Improving a Software Modernisation Process by Differencing Migration Logs

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

[1] Université catholique de Louvain, Louvain-la-Neuve, Belgium
[2] Raincode Labs, Brussels, Belgium
[3] Universiteit Twente, Enschede, The Netherlands

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

Abstract. Software written in legacy programming languages is notoriously ubiquitous and often comprises business-critical portions of codebases and portfolios. Some of these languages, like COBOL, mature, grow, and acquire modern tooling that makes maintenance activities more bearable. Others, like many *fourth generation languages* (4GLs), stagnate and become obsolete and unmaintained, which first urges and eventually forces migrating to other languages, if the software needs to be kept in production. In this paper, we dissect a software modernisation process endorsed by Raincode Labs, utilised in particular to migrate software from a 4GL called PACBASE, to pure COBOL. Having migrated upwards of 500 MLOC of production code to COBOL using this process, the company has ample experience with this process. Nevertheless, we identify some improvement points and explain the technical side of a possible solution, based on migration log differencing, that is currently being put to the test by Raincode migration engineers.

Keywords: Software modernisation, legacy programming languages, software migration, software evolution, code differencing, COBOL, PACBASE, 4GL

1 Introduction

When COBOL was first introduced and published in 1960 [6], it enabled writing software that replaced the manual labour of thousands of people previously performing pen-and-paper bookkeeping or at best manual data entry and manipulation. When 4GLs (fourth generation languages) started emerging, they allowed developers to write significantly shorter programs, and enabled automated generation of dozens of pages of COBOL code from a single statement [22,29]. Nowadays, in the era of intentionally designed software languages [18] and domain-specific languages [31], conciseness and brevity is appreciated as much as readability, testability, understandability and ultimately, maintainability [9]. Yet, legacy software continues to exist due to the sheer volume of it: just COBOL alone is estimated to have at least 220 billion lines of code worldwide, according to various sources. Business-critical legacy systems still make up a massive

fraction of the software market: in 2017, it was reported that 43% of all banking systems in USA were built on COBOL, and that 95% of ATM swipes end up running COBOL code [27]. Migration projects are possible, but extremely challenging [30] and prone to failure due to overconfidence, misunderstanding, and other factors [5].

Over the last 25 years, Raincode Labs [25], a large independent compiler company, has conducted many different legacy modernisation projects. In this paper, we focus on the modernisation process for one specific kind of such projects [26]: removing the dependency on the compiler and the infrastructure of PACBASE [14], a 4GL that will be explained in section 2. The research presented in this paper is a part of the CodeDiffNG [39] research project, an Applied PhD grant of the Brussels Capital Region that funds a collaboration between the UCLouvain university and the Raincode Labs company. The project is aimed at exploring the opportunities to push code differencing further by investigating advanced forms thereof, providing engineers with a structural view of the changes and with an explanation of their intent, especially in the context of realistically large codebases being used over long periods of time. In that context, this collaboration initiative with Raincode engineers aims to identify some of the problems they are still facing in their different software modernisation processes relating to the topic of code differencing, and design use cases based around those.

In section 2 we will explain in detail the problem context and software modernisation process currently adopted by the company to migrate PACBASE-generated COBOL code to a more maintainable and human-readable equivalent. In section 3, we will present the concept of code differencing, which we want to use to improve on the process described in the previous section. We will then detail where exactly our solution would fit, as well as describe the principles that we intend to put in place. Then, we will present the current state of our work in section 4. Finally, in section 5, we provide an overview of other approaches that we have explored to some extent and might explore in the farther future.

2 The Problem Space: PACBASE migration

PACBASE [14] is a language and a tool created in the 1970s. Its original name was PAC700, for "programmation automatique Corig" (French for "Corig automated programming", where Corig was "conception et réalisation en informatique de gestion", a structured programming methodology popular 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 program concise PACBASE macros, and COBOL code would be generated for them automatically. PACBASE was widely used since its creation throughout its life cycle [2]. The "Compagnie Générale d'Informatique", which developed it, was absorbed by IBM in 1999. In 2000, PACBASE itself was modernised and rewritten in Java [28], but this did not suffice to prolong its life. The first attempt to suspend its support was made in 2005, and its definitive retirement was an-

nounced in 2015 [11]. Hence in companies that still rely on it, there is a pressing need to migrate software written in PACBASE to plain COBOL.

