Understanding large codebase refactoring through differencing

Céline Deknop

Thesis submitted in partial fulfillment of the requirements for the Degree of Doctor in Engineering Sciences and Technology

November 2023

ICTEAM
Louvain School of Engineering
Université catholique de Louvain
Louvain-la-Neuve
Belgium

Thesis Committee:

Pr. Kim **Mens** (Advisor)	UCLouvain/ICTEAM, Belgium
Pr. Vadim **Zaytsev** (Co-Advisor)	UTwente, The Nederlands
Pr. Charles **Pecheur** (Chair)	UCLouvain/ICTEAM, Belgium
Pr. Siegfried **Nijssen** (Secretary)	UCLouvain/ICTEAM, Belgium
Pr. Nicolas **Anquetil**	University of Lille, France
Pr. Serge **Demeyer**	University of Antwerp, Belgium
Dr. Johan **Fabry**	Raincode Labs, Belgium

Understanding large codebase refactoring through differencing

by Céline Deknop

ICTEAM
Université catholique de Louvain
Place du Levant, 2
1348 Louvain-la-Neuve
Belgium

This work was supported by the AppliedPhd program from Innoviris, 2019-PHD-7 CodeDiffNG.

Abstract

This thesis concerns advanced code differencing techniques in the context of automated large-scale refactoring for legacy software systems. Commonly, differencing compares two artefacts (code, but not only), and is based on the largest common subsequence algorithm developed by Douglas McIlroy and James Hunt in 1976. This algorithm considers both artefacts as blobs of text and outputs the shortest sequence of modifications (adds and deletes) that is necessary to go from one version to the next. Flat textual comparison like this 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.

We explored ways to enhance the experience of differencing users by creating new differencing techniques that take advantage of modern techniques and knowledge of the nature of the artefacts they are analysing. This thesis was done in collaboration with a company called Raincode Labs. They are compiler experts and offer various services in the realm of legacy software and automated refactoring. One of those services, which is the use case of this thesis, is automated refactoring of generated COBOL code. While this process is well-known to experts within the company, they have found that, since the process is fairly intricate and behaves as a black box, clients have a hard time trusting the newly refactored code. This causes communication issues, and experts have reported that they wish for a tool that allows them to better transmit their expertise to their clients.

Combining the new differencing technique and the need present at Raincode Labs to better explain their process, many tools and techniques were created during this thesis, including a log-based behavioural differencing algorithm, a COBOL semi-parser, and a trace equivalence comparator. These tools are accompanied by visualisation modules, facilitating better communication with clients and reinforcing trust in automated refactoring processes. The contributions of this work have the potential to benefit both Raincode Labs and the wider software engineering community, providing solutions to challenges in legacy software maintenance and refactoring.

Acknowledgments

To be representative of this thesis, cette partie se doit d'être écrite dans un *melting-pot* français-anglais.

First, I would like to thank Innoviris for the funding of this PhD project, as well as all of the jury members that have followed me during these four years. I also thank Xavier Lampe, without whom we would not have been able to complete this work.

In English still, I want to thank every single one of my colleagues at the INGI department. Our lunchtime conversations and all of the *belote* games have made this journey even more memorable than it would have been without you. *Belote pour toujours, à bas les whisteux !* In particular, I am grateful for the discussions, the good vibes and the great playlists that I shared with my *co-bureau*: Benoît, Olivier, Julien and Pierre (even though we did not get to see a lot of him). Since this PhD was done in collaboration with a company, I also have my Raincode colleagues to thank: you've welcomed me into your office with open arms and have always been so prompt to support my research. A very special thank you goes to my two validation participants, Boris and Yannick and to Alexandre Bergel, who helped me design both the visualisation tools.

En français maintenant, je tiens à remercier mes amis et ma famille, qui m'ont soutenue pendant quatre ans. Vous y avez toujours cru, parfois plus fort que moi, et je ne sais pas si j'y serais arrivée sans votre patience et votre soutien. Lucie, Arnaud, Seb, Pierre, Benoît, Damien, merci d'être toujours là. Finalement, ma maman m'a dit un jour qu'un des buts qu'elle s'était fixé pour mon éducation était de me faire aimer apprendre. Maman, je pense qu'on peut dire que cet objectif est atteint.

This language sandwich would not be complete without some more English. I am absolutely convinced that I had the best team I could ever have asked for to lead me through academia and this PhD. Johan, Kim and Vadim, you have been able to know when I needed to be pushed and when I needed to be supported. I am forever grateful for everything you taught me and proud to say that I did, in fact, do it.

Jamais deux sans trois.

Contents

List of Figures

Listings

Part I

Preface

Introduction 1

1.1 Context

This doctoral thesis is the result of a partnership with the company Raincode Labs [Rai23]. They mainly work in the domain of legacy software systems and offer various services, among them automated refactoring of generated code. While they are experts in their domain, they expressed interest in collaborating with researchers, in order to strengthen some parts of their processes.

The focus of this work is therefore to explore the potential of advanced code differencing techniques in domains that Raincode Labs identified as weak spots, where our research could enhance the process via new tools and a different point of view. This work does not aim to replace the established refactoring process and tools of Raincode Labs, which have been successfully used for over 15 years.

To identify areas of improvement within their processes, extensive discussions were conducted with Raincode Labs to gain a comprehensive understanding of their operations and identify their concerns. The primary challenge highlighted during these discussions was the difficulty of effectively communicating with their clients. The precise expertise possessed by Raincode Labs often proves challenging to convey to individuals, even to those with some technical background. This is why there is a demand for tools or methodologies that can provide objective proof and reinforce their expertise to foster stronger client relationships.

Addressing these concerns and providing solutions in the form of advanced differencing techniques will be the focal point of this research. By leveraging these techniques, we aim to develop tools and methodologies that enhance understanding and communication with clients, facilitate knowledge transfer, and provide tangible evidence of the expertise offered by Raincode Labs.

1.2 Refactoring in practice

When software ages, it tends to expand beyond the features initially planned, an effect known as scope creep and defined as *"any change in project scope or pressure to deliver more than what was agreed between customers and vendors*

initially" [Kom+20]. Constant introduction of features and bug fixes makes code less healthy, harder to maintain, less habitable [JF09]. While this is unavoidable in the software life cycle, uninhabitable code is difficult to read and change, making the maintenance process less efficient and more costly.

Whenever code gets harder to maintain, it is good practice to clean things up through code refactoring, a task nowadays routinely performed throughout the software life cycle. However, the process of manually applying said refactorings is known to be tedious and error-prone [CM08]. In response to this, refactoring tools have been created. These tools can automatically apply refactorings, which makes it easier and faster for developers to improve the quality of their code while avoiding the errors that come with repeated manual labour.

