Advanced Differencing of Legacy Code and Migration Logs
(Work in Progress)

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

[1]Université catholique de Louvain & [2]Raincode Labs
Belgium
celine.deknop@uclouvain.be / johan@raincode.com / kim.mens@uclouvain.be / vadim@grammarware.net

Abstract

Most software engineers interact with some form of code *differencing* every day, either directly or indirectly. Yet, many existing algorithms and tools used in that context have not significantly evolved since the basic Unix `diff` utility. As a consequence, many specific characteristics and semantics of code are not taken into account in the process, making the task of analysing the output of such tools difficult for developers and causing valuable time to be lost. In this work-in-progress paper, we describe a concrete industrial use case that could benefit from more advanced ways of performing differencing. Additionally, we provide two leads to solutions to solve the problems faced in this context.

1 Introduction

Code differencing aims at comparing two pieces of data, typically textual source code. One piece of data is considered as source and one as target. Code differencing produces difference data that specifies what changes or edits are necessary to go from the source to the target. This technique can be used directly, in version control systems such as `git` [5] or the Unix command `diff` [20]. It is also used indirectly, in the context of code merging [17], compression [7], convergence [27] and clone detection [18].

In: A. Oprescu, E. Constantinou (eds.): Proceedings of the 13th Seminar on Advanced Techniques and Tools for Software Evolution, Amsterdam, The Netherlands, 01-07-2020, published at `http://ceur-ws.org`

Even today, many differencing tools still rely on the basic algorithm created by Hunt and McIlroy [8] in 1976, or variants thereof. These tools treat their input as simple lines of text or binary data. However, code is more than just a random stream of characters. It conforms to quite specific and strict syntactical structure, ready to be exploited, and it implies a logical flow of control and dependencies among its components. The same code fragment can also reoccur in multiple places within the file or across multiple files. All those subtleties are lost when using the Hunt-McIlroy algorithm. Flat textual comparison does not reflect the goals of a developer and obscures the intent of the changes by the overflow of displayed low level information, which can lead to frustrating and tedious experiences.

The research presented in this paper is part of the CodeDiffNG [28] project, an Applied PhD grant of the Brussels Capital Region [10] that funds a collaboration between UCLouvain (a university) and Raincode Labs (a company). Our research seeks to push code differencing to a higher level by investigating advanced forms thereof, providing engineers with a structural view of the changes and with an explanation of their intent. In section 2, we introduce the context of this project by providing a typical use case of Raincode Labs' activities around migration of legacy software systems, and discuss some of the difficulties experienced by migration engineers. In section 3, we move on to list a couple of approaches that we will explore in the near future, to help overcome these difficulties. Finally, in section 4, we provide an overview of other approaches that we have explored to some extent and might explore in the farther future.

2 Motivating Example

Raincode Labs is involved in many different projects, most revolving around modernisation of legacy soft-