Since 2002 Raincode Labs has often undertaken such projects of PACBASE-generated COBOL to plain COBOL migration, one example being the case of the insurance broker DVV Verzekeringen reported by Bernardy [4]. Raincode Labs takes PACBASE-generated COBOL code and refactors it into a shorter, more readable equivalent that can be maintained manually [26]. PACBASE is an aged technology that will ultimately disappear, and an extensive discussion of the way PACBASE itself works is out of scope of this paper. Our main focus here is on the migration process of PACBASE-generated COBOL to a more concise, maintainable, and human-readable equivalent, and on how differencing of migration logs can help to improve that process.

The PACBASE migration process is achieved through a set of 140 transformation rules developed and refined by Raincode Labs over many years. Each single transformation rule can be seen as a local automated refactoring designed to be simple enough so that it can be proven not to change the semantics of the code; yet making it just a bit more concise, readable or maintainable. All rules are applied iteratively to the code until no further refactorings can be applied. This entire process and the artefacts involved are summarised in Figure 1. Apart from summarising the migration process, which is the focus of this section, Figure 1 also illustrates how, as a side-effect of the migration process, a migration log is produced. We will see in section 4 how differencing of such migration logs could help to further improve the migration process.

A concrete example of a COBOL-to-COBOL program transformation is given in Figure 2. All `GO TO` statements are removed and the control flow is turned into a `PERFORM`, the COBOL equivalent of a `for` loop. The logic that allows to iterate 10 times, contained in lines 1 to 9, has been translated into a more familiar `VARYING UNTIL` clause. Additionally, the concrete action of the loop, on lines 10 to 15 and 17 to 22 before, was simplified into a single `IF ELSE` that performs the same actions. Undeniably, this new COBOL code is more concise, more readable and more maintainable than the original (generated) one.

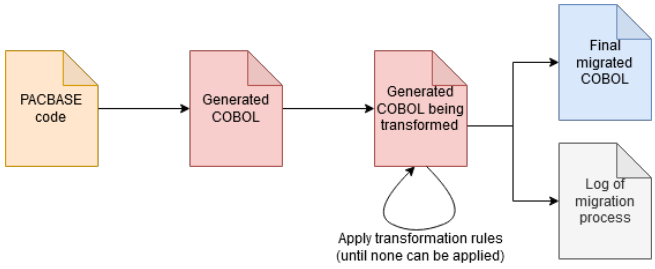
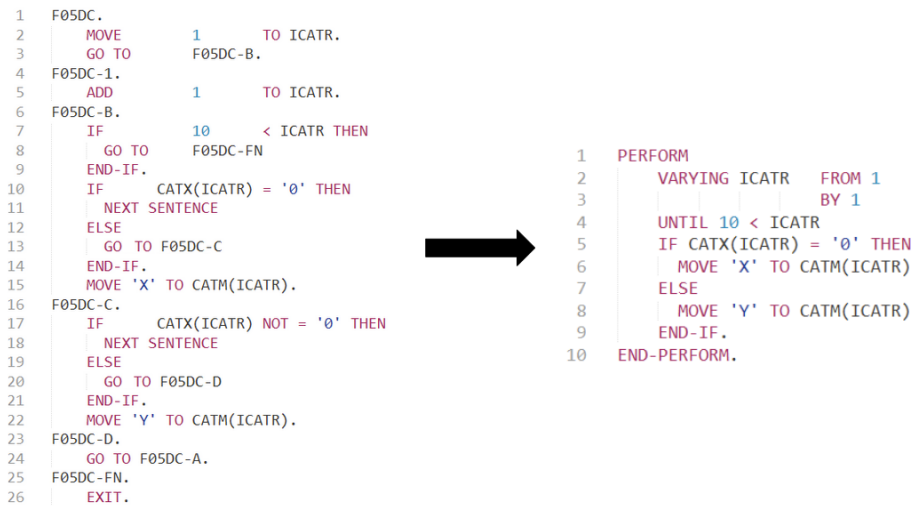


Fig. 1: Summary of the migration process, showing the different artefacts involved and their transformations.



```
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.
```

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

Transformation rules are applied iteratively, and it takes 33 intermediary steps to perform the migration from Figure 2. Let us take a closer look at some of them. The first transformation rule that is triggered is fairly simple as it simplifies the code in "one go", while others may need a few iterations, as we will see later. This first rule is called **NEXT SENTENCE Removal**, and is applied twice. As the name suggests, it removes the two `NEXT SENTENCE` instructions on lines 11 and 18, replacing them by the instruction `CONTINUE`. In COBOL, `NEXT SENTENCE` jumps to the instruction after the next full stop (here, it jumps respectively to lines 15 and 22), whereas the `CONTINUE` instruction simply does nothing and is used as a placeholder where code is required but nothing needs to be done (here, in the body of an if statement). In this particular example, we can easily see that this transformation preserves behaviour, since the full stop is right after the end of the if statement, where execution naturally continues.