In industry, when dealing with legacy software, refactoring presents a double challenge: not only does the codebase get smelly, but the knowledge about its technical specificities gets less and less common among developers [Kha+14]. This means that it is sometimes impossible to maintain with in-house developers, requiring companies to employ very expensive independent experts.

Another option available to companies is to refactor the code on a very large scale, sometimes moving entirely away from a previous programming language or getting away from previous dependencies on ageing technologies. This is sometimes referred to as re-engineering [CC90; SN02].

In practice, such undertakings can be done in-house but will take a great deal of time and effort, ranging from weeks to months as estimated in a recent survey [Ive+22], a time during which the usual maintenance tasks of creating features and fixing bugs need to be halted. Since this is not always possible, the reengineering can be outsourced to experts outside of the company. In these cases, the exact specifications of what is to be done and the expected output first need to be determined before the code gets transformed into a new version that will be rolled into production and then maintained by the original company.

We see issues with communication and understanding between both parties in such large-scale refactoring projects; the code owners as well as the refactoring experts. The first step is to determine exactly what needs to be transformed, as well as how. These decisions will be guided by the experts but we know requirements engineering needs to be done with care [Mac12] and experts often find it difficult to convey their knowledge to clients.

The second friction point happens after the refactoring, when the new code is released. Since none of the code owners were actively involved in the refactoring, the previously known structures in the code will be entirely different and unfamiliar. While the new code is arguably more habitable, it is often completely unrecognisable. Yet, it has to be rolled into production to

continue its life cycle. This new code is hard to trust, putting the responsibility on the experts to reassure the code owners.

An obvious way to alleviate scepticism about this new version of the code would be for the owner to run an extensive test suite. If all the tests that passed with the previous version still are green with the new one, it should be safe to deploy. However, test suites can be completely absent or do not have a satisfying coverage of the code when they exist. Manually creating new tests is expensive and time-consuming. While the discipline of automated software testing has seen lots of interest lately, it cannot fully replace manual testing [BWK05] nor has it been designed for legacy software that is not well-comprehended in the first place. In practice, we see clients asking the refactoring companies to provide proof that the two versions of the code are equivalent and safe for production or to at least provide guidance on which parts of the new system are most likely to fail.

1.3 Problem statement

We have argued that complex processes like large-scale refactoring are difficult to understand and trust when they are performed either by automated tools or third parties. This leads to less usage of automated tools and delays during large-scale refactoring projects handled by external experts. We have found that even when working with a company that has over fifteen years of experience, code owners struggle with trusting the unfamiliar results they get, asking for proof that it is safe to use and leading to time loss for both parties.

Therefore, it would be beneficial to help the understanding of the refactoring process itself, as well as to show that the behaviour of the code is preserved after an automatic refactoring has been applied.

In this thesis, we set out to tackle the problem of trusting and understanding an automated refactoring using advanced differencing techniques. Since we collaborated with the company Raincode Labs (chapter 4 details both the company and the use case we chose), the goal of this thesis is twofold. First, to produce advanced differencing techniques and tools that will be broadly applicable and add to the state of the art in the domain. Second, to ensure that the techniques and tools scale to the reality of industrial data and can be later used by our partner company.

We tackled the project from two angles, using different artifacts from the refactoring process. First, we analysed the logs of the refactoring process, comparing two consecutive executions and showing how the process evolves when the input or the configuration file changes. With this, we aim to help giving an explanation of the behaviour of the automated refactoring process itself. We provide our experts with a tool that allows them to show their

clients the effects of modifying the configuration or input files.

Second, we analyse the code, comparing it before and after the refactoring process. We identify the code structures most affected by the refactoring process, extract them from the code, and then translate them to models that we compare. This allows us to guarantee, for part of the code, that its behaviour (possible paths of execution) remains the same after having gone through the refactoring process.

With both those tools, Raincode experts should be able to better convey the reasoning behind their processes as well as reassure their clients that the automated refactoring produces code that is safe to put in production. To ensure that this is the case, we performed validations on both tools we produced, involving Raincode experts and taking advantage of their feedback.

Having set the overall context, we will detail the research questions that guided our work.

1.4 Research questions

The discipline of code differencing emerged in 1970, but since then, tools have arguably not evolved much. Therefore, most users performing differencing tasks do not benefit from the new techniques discovered by the community and are still treating code as a blob of text, ignoring the structural and semantic dimensions that it is known to have. We set out to use this unused information and produce tools that would help enhance the understanding of the changes presented by differencing techniques.

This leads us to the following main research question to guide our research: *How can we help code owners and refactoring engineers understand the process and results of automated refactoring on large legacy code bases?* This big question can then be broken up into smaller research questions:

RQ1 *How can we provide insights as to how a large-scale process (e.g. an automated refactoring) reacts to changes?*
One of the reasons for the lack of trust in automated processes is the fact that they behave as black boxes. Due to their many parts, even a small change to their configuration or input can have large effects that are both difficult to understand for an outsider and difficult to explain by an expert.

RQ2 *How can we extract from the code the structural information impacted by the automated refactoring, and do so in a language-agnostic way?*
Extracting parts of code to later analyse it is a very common endeavour [Cor+00; DKU17; HH01; CG14]. The first question that presents itself is: Which parts of the code are necessary for the comparison and which can be left aside?

Once the parts to extract have been identified, the next question is: What technique is best suited to obtain them? Which will most fulfil our needs of having the most flexible and language-agnostic tool possible?

RQ3 *What models do we need in order to compare them and show that they are equivalent?*
Formally proving that two programs are equivalent in their behaviour is a subject that has been explored many times [How96; Cio+14; Cra02; GS10; WD14]. Before we can perform any comparison, we need to ask what model(s) would be best suited for our needs. Since we also want to create a visualisation tool, we need a representation of the code that will be easy to interpret for developers. However, we also need a model that will be comparable. Are those goals in opposition to each other and would we benefit from having two models and translating from one to the other?

The second part of the question requires us to pick a technique to compare our models. We need the technique to prove as much equivalence in the behaviour of the code as possible as well as provide details on its process so we can visualise it properly.

RQ4 *How can we visualise the output of our tools? How do we manage to scale to the size of a real industrial use case?*
Visualisation is a great way to analyse the information produced by comparison tools. Taking advantage of well-known colour schemes (red/green/orange) makes a visualisation tool easily accessible while intuitive interactions allow guiding the flow of a user's reflection. However, the scale of industrial data poses question: Can everything be shown or is it helpful to hide some information to help users get the bigger picture?

RQ5 *How can the tools produced in this thesis be used, in which contexts within and outside of the company?*
Since this thesis was conducted in collaboration with a company, it is important to reflect on the techniques and tools produced in terms of generalisability. Firstly, can the tools be used within the company? If so, is it only in a very specific context or are the experts able to imagine several uses for the tools produced within their processes? Secondly, is this applicable to other contexts outside of the company?