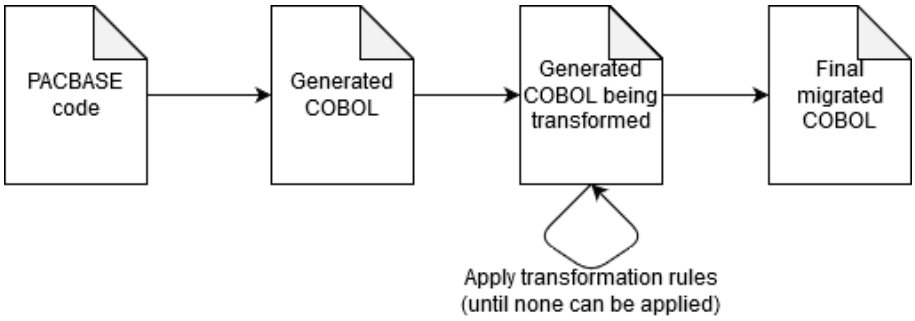

Figure 1: Summary of the migration process, showing the different artefacts and their transformations

```
1  F05DC.
2      MOVE        1       TO ICATR.
3      GO TO       F05DC-B.
4  F05DC-1.
5      ADD         1       TO ICATR.
6  F05DC-B.
7      IF          10      < ICATR THEN
8        GO TO     F05DC-FN
9      END-IF.
10     IF      CATX(ICATR) = '0' THEN
11       NEXT SENTENCE
12     ELSE
13       GO TO F05DC-C
14     END-IF.
15     MOVE 'X' TO CATM(ICATR).
16 F05DC-C.
17     IF      CATX(ICATR) NOT = '0' THEN
18       NEXT SENTENCE
19     ELSE
20       GO TO F05DC-D
21     END-IF.
22     MOVE 'Y' TO CATM(ICATR).
23 F05DC-D.
24     GO TO F05DC-A.
25 F05DC-FN.
26     EXIT.
```

```
1  PERFORM
2      VARYING ICATR   FROM 1
3                      BY 1
4      UNTIL 10 < ICATR
5      IF CATX(ICATR) = '0' THEN
6        MOVE 'X' TO CATM(ICATR)
7      ELSE
8        MOVE 'Y' TO CATM(ICATR)
9      END-IF.
10 END-PERFORM.
```

Figure 2: Example of a migration from PACBASE-generated COBOL to equivalent COBOL code that is more readable and maintainable.

ware. In that context, the company identified a few problems they are facing in their different processes relating to the topic of code differencing, and the research performed in the PhD will revolve around those use cases. The first such use case, which will be the focus of this paper, is PACBASE [9] migration [19].

2.1 PACBASE

PACBASE is a language and tool created in the 1970s. Its selling point was offering a DSL to its users that was at a higher level than available alternatives such as COBOL. The end users would then program in concise PACBASE macros, and COBOL code would be generated for them automatically. It was widely used since its creation but the end of support was announced in 2015. Since it is an ageing technology that will ultimately disappear, a more extensive discussion of PACBASE itself is out of the scope of this paper.

Since 2002 (the case of DVV Verzekeringen, an insurance company), Raincode Labs often undertakes projects of COBOL-to-COBOL migration: they take PACBASE-generated code and refactor it into its shorter, more readable equivalent which can be maintained manually after retiring PACBASE generators.

The PACBASE migration process is done through a set of about 140 transformation rules that Raincode Labs developed and refined over the years. Each individual transformation rule can be seen as a local automated refactoring: it is designed to be simple enough so that it can be proven that it does not change the semantics of the code; yet making it just a bit better and more maintainable. All rules are applied iteratively to the programs until no more refactorings can be applied. The entire process and its artifacts are detailed in Figure 1.

An example of such a COBOL-to-COBOL program transformation is given in Figure 2: we can see that all `GO TO` statements are removed, and that the control flow has been turned into a `PERFORM`, the COBOL equivalent of a `for` loop. The logic allowing to iterate 10 times that was expressed on lines 1 to 9 has been translated into a more familiar `VARYING UNTIL` clause, while the concrete action of the loop that was before on lines 10 to 15 and 17 to 22, was simplified

in a single `IF ELSE`, performing the same actions. Undeniably, this new COBOL code is more readable and maintainable than the original generated one.

Each such migration project takes on average around two weeks, needed for tweaking the configuration, enabling/disabling/applying rules, testing the result, etc. During those two weeks, the customer's programmers continue active development of the original system, making it diverge from the snapshot Raincode migration team is working on. The process to integrate these changes to the original code into the already migrated code is called a *redelivery*, and will be explained shortly. The larger the migration project, the more redelivery phases can be required to ensure customer's utter satisfaction and the end of Raincode's work.

Even though these PACBASE migration projects are largely automated, a few manual steps remain. We believe they could be improved by advanced code differencing. One of the goals of this Applied PhD project is to produce tools that are both academically relevant and concretely useful for the company, so it will be the first focus of our research. The following sections will give more details about the specifics of a PACBASE migration project, highlight the challenges and propose some ideas for solutions.

2.2 The process of migration

In this section, we dive deeper into the process that Raincode engineers go through when migrating PACBASE projects, in order to identify manual work that can be possibly facilitated or automated.

In the first phase, the transformation rules to be applied are selected from the 140 available ones: this is done in collaboration of Raincode engineers having previous experience and capable of showing transformation effects, and the customer company's engineers familiar with their coding standards. Some rules such as `GO TO` elimination are always selected, others are less popular.

