying it to the industrial data obtained from Raincode Labs. In Section 5, we present some improvements over the first implementation and reactions from our validation participants. Finally, Section 6 gives a brief conclusion, an overview of possible future work, and provides a link to a replication package that can be used by anyone wishing to give our tool a try.

2 A brief history of differencing

The algorithm used to perform differencing that we are most accustomed to (e.g., to discover differences between two commits on GitHub) has been created in 1976 by Hunt and McIlroy [7] and has stayed mostly unchanged since. While this algorithm remains quite limited, more advanced techniques might benefit a software company like Raincode.

We therefore explored what techniques are out there, and how they could be applied to the reality of Raincode's day-to-day business. Some techniques, like Kim and Notkin's *LSdiff* [8], while interesting, seemed too object-oriented for our specific use case. We explored a bit the modelling community (UMLDiff [11], ADDiff [9]), as well as the domain of code clones and mining (ROSE [13], CloneDiff [12]), before deciding on the technique we will detail in this paper: log-based behavioural differencing, as proposed by Goldstein et al. [6].

3 Raincode Labs and their migration projects

Raincode Labs is an independent compiler company that provides services for migration and modernisation of legacy systems. Among thoses services, we will focus on one: PACBASE migration [10].

PACBASE is an aging fourth generation language that allows engineers to use concise macros to generate COBOL code, instead of writing COBOL code directly. PACBASE support having ended in 2015, reliance on it has turned into a liability. Ideally the language would be retired, and companies could maintain the generated COBOL code. Yet, the COBOL code generated by PACBASE is not the most readable for humans, and rewriting all COBOL code from scratch is not feasible.

Raincode's PACBASE migration refactors PACBASE-generated COBOL code into human-readable COBOL using a set of refactoring rules that are applied iteratively on the codebase, producing new COBOL files that are maintainable by the company. Since the code being migrated is often a critical part of the core business of the client company, it is important for Raincode Labs to ensure that the client will feel "at home" in the new code. For that, the exact set of refactoring rules can be tailored to the client's needs.

Picking the exact set of refactoring rules is the first step of a migration project. First, all of them are presented to the client, so they have an idea of what the rules achieve. A first set of rules is then picked following advice from Raincode engineers. Next, a small subset of programs are migrated by Raincode engineers using the chosen set of rules. This allows the client to look at the result of a migration to get a clearer idea of what it does. After that and again with the help and advice of Raincode engineers, the set of rules can be refined. The partial migration is run again to see if the output is better, and some iterations of this process can happen until a client is fully satisfied.

This part of the process is precisely where our application of advanced differencing could be applied: there are around 140 refactoring rules available, and

while Raincode engineers are experts in the domain and have great intuition, it is sometimes difficult for them to give a precise justification of why a specific rule should be turned on or off. It is also difficult for them to explain exactly how certain changes to the chosen rule set would impact the output, because all rules can interact with one another. To help in this matter, Raincode engineers would like to have more objective arguments as to why a rule should be picked or not, and our tool could help to do just that.

4 Log-based behavioural differencing

The initial idea we based our tool upon is explained in detail in Goldstein et al.'s paper [6]. It can be summarised as follows:

1. Take logs from two executions of a system you want to analyse, and sanitise them (remove any information you don't want in the final graphs).
2. Create a Finite State Automaton (FSA) from those logs. The original authors advise to use the KTails algorithm [2], which is what we did. Many tools exist for this, we chose Synoptic [1].
3. Compare the resulting FSAs using the algorithm proposed by Goldstein et al.
4. Visualise the resulting differences.

This approach is easily applicable to Raincode's migration projects since they already generate logs describing the exact order in which rules got triggered, along with some more details (timestamp, exact lines of code on which the rule was applied). In our case, we decided to use only the names of the rules being triggered, and exclude any errors or warnings.

As for the visualisation, we decided to keep close to the idea of Goldstein et al., keeping a graph structure with START and END nodes, and transitions between nodes representing the probabilities to change from one state to another in the automata. However, we decided to take it a little further, with the addition of colours (red/orange/green for a removal/modification/addition).