1.5 Approach

To develop the tools proposed in this research, we employed a systematic and iterative approach, using the following methodology. The process begins with comprehensive discussions with experts from Raincode Labs to identify specific areas within their existing processes that could benefit from the application of advanced code differencing techniques. Subsequently, a thorough analysis of the state of the art is conducted to determine the most suitable techniques and methodologies.

Once a technique is selected, a detailed plan is presented to Raincode Labs experts in order to have a transparent and collaborative experience. During this discussion, their feedback, concerns, and prioritised features are carefully noted to ensure that tools align with their specific requirements and address the identified weaknesses effectively. This iterative feedback loop facilitates the creation of tools that not only meet the research objectives but also resonate with the practical needs of Raincode Labs.

The development process comprises two essential components: main algorithms and visualisation modules. Main algorithms are designed to incorporate selected code differencing techniques, enabling the identification of meaningful elements. Those elements are then highlighted in the visualisation modules, using colours and interaction to guide users in understanding them. Throughout the development process, the research community's expertise and input are sought via the publication of papers and engagement in academic conferences. This allows for a broader perspective and constructive criticism, enhancing the robustness and effectiveness of the tools being developed. The feedback and insights gained from the research community further refine the tools and ensure their alignment with both industry standards and academic rigour.

Following the creation of the tools, validation phases are conducted in collaboration with the Raincode engineers. This is the first time they directly interact with the tool and provide hands-on feedback. Their reactions, concerns, and suggestions are considered, forming the basis for further improvements and fine-tuning of the tools. Reflections and lessons learned from these validations are then looped back into the research community through scientific papers, allowing us to share the benefits of having an industrial partner with real-world experience.

1.6 Contributions

To enhance the understanding of large-scale refactoring processes, we provide a collection of tools, including two visualisation tools, two advanced differencing techniques, and a semi-parser. The development of each element

entails a dual focus: ensuring scalability for accommodating the industrial-scale data provided by our partner company and maintaining a level of generality that allows for broader application beyond our immediate context. Finally, we provide a validation for each of the tools we created, either via automated testing and performance comparison or with a validation performed thanks to the feedback of experts from our partner company.

C1 *Application of an existing log-based behavioural algorithm to industrial-scale data*
Goldstein et al. [GRS17] presented a technique allowing to extract relevant information from logs, transform them into a Finite-State Automaton and compare the models extracted from two successive executions of a process in order to provide insights. We adapted their log-extraction strategy to the logs produced by our industrial large-scale refactoring process, reimplemented and applied their comparison algorithm to the generated models. This contribution tackles **RQ1**.

C2 *Creation of a visualisation tool for log-behavioural differencing*
When applying the visualisation proposed by Goldstein et al. to our industrial data, we encountered limitations that rendered the visualisation unusable in its basic version. We proposed solutions to overcome said limitations, implemented them in a visualisation tool and validated that our solutions did solve the encountered limitations. This partially tackles **RQ4**.

C3 *Implementation of a COBOL semi-parser*
In order to show that the behaviour of the code is preserved after going through the automated refactoring process, we identified the code structures that are most affected by the refactoring. We then wrote a semi-parser using the technique of fuzzy parsing that allowed us to extract said structure from the code. We remained as flexible and language-agnostic as possible through the use of a configuration file specifying in detail what is necessary to extract from the code. This tackles **RQ2**.

C4 *Implementation of a trace equivalence tool*
We transformed the output of our semi-parser into Labelled Transition Systems, and compared them using a trace equivalence algorithm. We found that the basic version of the algorithm presented some limitations, and proposed solutions to overcome them. This tackles **RQ3**.

C5 *Creation of a visualisation tool for trace equivalence between two programs*
We augmented the base trace equivalence algorithm to mark states and

transitions as they are explored, allowing us to then visualise the different execution paths explored as well as show when they are considered equivalent and when they are not. We encountered scaling issues in this visualisation as well and presented solutions to overcome them. This tackles **RQ4**.

C6 *Validation of all preceding contributions*
For each of the contributions described above, we performed validations in order to ensure that they scale with regard to the time of execution or size of the visualisation. We presented in detail why we were able to apply the techniques that we chose to our use case, as well as described what are their strength and weaknesses. We finally validated the tools with Raincode Labs experts to find if and how they could be used in their specific context. This tackles **RQ6**.

1.7 Supporting publications

This dissertation is supported by the following publications, all of which have the doctoral candidate as the first author.

[Dek+20a] This paper *"Advanced Differencing of Legacy Code and Migration Logs"* presents early ideas on the application of the log-based behavioural differencing algorithm. It was presented at the SATTOSE in 2020, but not published since we wanted to write a more mature version of this work thanks to the feedback from the community.

[Dek+20b] In the paper *"Improving a Software Modernisation Process by Differencing Migration Logs"*, we presented our finalised log-based behavioral differencing tool and gave intuitions on how well the tool scales with industrial data. This paper was presented at PROFES in 2020 and adresses **RQ1** with **C1**.

[Dek+21] In this paper *"A Scalable Log Differencing Visualisation Applied to COBOL Refactoring"*, we present the visualisation tool for our log-based behavioural differencing algorithm, show the limitations that we found with the original visualisation and propose a solution to fix said limitations. We then validate with experts that our solution does indeed help with the limitations. This was presented at VISSOFT2021 and parially addresses **RQ4** and **RQ5** through **C2** and a part of **C6**.

[Dek+22a] The paper *"Generating Customised Control-Flow Graphs for Legacy Languages with Semi-Parsing"* details the code structures most affected by the automated refactoring process as well as how we extracted them from the code through semi-parsing. We compare our semi-parser to

an industrial-grade full parser in terms of both time of execution as well as ease of use, highlighting cases in which the use of this technique will be most useful. This addresses both **RQ2** and **RQ5** through **C3** and a part of **C6**.

[Dek+22b] In this paper *"Visualising CFG Differences Through Trace Equivalence"* we present how we translated the Control-Flow Graphs that we generated into models that are compatible with our chosen technique of trace equivalence in order to compare them. We also justify why the technique of trace equivalence is the best fit for our use case. This was presented at ICSME2022 tackles **RQ3** through **C4**.

To be published In the paper *"Understanding Large-scale Codebase Refactoring through Semantic-aware Differencing"*, we present our trace equivalence algorithm in more detail, as well as explain how we enhanced the base algorithm to better fit our use case of comparing refactored programs. We also describe the visualisation tool that we created using Roassal and validate both it and the trace equivalence algorithm. This finishes to answer **RQ4** and **RQ5** through **C5** and the last part of **C6** and has been submitted to SANER2024.