Once the chosen set of rules is validated, the client needs to make sure that the original functionalities of their programs are maintained when they are migrated. The migration of PACBASE is a service that has been used for over fifteen years and has seen millions of processed lines of code successfully go in production. It is exceptional that bugs are introduced by the PACBASE migration, but nonetheless the client typically wishes to be convinced that the new code will work as intended.

To facilitate this, Raincode' engineers will run the migration a first time, so they can determine how best to make sure that all functionalities are maintained. They then partition the entire codebase in 3 parts:

- 10–30 crucial programs to be tested exhaustively;
- 80–100 programs to be tested thoroughly with both unit and integrating tests;
- the rest of the programs, to be tested for integration or even not tested at all.

All the transformation rules triggered in the migration process were applied at least once in the first partition, assuring that all rules get manually tested at least once by the client. When all lights are green, the entire generated COBOL codebase is sent to Raincode engineers, who perform the first cursory analysis: due to the scale of the codebase (typically 10–200 MLOC), the delivery may contain uncompilable code or non-code artefacts. When both parties agree on what exactly needs to be migrated, the actual process starts and after a few quality checks, the result is delivered to the customer.

As already mentioned, it is frequent that in the mean time at least a few modifications have been introduced on the customer side, in parallel with the migration process detailed in Figure 1. In that case, a redelivery phase is needed. The customer sends everything that has changed, triggering another phase of manual analysis for Raincode engineers. This time, not only do they have to make sure that everything is compilable code, they also need to find out if it is something that they have previously migrated or something completely new. If it is new, they have to make sure whether it should indeed be migrated. If it is an update, they need to know if it has changed, since some changes on the PACBASE level might result in functionally the same generated code.

After that, the migration process is performed again. Before sending the results to the customer, the engineers need to not only do some quality-checks, but also to make an analysis of what changed since the delivery. This is done mostly manually and is a very subjective process: the engineer that we interviewed described it as *"we send the new migrated files to the client if they look good enough"*.

In more concrete terms, they check if new rules got triggered in the migration process, then look at the output of `diff` between the previously sent version and the new one. If it is small enough to be manageable, they analyse it; if it is not, they briefly read the code of the new output and decide whether it "looks good".

The lack of a specific and concrete argument to motivate this need to partially redo a testing phase makes it difficult to convince the customer completely. The criteria of triggering new migration rules and measuring the diff size invalidating the previous testing may be too weak. The process of testing being lengthy and

costly, most customers are reluctant to go through it again. Instead they tend to request detailed explanations of why they need to, or a more precise indication of what part of a program needs to be looked at. This often causes more discussion, time losses, and possibly another round of redeliveries with similar issues on the next iteration.

2.3 Challenges of migration engineers

We identified two key places where nontrivial manual work tends to occur in this process: the initial codebase analysis and the redelivery. The codebase analysis is almost completely manual, but fairly quick and painless. There have also been successful attempts to automate it with ML-powered language identification [11]. Thus, we have chosen to not focus on this part at the moment.

The rest of the paper will focus on addressing codebase redelivery: Raincode engineers could really use a tool that would allow them to say precisely and confidently, what (parts of a) program(s) need to be tested again after a redelivery. Such a tool would allow the engineers to present the changes to the customer in an easily understandable way, instead of expecting them to trust their instincts. It would help negotiations if such a tool could provide insights on the reasons of changes. Indeed, even very small changes done on the PACBASE level can have big consequences on the COBOL output and therefore the new migrated version, which is something most customers do not realise.

3 Envisaged Approaches

The migration process takes the initial generated COBOL files and produces new migrated versions of this code, as well as some logs of the process. There is one log file per migrated program, and it contains the order and nature of the rules that were triggered during the migration. This log hence describes the exact process to go from the initial to the final version of each program, splitting it in multiple sub-versions.

An example of such a log is shown on Figure 3. Each line is either a triggered rule along with the number of times (patches) it got triggered, or an intermediary version. Other lines containing warnings or debug information have been removed and will not be studied in our future work—we can assume that Raincode engineers will only diff successfully migrated files.