Some transformation rules remove artefacts that are no longer useful. An example of such rule would be **Remove Useless Dots**, that is applied four times to the code a few steps later. Indeed, since we removed our `NEXT SENTENCE` instructions, we do not need the full stops signalling such sentences anymore. Therefore, the full stops on lines 21 and 14 get removed. At the same time, the ones on lines 2 and 9 are deleted as well, since they never really served a purpose. Another example of such a transformation rule would be to **Remove Labels**, i.e., delete labels when they are no longer needed (in our example all labels ultimately get removed).

Some bigger transformations, such as the creation of the `PERFORM` loop visible in the resulting code in Figure 2, may require applying quite a few intermediate

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

(a) Flipped conditions without `CONTINUE`

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

(b) Start of the if/else `GO TO` loop structure

```
1  * Raincode removed Label  F05DC.
2       MOVE 1 TO ICATR
3       .
4
5
6   F05DC-B.
7       IF 10 < ICATR
8         GO TO F05DC-FN
9       ELSE
10  * Raincode removed Label  F05DC-C.
11      IF CATX(ICATR) = '0'
12      MOVE 'X' TO CATM(ICATR)
13      ELSE
14      MOVE 'Y' TO CATM(ICATR)
15      END-IF
16      END-IF
17  * Raincode removed Label  F05DC-D.
18  * Raincode removed Label   F05DC-A.
19      ADD 1 TO ICATR.
20      GO TO F05DC-B.
21
22   F05DC-FN.
23      EXIT.
24
25
26
```

(c) Simplified if/else clause doing the **MOVE**s

```
1  * Raincode removed Label  F05DC.
2       MOVE 1 TO ICATR
3       .
4
5
6    F05DC-B.
7       PERFORM UNTIL FALSE
8        IF 10 < ICATR
9         GO TO F05DC-FN
10      ELSE
11  * Raincode removed Label  F05DC-C.
12      IF CATX(ICATR) = '0'
13      MOVE 'X' TO CATM(ICATR)
14      ELSE
15      MOVE 'Y' TO CATM(ICATR)
16      END-IF
17      END-IF
18  * Raincode removed Label  F05DC-D.
19  * Raincode removed Label   F05DC-A.
20      ADD 1 TO ICATR
21      END-PERFORM.
22
23   F05DC-FN.
24      EXIT.
25
26
```

(d) Creation of the `PERFORM`

Fig. 3: Snapshots of the loop-creation process

transformation rules. To remain relatively concise, we will highlight only a few of them in Figure 3. First, a transformation rule **Remove Idle Instructions** is triggered, allowing to flip the condition of the if statements and the corresponding line of code, so that we then can get rid of the else condition now containing the `CONTINUE` (Figure 3a). Then, a transformation rule **Recognise Loop-like Patterns** is triggered twice in a row, and after some clean-up steps, we can start to see an if-else structure containing `GO TO`s that starts to resemble a loop on lines 9, 10 and 21 (Figure 3b). Quite a few more steps are needed however to bring all the conditions into the simple if-else clause that we get in the end (Figure 3c); before the **Replace GO TO by Loop Exit** transformation rule can finally create the `PERFORM` loop that we can see in Figure 3d. The final steps create the `VARYING` and performs some cleanups, resulting in the code on the right-hand side of Figure 2.

Need for redelivery. The process of migrating an entire codebase takes on average around two weeks, which includes tweaking the configuration, enabling/disabling/applying the transformation rules, testing the produced result, etc. (Al-

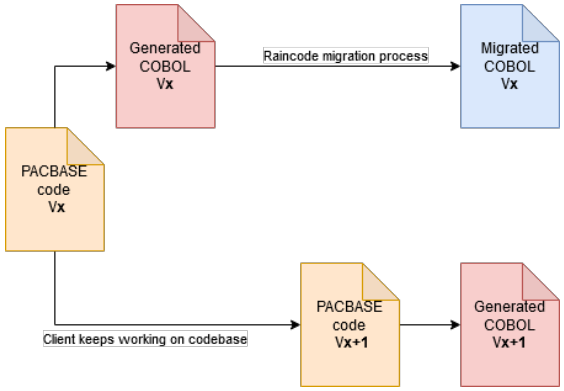


Fig. 4: Migration process, affected by the customer still working on the PACBASE code.

though all transformations were designed to be behaviour preserving, this testing phase can help convince the customer that the program indeed behaves the same way before and after the transformation.) During those two weeks, the customer's programmers typically continue with active development on the original system, making it diverge from the snapshot Raincode's migration team is working on, as depicted in Figure 4. 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 the customer's complete satisfaction at the end of Raincode's migration process.