During this project, we paid special attention to making our work reproducible, which is why we produced the following artifacts, available on Zenodo:

- Log-based visualisation tool, accompanying our VISSOFT paper [Dek+21], found at: https://zenodo.org/record/5450770
- Validation of our fuzzy parser, accompanying our ICSME paper [Dek+22a], reciever of the ICSME2022 Best Artefact Award,
  found at: https://zenodo.org/record/6806075

While our industrial data is confidential, every tool and technique produced during this thesis is open-source and can be found in the following GitHub repositories:

- Log-based behavioural differencing and visualisation tool:
  https://github.com/CelineDknp/CodeDiffNG
- Fuzzy-parser and trace equivalence tool:
  https://github.com/CelineDknp/SemiParsingCFG
- Visualisation tool for the trace equivalence:
  https://github.com/CelineDknp/semiParsingVisualisation

1.8 Roadmap

In this section, we describe the structure of this dissertation, which is divided into three parts: preface, analysis of the refactoring process and analysis of the effects of the refactoring process.

Preface

This first part is meant to provide all the necessary information to approach the following two parts. We start with an introduction to the problem, how we approached it and what research questions motivated our work, chapter 1. We follow this with background information in chapter 2 encompassing some details about the industrial context and basic information on the legacy programming language used in this thesis, COBOL. Next, we provide a quick history of the disciplines of refactoring and differencing which are key to our work. We then move on to the state of the art in the domain with chapter 3. To conclude, chapter 4 presents Raincode Labs in more detail, then explains the different steps and artifacts included in our use case, an automated large-scale refactoring.

Analysis of the refactoring process

The second part of this thesis focuses on the analysis of the refactoring process itself, in order to help experts explain it to their clients. In chapter 5, we present our chosen technique of comparison, log-based behavioural differencing, how it applied to the use case and explain how we overcame the limitation that we found in the base visualisation algorithm. In chapter 6, we present how we validated our technique and visualisation tool. We start by showing that the technique is practical to use in the industrial context in terms of time of execution, then detail how we validated the visualisation tool using the help of Raincode Labs experts.

Analysis of the effects of the refactoring process

In the third and last part of this thesis, we present how we analysed the input and output of the automated refactoring process in order to show that its behaviour is preserved. We show, in chapter 7, how we analysed the process in order to find the code structures most affected by it, then extract them using semi-parsing. The chapter 8 shows how we validated our semi-parser by comparing it to a full parser available from Raincode Labs, both in terms of correctness and time of execution.

In chapter 9, we present the models we extracted from the code, as well as justify why the use of both Control-Flow Graphs and Labelled Transition Systems is necessary for our purpose. We explain how we applied the base trace equivalence algorithm to the models and then show the extension that we made to the base algorithm in order to better fit the use case. We validate our work in chapter 10, showing how much of the code from the industrial partner we were able to show as being trace equivalent.

In chapter 11, we detail how we created the visualisation tool for the trace equivalence between two programs. We explain how we adapted the algorithm in order for it to mark the models and how we make use of the two different models to produce a satisfactory visualisation. In chapter 12, we present how we validated the previously described visualisation tool with our experts.

Conclusion and future work

Finally, chapter 13 concludes this dissertation by presenting how we could improve the tools and techniques applied in this thesis in the future and discussing final lessons learned.

Background 2

In this chapter, we provide background information about our partner company (2.1) as well as a quick overview of the features of COBOL (2.2), the legacy language we will use in this thesis. Finally, we give a quick history of the disciplines of differencing (2.3) and refactoring (2.4).

2.1 Industrial context

Our partner company, Raincode Labs, provides various services in the realm of legacy code maintenance (more details can be found in section 4.1). We are interested in one of these services in particular: a large-scale automated refactoring process. In these projects, the experts from the company take generated COBOL code and refactor it through many *migration rules*, transforming the unmaintainable generated code into healthier COBOL. Much like smaller-scale refactoring tools, this process was created internally by Raincode Labs and behaves as a black box to outsiders.

As described previously, in these projects, Raincode Labs frequently finds itself in a position where they are unable to provide a clear explanation to their clients, which sometimes leads to tensions and misunderstandings. The experts expressed a strong desire for tools that can visually present and facilitate a clearer explanation of their work to clients. They seek a means to bridge the gap in understanding and demonstrating the details of their process in an externally accessible way.

Moreover, Raincode Labs expressed a need for tangible evidence that can support their claims and assertions. While their expertise is unquestionable within the organisation, the lack of external validation outside of their past projects poses a challenge when establishing credibility. Indeed, they have observed that despite being told that they have successfully migrated millions of lines of COBOL for other companies, their clients always believe that their particular codebase is different and special in a new way that makes the migration uncertain.

While clients do understand the COBOL code outputted by the migration process, they often ask questions about why it is the way it is. These questions are very difficult to answer by the experts due to the iterative nature of the refactoring, which applies many different rules on every line of code

in sequence requiring them to dig very deep into logs from the process to explain the effect of a single rule being triggered. Since the transformation is never done in one step, it is difficult to answer questions such as *"Why did this specific condition get flipped and not this other one?"*

In our case, we approached this use case of large-scale automated refactoring from two sides: first looking at the log of the process in order to help explain how exactly it happened and why one rule triggered rather than another; second looking at the code itself, comparing both versions before and after the migration to help show what changes were made and that they are behaviour-perserving. We also created visualisation modules to complement the algorithm by presenting the results in an intuitive and visually comprehensible manner, aiding in the communication and understanding of the differencing outcomes. We aim to provide our partner company with a practical and tailored solution to improve their communication with clients, validate their expertise, and enhance their overall service delivery.

2.2 A COBOL crash course

The collaboration established with Raincode Labs provides a valuable advantage by granting access to domain experts and the availability of authentic industrial-scale data (despite the data's confidentiality due to non-disclosure agreements).

This partnership necessitated a direct engagement with legacy code, specifically COBOL code in the context of our use case. While it is not imperative for the reader to be capable to create COBOL code from scratch, a basic understanding of this language will prove beneficial.

In this section, we explain essential COBOL concepts that will help the comprehension of the COBOL code fragments presented throughout this thesis. This knowledge will also help the reader to understand the challenges encountered during this work due to the nature of COBOL itself.

2.2.1 Before computer files: punched cards

When early programming languages like FORTRAN [Ora23] or COBOL [IBM23b] (COmmon Business Oriented Language) were created, writing code was very different from what we know today. Back then, programmers first wrote their code and then handed it to keypunch operators that used machines to manually create *punched cards* encoding the created program, each card representing one line of code. An example of a punched card can be seen on Figure 2.1.