We could apply differencing to any of the above artefacts, since we have both versions of the generated COBOL files, the migrated COBOL outputs and migration logs for each program. The idea of using the initial generated COBOL files was quickly discarded: they are known to be hard to understand, can change drastically when regenerated from a slightly adjusted PACBASE source, and can already be `diff`'ed.

```
1:tmp/filename.COB.rea
Rename Level 49 (0 patches done)
Done (0 patches done)
1:tmp/filename.COB.reb
..
1:tmp/filename.COB.rec
Next-Sentence removal (28 patches done)
Remove Useless Dots (51 patches done)
...
Done (0 patches done)
1:Result/filename.COB
```

Figure 3: A simplified migration log

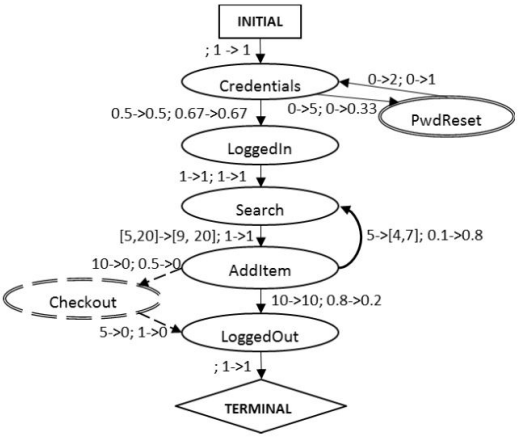

Figure 4: The result of log differencing for a simple shopping cart example [14]

Both of the remaining artefacts (the log and the migrated program) seem to be an interesting lead, and each one could address a different part of the challenge. One would give an explanation as to *why* and *how* things changed, the other giving a clearer answer as to *where* things changed in the output code. First, we will look at the log produced by the migration process.

Goldstein et al. [6] presented a way to use log messages to create Finite State Automata representing the behaviour of a service when it is known to be in a normal and working state. They create a second model from updated logs and compare it to the first model. With that, they manage to identify outliers or behaviour that is different and therefore considered abnormal. This work was used in the context of mobile networks where an abnormal behaviour can translate to network congestion. An example of the results obtained by them [6] can be found on Figure 4. Transition probability evolution is shown with an arrow, while new nodes have a different border.

This idea is relatively easy to transfer to our context. The states of the automata would be the different

rules that were applied, the transition between them representing a step in the migration process. The normal behaviour would be the old version of the logs. We would then use the new version of the logs and identify new rules that got triggered as outliers. The new transitions would correspond to changes in how the rules were applied, providing insight on how things happened this time, and more importantly, it could give an explanation as to *why* it changed.

The second approach we consider is to use the migrated COBOL directly, and try to compare it in a way useful for our use case. Laski and Szermer [14] propose a way to make structural diffs using reduced flow graphs (see Figure 5) in the context of code revalidation, which suit our purpose quite well.

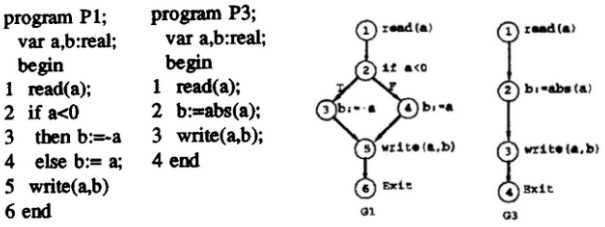

Figure 5: Two reduced control-flow graphs representing successive version of a program [14]

They create reduced flow graphs for both versions and apply classical modifications (relabelling, collapsing and removal) to their nodes in order to find an isomorphic graph (see Figure 6). In the resulting graph, all the nodes that have their initial labels represent the code that has not changed, all the differences being abstracted by *MOD* nodes that were transformed.