The PACBASE migration process is largely automated, yet some manual steps remain. In what follows, we will explain how we believe they could be improved through advanced code differencing. One of our goals is to produce tools that are both academically relevant and concretely useful for the company. In the paragraphs below, we dive deeper into the process that Raincode engineers go through when migrating PACBASE projects, in order to identify manual work that possibly can be facilitated.

Raincode Labs' migration service is as tailored to the customer as possible. Raincode engineers experienced in PACBASE migration collaborate closely with the customer's engineers familiar with customer-specific coding standards. In the first phase, the customer selects from the 140 available transformation rules, the ones they want to apply to their codebase. The available transformation rules include universally appreciated `GO TO` elimination and rearranging control flow for code readability. Other transformations are more cosmetic and concern data alignment, code formatting or layout — they can be switched off when incompatible with the customer's coding standards. Raincode Labs' migration engineers coach the customer in choosing which rules to include, showing the transformation effect and providing suggestions about what would work best.

Once the chosen set of rules is validated, the customer wants to be sure that the original behaviour of their programs is maintained after migration. Since the PACBASE migration service has been used for over fifteen years and has seen millions of processed lines of code successfully go in production, it is quite exceptional that bugs are introduced by the PACBASE migration. Nonetheless, the customer typically wishes to be convinced that the migrated code (that is often business-critical) will work as intended.

To facilitate this, Raincode engineers perform a test run of the migration, and in collaboration with the customer partition their codebase in three parts:

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

It is verified that all transformation rules that were triggered in the migration process are applied at least once in the first partition, assuring that all rules get manually tested at least once by the customer. When all lights are green, the customer's entire PACBASE-generated COBOL codebase is sent to Raincode engineers, who perform a cursory analysis of the codebase. Due to the scale of the codebase (typically 10–200 MLOC), the delivery may contain uncompilable code or non-code artefacts. Only 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 previously mentioned, it is frequent that in the meantime modifications have been introduced to the PACBASE code on the customer's side, in parallel with the migration process. In that case, a *redelivery* is needed. The customer sends all PACBASE-generated COBOL code that has changed, triggering another phase of manual analysis for Raincode engineers. This time, not only do they have to make sure that all the code is compilable, but also need to determine for each program if it has been migrated previously, or is something completely new. If it is new, they have to reevaluate whether it should indeed be migrated. If it is an update, they need to know if it has actually changed, since come minor readability tweaks on the PACBASE level might not propagate at all to the COBOL code or yield functionally equivalent code.

After this manual verification, the automated migration process is performed again. Before sending the results to the customer, Raincode's engineers now need to do not only some quality-checks, but also 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 customer when they look good enough"*. More concretely, they check if new rules got triggered in the migration process, then look at the output of a `diff` between the migrations of the previously sent version and the new one. If the difference is small enough to be manageable, they analyse it; if it is not and they feel like they can't confidently assure that the behaviour is identical, they ask the customer to perform a new test phase on those files.

Challenges of migration engineers were identified in two key places where nontrivial manual work tends to occur in this migration process: analysis of the initial codebase and redelivery. The codebase analysis is almost completely manual, but fairly quick and painless. There have also been successful attempts to automate it with language identification powered by machine learning [15]. Thus, we have chosen not to focus on this part at the moment.

The remainder of the paper will focus on addressing codebase redelivery instead. Raincode engineers could benefit from a tool that would allow them to say precisely and confidently, what (parts of a) program(s) need(s) to be tested again by the customer after a redelivery. Such a tool would allow the engineers to present the changes in the migrated codebase (that were triggered by changes to the PACBASE-generated COBOL) to the customer in an easy-to-understand way, instead of expecting them to trust their instincts. It would help negotiations if such a tool could provide insights on the reasons of why and how the migrated code was changed. Indeed, sometimes even very small changes to the original PACBASE code can have consequences on the COBOL output so significant that the new migrated version will also drastically change. This effect is not anticipated by most customers.

3 The Solution Space: Code Differencing

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` or the Unix command `diff`. It is also used indirectly, in the context of code merging [23], compression [12], convergence [38] and clone detection [24].

Even today, many differencing tools still rely on the basic algorithm created by Hunt and McIlroy [13] 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. A same code fragment can also reoccur in multiple places within a file or across multiple files. Such subtleties are lost when using the Hunt-McIlroy algorithm. Flat textual comparison does not reflect developer goals and obscures the intent of the changes due to the excess of low level information displayed, which can lead to frustrating and tedious experiences.