The use of physical cardboard cards for coding presents several challenges. Firstly, making even minor changes to the code may require the complete re-punching of the cards. Moreover, ensuring the proper order of these

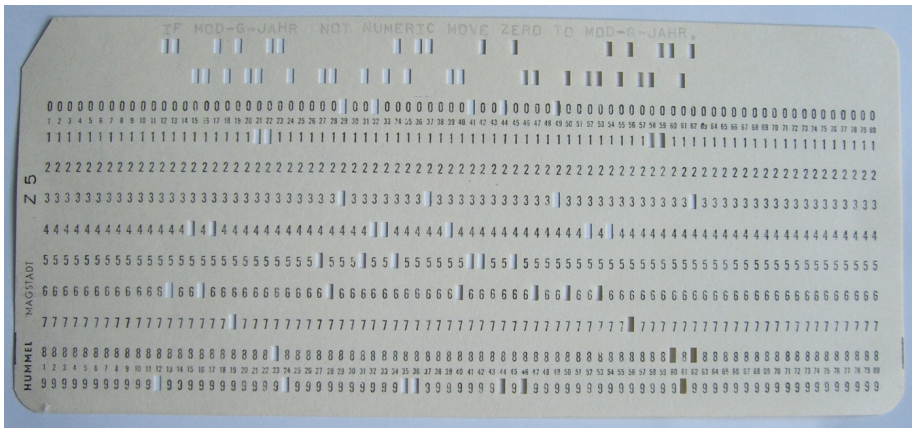


Figure 2.1: An example of COBOL punched card containing the code `IF MOD-G-JAHR NOT NUMERIC MOVE ZERO TO MOD-G-JAHR.` Author: Rainer Gerhards, image licensed under CC BY-SA 3.0

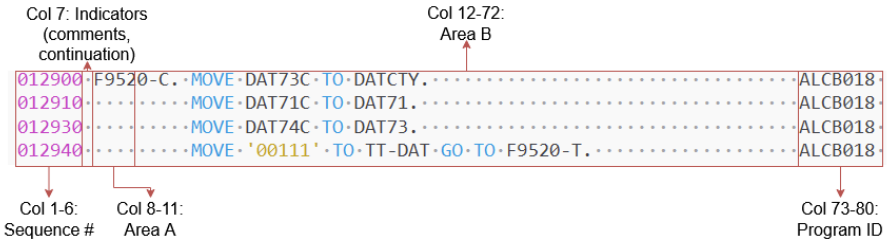


Figure 2.2: A generated COBOL program fragment: only columns 8–72, also known as "Area A" and "Area B", contain actual code (Image free of use found on Wikipedia)

cards is crucial, as the sequence of code execution directly impacts functionality. Without sequencing information, a shuffled stack of punched cards could render the entire set unusable. To address this, a specific portion of each card was dedicated to storing essential details like line numbers (also called sequence numbers) and the program ID. This design allowed for the reordering of card stacks (at first by hand, then by machine), even when mixed with cards from other programs. Figure 2.2 shows an example with a few lines of generated COBOL and illustrates all the dedicated sections in the card.

This means that only part of the card contained actual code, in the case of COBOL columns 8 to 11 (Area A, used for the start of the declaration of *labels*) and 12 to 72 (Area B, used for the declaration of statements etc.). Even if there is no more risk of mixing lines of code today, the use of only those columns for code stayed as all other columns are ignored by the compiler. In generated COBOL like the one seen on the figure, we still have the line numbers and

program IDs. We detail how we handled this in subsection 7.3.3.

2.2.2 Structure of a COBOL program

COBOL programs must follow a very strict structure. The most important component of such structure is also the topmost, the *DIVISION*. A COBOL division generally contains several *sections* which are themselves further divided into *paragraphs* (they are defined in Area A, the columns before the code). We will not focus on sections in this work, but we will use the paragraphs found in one specific division.

There are at most four divisions in a COBOL program:

1. Identification division. This must be the first division. It is meant to define details about the program like its author name, the date it was written, etc. It must also specify a *PROGRAM-ID*, the program name containing 1 to 30 characters. This will be present in our manually written examples since it is necessary for the program to be correct, but not used in practice.

2. Environment division. Specifies the systems on which the program is executed, as well as its input and output if any. This will not be used in this work.

3. Data division. Used to declare variables and parameters and specify their data types, size, etc. It is also not used in this work.

4. Procedure division. This is the main division of a COBOL program, containing all the logic and executable statements. This is the part of the code we will focus on when using COBOL code as input for our work.

Further dividing the divisions and sections are paragraphs. A paragraph starts with a user-defined name called a *label* (one paragraph with label *F9520-C* is shown in Figure 2.2) that can be referenced by statements such as `GO TO` and contains one or more *sentences*. A sentence is a combination of one or more *statements* and must end with a *period*. COBOL statements are equivalent to statements in other programming languages, we will present the ones that we will be using in this work in the next section.

The use of sentences in a programming language might be unfamiliar to most. As stated, a sentence contains one or more statements and ends with a period. That period, when met, closes all statements no matter how deeply nested they are. Changing the placement of a period can thus have strong semantic effects, which is why we need to pay special attention to them. We will demonstrate how periods change the semantics of code in subsection 7.3.5.

2.2.3 Statements that will be analysed in this work

An exhaustive list of all COBOL statements is outside of the scope of this thesis. However, we will describe all of the statements that will be used in our work. We will give an explanation as to why those statements were chosen in subsection 7.3.1.

2.2.3.1 Conditional statements

Two statements exist in COBOL to branch around conditions: the `IF` and the EVALUATE. The first one works exactly as in all modern programming languages, while the second is similar to a `switch` in Java.

To open and close some statements in COBOL, there are starting and ending keywords, much like the `if ... fi` that we can find in shell programs. In the case of IF in COBOL, we have `IF ... END-IF`.

It is important to note that COBOL allows a lot of variation in its syntax. In the case of the `IF`, for example, both `IF cond THEN body END-IF` and `IF cond body END-IF` are valid COBOL statements. In our industrial data, we found the second form (without "then") to be used exclusively, but we still made sure to be able to handle both variations. More so, since a period closes any statement, writing `IF cond body .` is also possible, and equivalent as long as the statement is not nested in others.

Finally, the conditions themselves can be expressed in various ways. COBOL supports contracted expressions, meaning that repeated variable names can be omitted. It also has keywords allowing to express some specific values (ZERO, SPACES, etc). The following list contains four variants of a condition that are all semantically equivalent:

- `IF VAR NOT = ' ' AND VAR NOT = 0 ...`
- `IF VAR NOT = ' ' AND NOT = 0 ...`
- `IF VAR NOT = ' ' AND 0 ...`
- `IF VAR NOT = SPACE AND ZERO ...`

Note that combinations of the above are also possible, even ones mixing AND and OR, rendering both the reading and the parsing of COBOL conditions challenging. This will be a point of attention in chapter 9.