We therefore set out by reproducing the original algorithm and applying it to our industrial data (for more details about the original algorithm and our results, see our VISSOFT paper [5]). The results of this first experiment were a bit disappointing: with our log files having a mean of around 280 lines, even after their transformation to an automaton, our resulting graphs contained on average 120 nodes, which was too large to analyse at a glance, as can be seen on Figure 1.

5 Shrinking the graphs and getting feedback

After closely observing our first outputted graphs, we noticed a trend that could be exploited to make them more readable: due to the very iterative nature of Raincode's migration process, our graphs had very long lines of nodes without

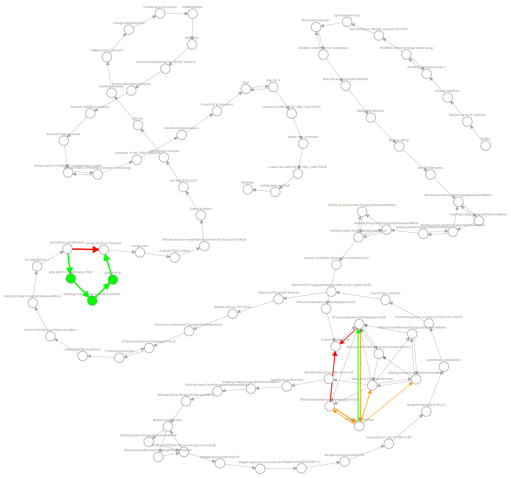

Fig. 1. Result of applying the algorithm from [6] as-is.

changes, that felt uninteresting and took up unnecessary space. We also noticed that most changes happened in only a few places of the graphs, in clusters.

Those observations lead us to create a merging algorithm to hide away uninteresting nodes, enabling us to highlight the more interesting information. Two nodes can be merged if:

- None of them are added or removed;
- No link towards them has seen a probability change;
- No link from them has seen a probability change.

When applying this merge algorithm to our graphs, we managed to reduce them from a mean of 123 nodes, to one of 33 nodes, a 75% reduction. We also decided to not just drop the nodes that were merged: they merely get hidden and can be inspected in a separate window, when the merged node is clicked. The resulting graphs felt readable enough to present them to Raincode engineers and get their feedback. For example, the merged version of the graph of Figure 1 is shown in Figure 2.

These graphs were shown to Raincode engineers during semi-structured interviews in which we focused on getting their opinion both on the usability of our tool in the context of their work, and in a more general way outside of their work. We also included some open question to allow them to express their likes and dislikes (details about our method, the exact questions and their results can be found at [4]).

The first result that jumps out is that both of them agreed that the merged version of the graphs was a strict improvement over the non-merged one. After seeing both versions, neither of them wished to use the unmerged variant, even if they would occasionally want to click on a merged node to see what is inside.

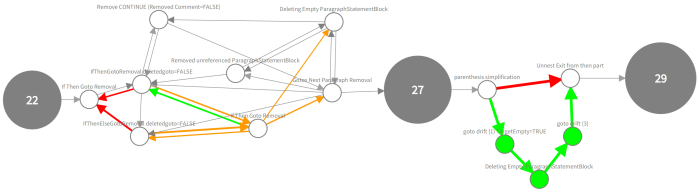

Fig. 2. Our merged graph (same input as 1)

The second result is that our tool seems pretty intuitive: with the help of a short two-page manual, both engineers were able to get situated and stop referring to the manual after 5 to 6 minutes. We did identify two points that required some verbal explanation on our part. First, the meaning of the orange color, its explanation in the user manual being too scientific to be user-friendly. Second, the feature of seeing inside a node when clicking on it was not intuitive enough for them to try and click, and should have been put forward more clearly.

Finally, when asked how exactly they would use our tool in their work, both interviewed participants came up with different ideas: one would use it as described earlier, to give justifications to a client, but the other (less client-oriented in those projects), would use it as a debugging aid, to make sure that changes to the migration process itself didn't have unwanted consequences.