Using this algorithm thus often results in outputs that are hard to use or understand by developers, because they are too detailed or miss important relationships between the components involved in the change. They neither communicate nor reflect the intent, and ignore the semantics of the changes — the very thing one tends to seek when looking at a diff. When interacting directly with the output of a diff tool, it is often hard to get a good understanding of what functionalities — if any — changed since the last version, simply because there

is just too much information to process at once. For example, if refactorings were applied, the behaviour of the program was expected to remain unchanged. Yet, the actions taken may result in changes that span over multiple files, and a developer would need to put a lot of effort in analysing these changes to understand or verify whether they indeed still respect the original program semantics.

Even interacting with diffs indirectly, like using a version control system with a good visualisation frontend (`gitk`, `gitx`, Git Kraken, Git Extensions, etc), can still be frustrating to users. This is because it occasionally forces them to be confronted with the differences at a fine-grained level and makes them perform the merge manually. Examples of this are when just a few edits and moves caused a file to be flagged as completely different, or when there were simultaneous changes to the same set of lines.

3.1 Improvement Opportunities

As was illustrated in Figure 1, 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 transformation rules that were triggered during the migration. This log file describes the exact process to go from the initial to the final version of each program, splitting it in multiple subversions. A snippet of such a log is shown in Figure 5. Each line represents either a triggered rule along with the number of times (patches) it got triggered, or an intermediary version. Those intermediary versions are stored to disk, to enable analysis of the exact effect of the rule that got applied. Two different types of intermediary versions exist: the main phases denoted with a letter (rea, reb, ...) and the subversions marked with numbers (here, 0030). Other lines containing warnings or debug information have been removed from the snippet as they will not be studied — we just assume that Raincode engineers will only diff successfully migrated files.

We could apply differencing to any of the above artefacts: we have both versions of the PACBASE-generated COBOL programs, the migrated COBOL

```
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)
1:tmp/filename.COB.0030
Remove Useless Dots (51 patches done)
...
Done (0 patches done)
1:Result/filename.COB
```

Fig. 5: A simplified migration log

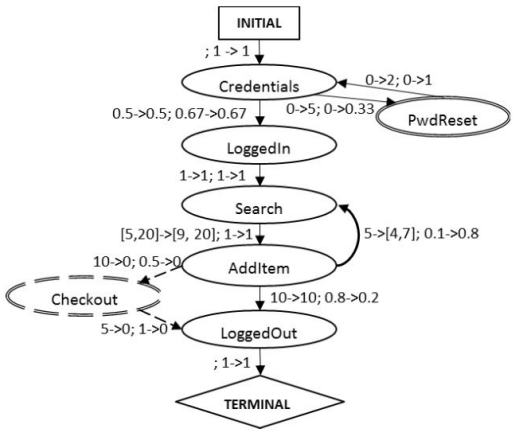


Fig. 6: The result of log differencing for a simple shopping cart example [19].

programs and migration logs for each migrated 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.

Both remaining artefacts (the log files and the migrated programs) are capable of providing valuable information, each addressing some of the challenges of migration engineers. 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. Thus, we think it would be beneficial to use a combination of differencing both these artefacts to construct a full picture. First, we will look at the log produced by the migration process.

3.2 Log differencing

In prior work, Goldstein et al. [10] 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 [10] is shown in Figure 6. Nodes considered different (added or removed) have a specific border, in our example these are the nodes *PwdReset* (added) and *Checkout* (removed). Edges are adorned with two values separated by a semicolon. The first value is the time in seconds that the transition took in the underlying log, which we will not consider in our work since we are not interested in the performance of the migration process. The second value is the transition probability evolution. The probability

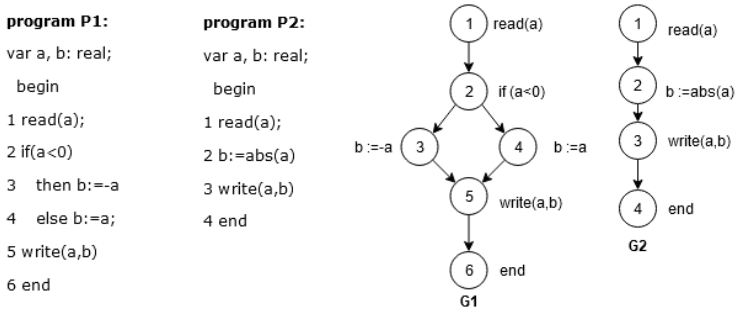


Fig. 7: Two reduced CFGs representing successive program versions [19]

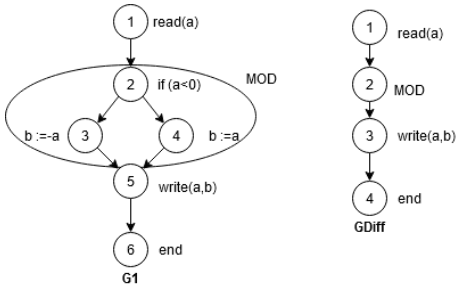


Fig. 8: Getting an isomorphic graph for the program **P1** from Figure 7 [19]

to go from the first INITIAL node to the *Credentials* node remains unchanged while the probability to go to the new node *PwdReset* evolves from 0 to 0.33.