2.2.3.2 Jump statement

Although less prevalent in contemporary programming languages, jumps are frequently used within COBOL. Two primary forms of "simple" jumps are pertinent to our analysis: GO TO and NEXT SENTENCE. The former, always

employed in conjunction with a label corresponding to a paragraph, continues the execution from said paragraph. On the other hand, the latter jumps the execution to right after the end of the current sentence, identifying the next period and executing the next statement after it. Listing 2.1 shows an example of the use of both statements (the statement DISPLAY is equivalent to `print`).

The NEXT SENTENCE of line 3, will jump all the way to the label *P2* on line 10 due to the period on line 9. The GO TO on line 12 will jump to line 1. Note that the DISPLAY on line 13 is dead code: it will never be executed. Either the condition B = 0 is false and the execution never enters the body of the IF, or the GO TO jumps before the statement can be executed.

Listing 2.1: Example of simple jumps

```
1     P1.
2        IF A > 0
3          NEXT SENTENCE
4        ELSE
5          DISPLAY 'Str1'
6        END-IF
7        IF B = 0
8          DISPLAY 'Str2'
9        END-IF.
10     P2.
11       IF B = 0
12         GO TO P1.
13         DISPLAY 'DEAD'
14       END-IF.
```

The jumps presented above simply redirect the flow of the execution. However, an alternative category of jump redirects execution to specific labels and comes back to its starting point after the execution of said label(s) is finished (when no other jumps break this flow). This is the case of the PERFORM and PERFORM THROUGH (sometimes also written THRU). Listing 2.2 is an example of the use of PERFORM: it is the same as the previous code sample, with editions only to lines 12 and 13. In this scenario, upon reaching line 12, the execution goes to *P1* as before. Subsequently, after having executed each statement within the paragraph *P1* for a second time, the execution trajectory returns to line 13, thus resulting in the execution of the DISPLAY statement. PERFORM THROUGH behave exactly as the PERFORM, except they redirect the flow of execution through a *series* of paragraphs (2 to n), then go back as explained before. Assuming there are paragraphs P1, P2, P3 and P4 in a program (in that order), the statement PERFORM P1 THROUGH P3 will execute all the code in P1, P2 and P3, then return to its starting point.

Note that all the statements described in this section do not have an accompanying END-.

Listing 2.2: Example of a returning jump

```
1      P1.
2          IF A > 0
3            NEXT SENTENCE
4          ELSE
5            DISPLAY 'Str1'
6          END-IF
7          IF B = 0
8            DISPLAY 'Str2'
9          END-IF.
10       P2.
11         IF B = 0
12           PERFORM P1.
13           DISPLAY 'NOT DEAD'
14         END-IF.
```

2.2.3.3 Loops

The equivalent to a Java for loop in COBOL is the PERFORM VARYING, sometimes called *inline perform* because it does not exit the paragraph it is in, contrary to the two previously shown statements, called *outline perform*. Listing 2.3 shows an example of a loop displaying the numbers 1 to 10. Contrary to the outline performs, the looping perform body is delimited by an END-PERFORM.

Listing 2.3: Example of a loop displaying numbers 1 to 10

```
1          PERFORM VARYING VAR FROM 1 BY 1
2                  UNTIL VAR < 11
3            DISPLAY VAR
4          END-PERFORM
```

2.2.3.4 Embedded SQL

To interact with databases, it is possible to embed SQL code within COBOL. The syntax is very simple: EXEC SQL tells the COBOL compiler that the text following must be interpreted as SQL, and END-EXEC returns to COBOL code. An example of embedded SQL code can be seen on Listing 2.4. We will use these pieces of SQL in chapter 7 to ensure that the refactoring done by Raincode Labs does not alter the semantics of the SQL statements.

Listing 2.4: SQL query embedded in COBOL

```
1          EXEC SQL
2            SELECT * FROM TABLE USER
3          END-EXEC
```

2.3 History of differencing techniques

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 made its first entry into the computer science world in the early 1970s, under the form of the Unix `diff` utility. Its first released version was shipped within the 5th edition of Unix in 1974 and developed by Douglas McIlroy and James Hunt. The first prototype was presented in 1976 [HM76], describing the original differencing algorithm that is now known as the *Hunt–Szymanski algorithm*. The algorithm is based on the problem of the longest common subsequence, where we want to find the longest subsequence of items that is present in both original sequences in the same order. In other words, we want to find the longest subsequence that can be obtained by deleting some data in both the source and target sequences.

To obtain a diff output as we know it today from such subsequence, we mark items that are in the source sequence but not the subsequence as deleted (-) and items that are not in the subsequence but present in the target sequence as added (+). For example, if the source sequence is "a b c" and the target sequence is "b c d", the longest common subsequence is "b c" and the diff representation will be "-a b c +d".

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 [Men02], compression [HVT96], convergence [Zay11] and clone detection [ML19].

Even today, many differencing tools still rely on the basic algorithm created by Hunt and McIlroy, 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 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 into 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 cause a file to be flagged as completely different, or when there were simultaneous changes to the same set of lines.

In this thesis, we set out to create differencing tools that take into account more semantic information about the code or do away with the idea of comparing text altogether. We will present related work on other forms of differencing in the next chapter.

2.4 Context on refactoring

As mentioned, ageing software tends to develop bad smells and become less habitable over time. When that happens, it becomes necessary to refactor the code in order to keep it maintainable. According to Opdyke, *"while refactorings do not change the behaviour of a program, they can support software design and evolution by restructuring a program in a way that allows other changes to be made more easily"* [Opd92]. Since this changeability is known to be related to the maintenance cost, it has become part of developers' everyday practice. The process of refactoring a piece of code can be performed iteratively by repeatedly applying simple refactorings until the code becomes habitable.

Since its writing in 1999, Fowler's book on refactoring [BF99] has grown very popular, providing a comprehensive catalogue of refactoring techniques to developers. In parallel, refactoring tools that can automatically apply refactoring techniques have also been created.

These tools are available for different programming languages and vary in automatedness, complexity to use and power level, as shown in 2006 [MS06]. Some are directly included in popular integrated development environments (IDEs) like Eclipse, IntelliJ, and Visual Studio. Those usually perform refactoring at a small scale, e.g. allowing a developer to rename all instances of a variable. Another popular form for refactoring tools is plugins. Examples like *BeneFactor* run in the background, parallel to the development process, sug-

gesting refactoring tasks that might be relevant to what the developer is doing [GDM12] while *JSparrow* focuses on smells and code optimization [IDE].

Refactoring tasks are divided into two categories: small and large-scale refactorings. The first category is also called floss refactoring and refers to small tweaks applied to a piece of code while making other changes. The second category is called root-canal refactoring [MB08] and refers to a time when refactoring is the only object, when correcting unhealthy code is necessary and the creation of new features is halted. It is sometimes referred to as reengineering [SN02] when the subject system is so deeply transformed that it is described as being reconstituted in a new form [Men+04].