6 Conclusion and replication

To conclude, we believe we produced a tool that can be used "as is" by our collaborating company, and that it could be adapted to other use cases fairly easily. Both engineers who assessed our tool had different ideas on how it might be helpful to their company, which suggests it is fairly versatile. Since finishing our validation, we also had the occasion to discuss about the tool and its purpose with several people and came up with various ideas, from the straightforward idea of comparing logs from boot executions of two Linux distributions to determine whether the second would be a good substitute for the first, to the more challenging idea of using our representation to analyse how a development team sticks to a development method (like scrum).

A good path to future work would be to use our tool in a completely different context to see how useful it can be there, and perform a second round of validation. For this, if anyone wishes to try, we have created a full replication package, with a VM and our tool installed, along with an obfuscated version of our data (Raincode's client data being confidential). We also provide documentation on how to obtain the input from another source of log, and how to run the tool on other data. All instructions to find and run the replication package can be found on GitHub [3], or else feel free to contact us.

Acknowledgements

Thanks go out to my advisors Kim Mens (UCLouvain) and Vadim Zaytsev (UTwente), the people at Raincode Labs: Johan Fabry, Yannick Barthol and Boris Pereira, and finally to Alexandre Bergel. We also thank the Innoviris funding agency for funding our CodeDiffNG Applied PhD research project.

References

1. I. Beschastnikh *et al.*, "*Synoptic GitHub page*," Online: https://github.com/ModelInference/synoptic, 2021.
2. A. W. Biermann and J. A. Feldman, "On the Synthesis of Finite-State Machines from Samples of Their Behavior," *IEEE Transactions on Computers*, vol. C-21, no. 6, pp. 592–597, 1972, DOI: 10.1109/TC.1972.5009015.
3. Céline Deknop, "Replication package and instructions," https://github.com/CelineDknp/VISSOFTArtifact, 2021.
4. ——, "Validation data," https://github.com/CelineDknp/PACBASEValidationData, 2021.
5. C. Deknop, K. Mens, A. Bergel, J. Fabry, and V. Zaytsev, "A scalable log differencing visualisation applied to cobol refactoring," in *2021 Working Conference on Software Visualization (VISSOFT)*, 2021, pp. 1–11.
6. M. Goldstein, D. Raz, and I. Segall, "Experience Report: Log-Based Behavioral Differencing," in *Proceedings of the 28th International Symposium on Software Reliability Engineering (ISSRE)*, 2017, pp. 282–293, dOI: 10.1109/ISSRE.2017.14.
7. J. W. Hunt and M. D. McIlroy, "An algorithm for differential file comparison," no. #41, 1976.
8. M. Kim and D. Notkin, "Discovering and Representing Systematic Code Changes," in *Proceedings of the 31st International Conference on Software Engineering (ICSE)*. IEEE, 2009, p. 309–319. [Online]. Available: https://doi.org/10.1109/ICSE.2009.5070531
9. S. Maoz, J. O. Ringert, and B. Rumpe, "ADDiff: Semantic Differencing for Activity Diagrams," in *Proceedings of the 19th Symposium on the Foundations of Software Engineering and the 13rd European Software Engineering Conference (FSE)*, T. Gyimóthy and A. Zeller, Eds. ACM, 2011, pp. 179–189.
10. Raincode Labs, "PACBASE Migration: Flexible Process," https://www.raincodelabs.com/pacbase/, 2021.
11. Z. Xing and E. Stroulia, "UMLDiff: An Algorithm for Object-Oriented Design Differencing," in *Proceedings of the 20th International Conference on Automated Software Engineering (ASE)*. ACM, 2005, pp. 54–65.
12. Y. Xue, Z. Xing, and S. Jarzabek, "Clonediff: semantic differencing of clones," in *Proceeding of the Fifth ICSE International Workshop on Software Clones (IWSC)*, J. R. Cordy, K. Inoue, S. Jarzabek, and R. Koschke, Eds. ACM, 2011, pp. 83–84. [Online]. Available: https://doi.org/10.1145/1985404.1985428
13. T. Zimmermann, P. Weibgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *Proceedings. 26th International Conference on Software Engineering*, 2004, pp. 563–572.