As we will see shortly in section 4, translating this example to our specific context is fairly simple. Nodes will correspond to log lines (representing either an intermediary version or a rule), and the edges and their probabilities will model the iterative process of the migration.

3.3 Code differencing

The second approach we consider is to use the migrated COBOL programs directly, and compare them in a way useful for our use case. Laski and Szermer [19] propose a way to make structural diffs using reduced flow graphs (see Figure 7) in the context of code revalidation, which may suit our purpose quite well.

They create reduced flow graphs for both versions (before and after the transformation) and apply classical modifications (relabelling, collapsing and removal) to their nodes in order to find an isomorphic graph (see Figure 8). In the resulting graph, all nodes having their initial labels represent code that did not change, whereas all the differences abstracted by *MOD* are nodes that were transformed.

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 the migration engineers. Places where modifications occurred are clearly identified and can easily be found in the code under consideration for retest, giving a clear answer as to *where* things changed in the migrated 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 8 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 of 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 Differencing log files

Of the two approaches we presented, we needed to start with one. We chose to focus on log differencing for now because it was more recent, but we fully intend to explore diffs of the code later on.

In our efforts to improve Raincode Lab's software modernisation process, we started adapting the log differencing method presented by Goldstein et al. [10] to their use case. First, we analysed the data provided by Raincode for the pilot study of a specific migration project. The entire migration process concerned about 3000 artefacts in total, and two redeliveries were performed. The first concerned 47 files, the second only 8.

The log files generated by the migration process were quite substantial with an average of around 1200 lines before cleaning up all unneeded information and around 900 useful log lines. Even if we hope that this amount can be reduced further by translating to graphs, we made some design choices early on to ensure having something small enough that we could analyse.

First, we decided to divide the logs into the main phases that can be observed in Figure 5: `rea`, `reb`, `rec`, `red` and `ref`. These correspond to naturally independent phases (preprocessing, clean up, etc) which have always been analysed separately by Raincode engineers. For the main migration phase, which still contains upwards of 400 lines, we are abstracting from all intermediate subversions by removing their identification number. Those numbers are too specific to a particular instance of migration, and would prevent the resulting graphs to be anything but linear and a simple offset in the identifying numbers would result in the two graphs being flagged as entirely different, even if the rules were applied in the same order.

With those decisions, we implemented the first prototype of a log file differencing tool. As the main algorithm has already been implemented, but no visualisation support is available yet, the images visible in Figure 9 were produced manually from our data. The upcoming versions of our prototype will generate these images automatically. Our preliminary conclusions drawn from analysing the three sets of files from both pilot study redeliveries are as follows:

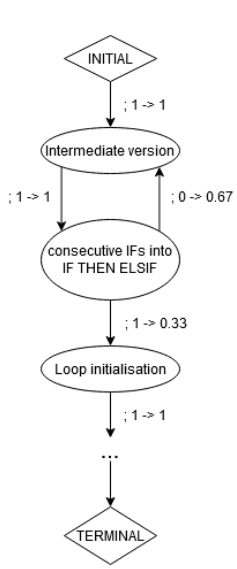
First, we observed that less work-intensive phases (like `rea`, `reb`, `red` and `ref`) produced no differences between the two sets of files. Although we cannot guarantee that this finding would remain valid when analysing more than just a small sample of files, one might find it somewhat intuitive that phases of preprocessing or cleanup are likely not to change when the changes made to the

files are not substantial. We will see whether this hypothesis can be verified for most of our other files.

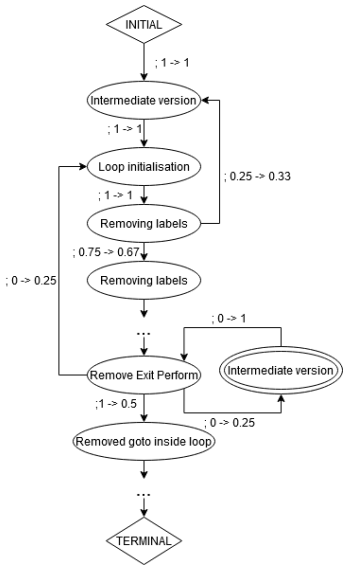
The second observation on both examples is that the resulting graphs tend to be quite linear. Most often, we find long successions of nodes having a transition probability of 1 to the next one. Then, we find some clusters where something changed between the two versions, creating a less linear path. This can probably be explained by the fact that the migration process is iterative, and could also provide an interesting way to improve our future visualisation. By collapsing such linear parts of the graphs, we could emphasise the parts that are different, enabling a more easy analysis of the differences "at a glance".