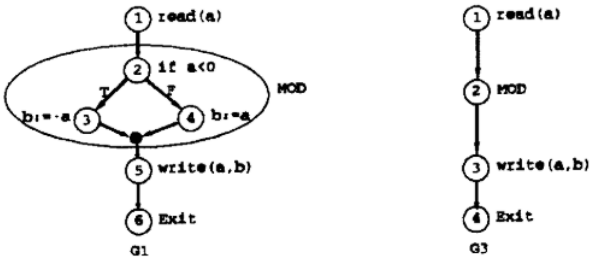

Figure 6: The process to get the isomorphic graph for the program **P1** from Figure 5 [14]

This representation is way more concise than the hundreds of lines given by a *diff* output, and has better chances of being more legible for both the customer's developers and migration engineers. The places where modifications occurred, are clearly identified and can easily be found in the code to be tested again, giving a clear answer as to *where* things changed in the outputted code.

It is important to note that the algorithm aims at giving the most coarse result possible without losing precision. Whenever possible, it would give an output such as the one in Figure 6 with parts left unchanged that do not need to be tested. However, if the difference in the code is too big, it would output a graph that just consists in a *MOD* node, which is still useful: it gives the migration engineers footing when they tell their customer that the entire code needs to be retested.

4 Related Work

We have presented two approaches that are directly related to what we are doing at the time of writing this paper, but many other options were explored, that may or may not prove useful for our future endeavours. We detail those here.

Papers presenting ideas or tools that perform differencing in a specialised or **advanced** way, are a rare find, but they still exist. The one closest to our current interest would be Kim and Notkin's *LSdiff* [12] (Logical Structural Differencing), an approach aiming at representing structural changes in a very concise manner, focusing on allowing the developer to understand the semantics of the changes. However, this approach seems to be more suited for object-oriented code, which does not correspond to our COBOL use case. We found multiple other papers focusing on the object-oriented paradigm, among them the tools calcDiff [2] and Diff-CatchUp [24].

The **modelling** community could teach us a few things in this regard as well, so we learned about tools that are made to perform clear and efficient differencing on a specific kind of models. Many of those exist for the widely-used conventions like UML (e.g., UMLDiff [23]), activity diagrams (e.g., ADDiff [16]) or feature models (e.g., in FAMILIAR [1]). Witnessing the abundance of many different tools for each kind of model, an approach to allow for a more generic way to difference models was also proposed by Zhenchang Xing [22].

We made sure to take note of the different techniques used when performing **data** differencing. From the starting point of the Hunt-McIlroy algorithm treating said data as simple text, to the extension to binary [21] when the need of differencing more heterogeneous artefacts. Afterwards, many different and modern techniques were developed, including those based on control flow graphs, as described in our second approach to the PACBASE use case and other tools making use of ASTs or at least parse trees as with GumTree [4] or cdiff [26]. We are also exploring the idea of enriching the initial data format with infrastructures as *srcML*, and how it can be applied to dif-

ferencing [15] as well as about its corresponding tool *srcDiff* [3].

Finally, we looked at what ideas could be leveraged from other software engineering disciplines, like software **mining** or code **clone** detection. In the work of Kim and al. [13], logical rules are mined from the code to help represent the structural changes. Tools using those practices were also developed, for example ROSE [29], that mines the code to be able to suggest which changes should happen together, or CloneDiff [25], that uses differencing in the context of clone detection.

5 Conclusion

In this paper, we first presented what code differencing is, along with how it works and in which contexts it is used. We then explained some of its limitations, and gave a concrete example of how those restrictions can impact industrial processes like our PACBASE migration use case. Finally, we proposed two leads that we will explore, and described other existing approaches that could prove useful in our future research.

It is important to note the main limitation of the way code differencing is used today: its disregard for the *nature* of what it analyses. Some research has been done to try and overcome this, but the way differencing is still taught in classes or used in the industry more or less corresponds to the initial Hunt-McIlroy algorithm in most cases. Moving forward, we should keep that in mind when designing new solutions and tools.

The next step for us will be to start experimenting, using the data from past migration projects provided by Raincode Labs. As soon as a first prototype is created, we will be able to get immediate feedback from the potential users and focus our efforts on getting a tool that will be of real use to the company.

References

[1] M. Acher, P. Heymans, P. Collet, C. Quinton, P. Lahire, and P. Merle. Feature Model Differences. In *CAiSE*, volume 7328 of *LNCS*, pages 629–645. Springer, 2012.