The automated refactoring process that is the use case of this thesis is a root-canal refactoring, transforming generated COBOL code so deeply that it is unrecognisable, even by its owners. An example of how deeply the process affects the code will be shown in subsection 4.3.3.

2.5 Conclusion

In this section, we gave the intuition of the industrial context of our partner company Raincode Labs, as well as an overview of the key concepts of the legacy language that will be used in this thesis, COBOL. We presented the structure of the language as well as the statements that will be used in the next chapters to give context to the reader. Finally, we presented a history of the techniques of differencing and refactoring that are both key to this work.

Related Work

3

In this chapter, we will present tools and techniques that are adjacent to the work done in this thesis. We will explain how these different techniques are related to the work we produced and the context in which we created it. Finally, we justify the choices that we made when deciding on the directions taken in this PhD.

3.1 Advanced code differencing in general

Two approaches exist when performing code differencing: syntactic differencing, which compares the concrete or abstract syntax of the code, and semantic differencing, which compares the meaning of the code. Syntactic differencing identifies the change operations that allow to transform the source version of the code into the target version. The simplest way to perform syntactic differencing is the edit-distance algorithm which is the base of the `diff` Unix utility described in the previous section.

While syntactic differencing excels at pointing *where* change happens, the size of its output, often called *delta*, grows with the amount of changes made and tends to become overwhelming to analyse. Many tools doing syntactic differencing have been developed over the years, addressing the issue of improving the understanding of the output with various means.

The work done by the *srcML* community [MC04] with *srcDiff* [Dec+19; TRB18] aims at arranging changes in what they call *edit patterns*, that are structures showcasing changes in a way that is intended to be clear to developers. Other work focuses on reducing the size of deltas so they are more manageable, like *cdiff* [Yan91]. Yet another approach is to enrich the usual language of differencing, introducing the concepts of move and rename to the usual add and delete, like in the tool *GumTree* [Fal+14]. *truediff* [ESP21] is another example of syntactic differencing that creates patches (groups of changed logically linked together), in a similar way to *GumTree*. This tool also focuses on type safety, which is not the case of other tools.

Those techniques can greatly improve the quality and readability of deltas. The nature of refactoring tasks is particularly suited for some of them, for example the introduction of the *move* changes since it is very frequent to move entire methods when refactoring code. We therefore tried *truediff*, hoping to

be able to highlight the effects of refactoring rules. However, the refactoring process performed by our partner company is at a scale too large to analyse every program. Even with outputs as concise as the one of *truediff*, with several millions of LOC and over 3000 programs in the example that we have (which Raincode Labs considers of middling size), it simply is unfeasible to analyse deltas for every program. This explains why we discarded the idea of using syntactic differencing in this work.

The other way to perform differencing, semantic differencing, does not look at the syntax of the code but rather focuses on extracting its meaning in order to express how it changes. Identifying the meaning of the code requires at least some background knowledge of the target programming language. Because of that, semantic differencing tools have been developed for specific programming languages like Java (*JDiff* [AOH04], *CLDIFF* [Hua+18]) or C (*Dex* [Rag+04]). While we did not find any tool performing semantic differencing for our target language, COBOL, the analysis of semantic-differencing tools inspired us to take that route for the second part of this thesis. In chapter 7, we show how we chose the parts of the code most affected by the refactoring process and extracted their meaning.

Finally, we found differencing to be used as a means to help developers perform other tasks, like handling changes in API calls (*Diff-CatchUp* [XS07]), bug detection (*LSdiff* [KN09]) or clone detection (in the work of Kim et al. [KNG07], in the tool *ROSE* [Zim+04] or in *CloneDiff* [XXJ11]). These approaches inspired us to look beyond the basic idea of showing what changed in the code and to use differencing in order to assist in the comprehension of the automated refactoring process itself.

3.2 Differencing of non-code artifacts

To have more concise outputs, we decided to abstract away from the code entirely, focusing our attention on explaining the refactoring process itself. For this, we turned to models.

The modelling community has produced many tools allowing to perform model differencing. They exist for most of the widely-used model types like UML (e.g., *UMLDiff* [XS05]), activity diagrams (e.g., *ADDiff* [MRR11]) or feature models (e.g., in *FAMILIAR* [Ach+12]). Zhenchang Xing even proposed a model-agnostic tool [Xin10]. Some of these tools aim to provide syntactic differencing while others are more semantic-based, with the work of Maoz [MR15] bridging the gap between them in a tool that proposes both.

However, no model of the automated refactoring process exists within Raincode Labs. While work has been done on automatically generating UML models from natural language [DB09] or use case scenarios [Gut+08; YBL10], another non-code artifact was readily available: log messages.

We discovered that some work has been done in differencing log messages produced by processes and chose to explore this avenue in this thesis. We adapted the log-based behavioural differencing algorithm proposed by Goldstein et al. [GRS17]. Other work in the domain includes Li et al's work [Li+19], who have investigated log differencing with the intention to detect unwanted duplicate log messages, such as those stemming from cloning logging code without appropriate adjustments. They detected five different patterns of harmful log message clones and developed a tool called *DLFinder* [Li19].

Gholamian and Ward [GW20] developed another tool, *Log-Aware Code Clone Detector* (LACC), that is capable of differencing two code fragments and predicting the location of a log point in one version based on the existence of a log point within another version. Tama and Comuzzi [TC19] have tried using 20 different classifiers to analyse logs in order to build models to predict next events by looking at their history. Perhaps the closest to our proposed approach (see chapter 5) is the paper by Bolt et al. [Bdv18], where the tool *ProM* was written with the goal of comparing two processes based on their event logs and producing concise results. There are more examples like this in the domain of process mining, such as the work on differential perspective graphs [Ngu+18] which in software engineering would be analogous to grammatical inference.

3.3 Comparison of code in the context of refactoring

Since this thesis was done with an industrial partner and in the context of an automated refactoring tool for legacy systems, related work on refactoring is relevant to us. A substantial body of academic research exists on refactoring of legacy systems in particular, with different objectives. The goal that we decided to pursue is to show that the automated refactoring process performed by Raincode Labs does not alter the behaviour of the programs.

During our analysis, we found several techniques offering refactoring methodologies for legacy code, which places them closer to the process proposed by Raincode Labs and further from what we want to achieve. This is the case of Liu et al. [LBL06] who performed feature-oriented horizontal decomposition of the code, relating code refactoring to algebraic factoring and creating a refactoring methodology. Bayer et al. [Bay+99] aim to produce product lines, Ribeiro and Borba [RB08] made a tool to recommend refactorings for an already existing product line. Raghavan [Rag02] proposed to separate common interfaces from components. Chiang and Lee [CL04] moved towards distributed architecture by putting legacy code in wrappers. Kolb et al. [Kol+06] specifically wanted improved reusability of code elements without damaging their performance. Khatchadourian and Matsuhara [KM17] wanted to change the existing code to make full use of newly introduced language features, etc.