The result in Figure 9a is representative of what we see most: a long, linear graph with changes that are quite localised (for this specific example, the entire rest of the graph is identical between the two logs). The changes are either variations in transition probabilities, indicating that the rules were applied more or less times, or the creation of new path, such as between the nodes **consecutive IFs into IF THEN ELSIF** and **Intermediate version**, meaning that the changes resulted in a new order in which rules got triggered.

The result presented in Figure 9b is the only occurrence we found so far of a node being added (no removal of node has been found yet). The added node is only an intermediary step and not a new rule, so it does not raise a major red flag. It should nonetheless be analysed in more detail by an engineer since it creates an entirely new path moving through a big part of the graph.



(a) a.



(b) b

Fig. 9: A shortened mock-up of the log differencing of one of Raincode's redelivered file.

5 Related Work

We have presented two approaches directly related to the pilot experiments we are conducting at the time of writing this paper, in the hope of improving Raincode Labs' software modernisation process. Many other options were explored as well, that may or may not prove useful for our future endeavours. We detail some of those here.

Papers presenting ideas or tools that perform differencing in specialised or *advanced* ways, though a rare find, still exist. The one closest to our current interest is be Kim and Notkin's *LSdiff* [16] (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. There are other papers focusing on the object-oriented paradigm, among them the tools cal-cDiff [3] and Diff-CatchUp [35].

The **modelling** community could teach us a few things in this regard as well, so we studied tools that are made to perform clear and efficient differencing on a specific kind of model. Many of those exist for the widely-used models like UML (e.g., UMLDiff [34]), activity diagrams (e.g., ADDiff [21]) 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 [33].

We also took note of 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 [32] 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 [8] or cdiff [37]. We are also exploring the idea of enriching the initial data format with infrastructures as *srcML*, and how it can be applied to differencing [20] as well as about its corresponding tool *srcDiff* [7].

Finally, we are also looking at what ideas could be leveraged from other software engineering disciplines, like software **mining** or code **clone** detection. For instance, in the work of Kim and al. [17], logical rules are mined from the code to help represent the structural changes. Tools using those practices were also developed, for example ROSE [40], that mines the code to be able to suggest which changes should happen together, or CloneDiff [36], that uses differencing in the context of clone detection.

6 Conclusion

In this paper, we have presented a real industrial case study of a process of software modernisation and language/technology retirement by way of iterative

code transformation. We identified some of the weakest links of this process, stemming from limitations of contemporary code differencing techniques, and showed how those restrictions can impact industrial processes like our PACBASE migration use case. Finally, we described some of the state of the art in code differencing and presented early results of how our work could build on them to improve the current migration practices within Raincode Labs.

To reiterate, the main limitation of the way code differencing is used today is its disregard for the *nature* of what is being analysed. 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 those limitations in mind, and try to surpass them, when designing new solutions and tools.

Acknowledgements

We thank the Raincode migration engineers Boris Pereira and Yannick Barthol for their collaboration, as well as the participants of the seminar SATToSE 2020, where an early version of this work was presented in June, for their feedback.

References

1. M. Acher, P. Heymans, P. Collet, C. Quinton, P. Lahire, and P. Merle. Feature Model Differences. In *CAiSE*, LNCS 7328, pages 629–645. Springer, 2012.
2. A. Alper. Users say Pacbase worth effort. *Computerworld*, Aug. 1987.
3. T. Apiwattanapong, A. Orso, and M. J. Harrold. A Differencing Algorithm for Object-Oriented Programs. In *ASE*, page 2–13. IEEE, 2004.
4. J.-P. Bernardy. Reviving Pacbase COBOL-generated code. In *Proceedings of the 26th Annual International Computer Software and Applications*. IEEE, 2002.
5. D. Blasband. *The Rise and Fall of Software Recipes*. Reality Bites, 2016.
6. CODASYL. Initial Specifications for a Common Business Oriented Language (COBOL) for Programming Electronic Digital Computers. Technical report, Department of Defence, Apr. 1960.
7. M. Decker, M. Collard, L. Volkert, and J. Maletic. srcDiff: A Syntactic Differencing Approach to Improve the Understandability of Deltas. *Journal of Software: Evolution and Process*, 32, 10 2019.
8. J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-Grained and Accurate Source Code Differencing. In *ASE*. ACM, 2014.
9. M. Feathers. *Working Effectively with Legacy Code*. Prentice-Hall, 2004.
10. M. Goldstein, D. Raz, and I. Segall. Experience Report: Log-Based Behavioral Differencing. In *ISSRE*, pages 282–293, 2017. DOI: 10.1109/ISSRE.2017.14.
11. Hewlett-Packard. Survival guide to PACBASE™ end-of-life. https://www8.hp.com/uk/en/pdf/Survival_guide_tcm_183_1316432.pdf, Oct. 2012.
12. J. J. Hunt, K.-P. Vo, and W. F. Tichy. An Empirical Study of Delta Algorithms. In *SCM*, page 49–66. Springer, 1996.
13. J. W. Hunt and M. D. McIlroy. An Algorithm for Differential File Comparison. CSTR #41, Bell Telephone Laboratories, 1976.