[2] T. Apiwattanapong, A. Orso, and M. J. Harrold. A Differencing Algorithm for Object-Oriented Programs. In *ASE*, page 2–13. IEEE, 2004.

[3] M. Decker, M. Collard, L. Volkert, and J. Maletic. srcDiff: A Syntactic Differencing Approach to Improve the Understandability of Deltas. *JS:E&P*, 32, 10 2019.

[4] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-Grained and Accurate Source Code Differencing. In *ASE*, page 313–324. ACM, 2014.

[5] Git. *Git main page*. Online: `https://git-scm.com/`, 2020.

[6] M. Goldstein, D. Raz, and I. Segall. Experience Report: Log-Based Behavioral Differencing. In *ISSRE*, pages 282–293, 2017.

[7] J. J. Hunt, K.-P. Vo, and W. F. Tichy. An Empirical Study of Delta Algorithms. In *SCM*, page 49–66. Springer, 1996.

[8] J. W. Hunt and M. D. McIlroy. An Algorithm for Differential File Comparison. CSTR #41, Bell Telephone Laboratories, 1976.

[9] IBM. *PACBASE documentation page*. Online: `https://www.ibm.com/support/pages/documentation-visualage-pacbase`, 2020.

[10] Innoviris. *Applied PhD page*. Online: `https://innoviris.brussels/fr/applied-phd`, 2020.

[11] J. Kennedy van Dam and V. Zaytsev. Software Language Identification with Natural Language Classifiers. In *SANER ERA*, pages 624–628. IEEE, 2016.

[12] M. Kim and D. Notkin. Discovering and Representing Systematic Code Changes. In *ICSE*, page 309–319, 2009.

[13] M. Kim, D. Notkin, and D. Grossman. Automatic Inference of Structural Changes for Matching across Program Versions. In *ICSE*, pages 333–343. IEEE, 2007.

[14] J. W. Laski and W. Szermer. Identification of Program Modifications and Its Applications in Software Maintenance. In *ICSM*, pages 282–290. IEEE, 1992.

[15] J. I. Maletic and M. L. Collard. Supporting Source Code Difference Analysis. In *ICSM*, pages 210–219. IEEE, 2004.

[16] S. Maoz, J. O. Ringert, and B. Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *FSE*, pages 179–189. ACM, 2011.

[17] T. Mens. A State-of-the-art Survey on Software Merging. *TSE*, 28(5):449–462, 2002.

[18] H. Min and Z. Li Ping. Survey on Software Clone Detection Research. In *ICMSS*, page 9–16. ACM, 2019.

[19] Raincode Labs. PACBASE Migration: More than 200 Million Lines Migrated. `https://www.raincodelabs.com/pacbase/`, 2018.

[20] Unix. *Unix diff man page*. Online: `http://man7.org/linux/man-pages/man1/diff.1.html`, 2020.

[21] Z. Wang, K. Pierce, and S. Mcfarling. BMAT — A Binary Matching Tool for Stale Profile Propagation. *JILP*, 2:1–20, 06 2000.

[22] Z. Xing. Model Comparison with GenericDiff. In *ASE*, pages 135–138. ACM, 2010.

[23] Z. Xing and E. Stroulia. UMLDiff: An Algorithm for Object-Oriented Design Differencing. In *ASE*, pages 54–65. ACM, 2005.

[24] Z. Xing and E. Stroulia. API-Evolution Support with Diff-CatchUp. *TSE*, 33(12):818–836, 2007.

[25] Y. Xue, Z. Xing, and S. Jarzabek. Clonediff: semantic differencing of clones. In *IWSC*, pages 83–84. ACM, 2011.

[26] W. Yang. Identifying Syntactic Differences between Two Programs. *SPE*, 21(7):739–755, June 1991.

[27] V. Zaytsev. Language Convergence Infrastructure. In *GTTSE*, volume 6491 of *LNCS*, pages 481–497. Springer, Jan. 2011.

[28] V. Zaytsev et al. *CodeDiffNG: Advanced Source Code Diffing*. Online: `https://grammarware.github.io/codediffng`, 2020.

[29] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining Version Histories to Guide Software Changes. In *ICSE*, page 563–572. IEEE, 2004.