14. IBM. *PACBASE documentation page*. Online: https://www.ibm.com/support/pages/documentation-visualage-pacbase, 2020.
15. J. Kennedy van Dam and V. Zaytsev. Software Language Identification with Natural Language Classifiers. In *SANER ERA*, pages 624–628. IEEE, 2016.
16. M. Kim and D. Notkin. Discovering and Representing Systematic Code Changes. In *ICSE*, page 309–319. IEEE, 2009.
17. M. Kim, D. Notkin, and D. Grossman. Automatic Inference of Structural Changes for Matching across Program Versions. In *ICSE*, pages 333–343. IEEE, 2007.
18. R. Lämmel. *Software Languages: Syntax, Semantics, and Metaprogramming*. Springer, 2018.
19. J. W. Laski and W. Szermer. Identification of Program Modifications and Its Applications in Software Maintenance. In *ICSM*, pages 282–290. IEEE, 1992.
20. J. I. Maletic and M. L. Collard. Supporting Source Code Difference Analysis. In *ICSM*, pages 210–219. IEEE, 2004.
21. S. Maoz, J. O. Ringert, and B. Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *FSE*, pages 179–189. ACM, 2011.
22. J. Martin. *Applications Development Without Programmers*. Prentice-Hall, 1981.
23. T. Mens. A State-of-the-art Survey on Software Merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002.
24. H. Min and Z. Li Ping. Survey on Software Clone Detection Research. In *ICMSS*, page 9–16. ACM, 2019.
25. Raincode Labs. https://www.raincodelabs.com.
26. Raincode Labs. PACBASE Migration: More than 200 Million Lines Migrated. https://www.raincodelabs.com/pacbase, 2018.
27. Reuters Graphics. COBOL blues. http://fingfx.thomsonreuters.com/gfx/rngs/USA-BANKS-COBOL/010040KH18J/, Apr. 2017.
28. C. Rémy. Un nouveau PacBase, entièrement Java. 01net, https://www.01net.com/actualites/un-nouveau-pacbase-entierement-java-114108.html, July 2000.
29. L. Schlueter. *User-Designed Computing: The Next Generation*. Lexington, 1988.
30. A. A. Terekhov and C. Verhoef. The Realities of Language Conversions. *IEEE Software*, 17(6):111–124, Nov./Dec. 2000.
31. M. Völter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. L. Kats, E. Visser, and G. Wachsmuth. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.
32. Z. Wang, K. Pierce, and S. Mcfarling. BMAT – A Binary Matching Tool for Stale Profile Propagation. *Journal of Instruction-Level Parallelism*, 2:1–20, 06 2000.
33. Z. Xing. Model Comparison with GenericDiff. In *ASE*, pages 135–138. ACM, 2010.
34. Z. Xing and E. Stroulia. UMLDiff: An Algorithm for Object-Oriented Design Differencing. In *ASE*, pages 54–65. ACM, 2005.
35. Z. Xing and E. Stroulia. API-Evolution Support with Diff-CatchUp. *IEEE Transactions on Software Engineering*, 33(12):818–836, 2007.
36. Y. Xue, Z. Xing, and S. Jarzabek. Clonediff: semantic differencing of clones. In *IWSC*, pages 83–84. ACM, 2011.
37. W. Yang. Identifying Syntactic Differences between Two Programs. *Software Practice & Experience*, 21(7):739–755, June 1991.
38. V. Zaytsev. Language Convergence Infrastructure. In *GTTSE*, volume 6491 of *LNCS*, pages 481–497. Springer, Jan. 2011.
39. V. Zaytsev et al. *CodeDiffNG: Advanced Source Code Diffing*. Online: https://grammarware.github.io/codediffng, 2020.
40. T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining Version Histories to Guide Software Changes. In *ICSE*, page 563–572. IEEE, 2004.