We did find some works that performed differencing, even though the actual comparison can be done in different manners and have different goals. This is the case for Ying and Miller [YM13] who tested the pre- and post-refactoring versions only for performance. One of the projects closest to what we did in chapter 9 was the one by Schuts et al. [SHV16] who extracted Mealy machine models from a legacy system and its refactored version, in order to demonstrate their equivalence to their industrial partner. In their case, they aimed at improving the refactoring process itself, more so than showing that it does not risk creating bugs. Other tools like *REdiffs* [HTS13] or *RefFinder* [Kim+10] focus on untangling changes due to refactoring from other changes made to functionalities, which should not be applicable in our case since no new functionalities are introduced by Raincode Labs.

In the context of program comprehension and/or source code analysis, it is often important to detect refactorings from existing commits. There are many contemporary tools for doing that in C++ [Hem+21], Java [TKD20; Hem+21; Sil+20; Pre+10], JavaScript [Sil+20], Kotlin [Kur+21] and Python [Atw+21; Dil+22]. More generic tools targeting the detection of refactoring in object-oriented programs also exist [DDN00] as well as tools looking to detect a particular kind of refactoring [SPD13]. Finally, attempts have been made at a language-agnostic refactoring tool [Tic+00]. While these approaches are interesting, detecting refactoring is not enough for our purpose.

Instead, we took inspiration from work focused on proving that modifications made to a program are behaviour-preserving [MDJ02], more specifically work done in the context of refactoring [Men+04]. In the same context, we found some analysis suggesting that refactoring does not always change metrics like cyclomatic complexity [SD10]. Another point of view is to identify transformations that are not behaviour-preserving and how they are linked to refactoring [Anq+22].

From the language constructs that we extracted in chapter 7, we created control-flow graphs and labelled transition systems. We found that comparison of control-flow graphs is a technique commonly used in the domain of computer security. It is used by Bruschi et al. [BMM06] in the context of detecting self-mutating malware. Both Behera et al. [BSL19] and Nagarajan et al. [Nag+07] used control-flow to counter obfuscation techniques. Le and Pattison [LP14] have used control-flow graph comparison for the sake of patch verification, while Awadhutkar et al. [Awa+22] used it to prevent compiler trap doors. Lim [Lim14] compared control-flow graphs on the level of entire blocks. Comparison of labelled transition systems is an even older topic, well-investigated at least since the 1980s in the testing context [De 87].

When used in the domain of computer security, many examples of malware code are often available and can be translated into models that serve as a basis to create a pattern miner or as training data for machine learn-

ing algorithms. This does not apply to us, which is why we chose to go the route of labelled transition system and trace equivalence in chapter 9. In this context, we found one more work similar to what we did but with another goal in mind, Ordóñez-Camacho et al. [Cam+10] who used a combination of lightweight semantic language annotations and bisimulation on control-flow graphs to verify equivalence of an original program and its translated version in the domain of spacecraft mission planning. Here again, the main goal was the creation of an automated program translator rather than the equivalence.

3.4 Visualising differences

In order to scale to the size of the industrial data of Raincode Labs, we worked with models (Finite State Automatons and Control-Flow Graphs). To best present the results of our chosen techniques, we created two visualisation modules. Since both our outputs are models, we analysed the state of the art on graph visualisation.

The discipline of creating clear and concise visualisations for graphs has been studied in depth [HMM00; Pur00]. Over the years, several frameworks have been created to help the endeavour of creating graph visualisations, for example *Tulip* [Aub04] or *WiGis* [Gre+09]. We picked *Roassal* [Ber+21] as our visualisation framework of choice because it gave us the opportunity to collaborate with its creator and visualisation expert, Alexandre Bergel. His feedback was instrumental in creating user-friendly visualisation tools.

Finally, we found two works on visualisations in the context of refactoring. First, Kanda et al. [KSI22] who visualised changes in program behaviour, in the context of reviewing bug fixing. They present a view very close to the standard differencing seen on GitHub for example, but augment it with information about the changes in variables (names and types). In the context of the large-scale refactoring of Raincode Labs, we chose not to work with variables and therefore this does not apply to our purpose. Second, Van Rysselberghe and Demeyer [VD04] used visualisation to identify relevant changes before re-engineering of a large software system. Their approach is very high-level, aiming to identify e.g. the code structures that are stable and do not change (much) throughout a software lifecycle. This is again not applicable to our purpose since no parts of the code is untouched by the automated refactoring process of Raincode Labs.

3.5 Conclusion

In this chapter, we presented work related to the techniques and tools created during this PhD thesis. For each of them, we explained how it can be related to our context of helping Raincode Labs increase their clients' trust in their

automated refactoring process. We present the techniques that we used, the ones that we took inspiration from, as well as the ones we discarded because they could not be applied to our use case.

Case study

4

4.1 Partner company: Raincode Labs

Raincode Labs originated as Phidani, a company established in 1990 by Darius Blasband, who is still CEO today. Initially, Phidani offered a range of IT services, including development and networking, without specialising in any specific technology.

Around 1993, as many believed Object-Oriented Programming (OOP) to be the future of software development, Darius Blasband held a different perspective. According to him, COBOL, an imperative language that had been popular since its creation in 1959, would continue to be relevant. Given the vast number of COBOL codebases in industrial environments, Darius was convinced that this technology would persist for a considerable period.

In 1995, Phidani recognised the need to concentrate its efforts on a particular domain to thrive. Drawing from Darius's PhD thesis [Bla98] where he created a programming language called YAFL and his conviction that dependence on ageing mainframe systems would become increasingly costly for companies, Phidani shifted its focus towards legacy migration.

Their initial significant projects involved code restructuring, which entailed transforming code generated by fourth-generation languages (4GLs) into plain COBOL, thereby eliminating dependencies on the generator. They began with IDEAL DATAKOM [IBM23a] and PACBASE [IBM20], initially performing the work manually and gradually developing automated tools to facilitate refactoring large volumes of code. Around that time, Phidani rebranded itself as Raincode.

In 2016, Raincode spawned several subsidiary companies, including Raincode Labs, which provides compiler services. In addition to its restructuring projects, Raincode has developed its own compilers, starting with COBOL, and then expanding to PL/I and assembly. Their compilers received technical achievement awards from Microsoft.

Today in 2023, Raincode Labs presents itself as compiler experts, aiming in part to liberate companies from reliance on 4GLs and other mainframe-related dependencies. They offer customised services to different companies, utilising a suite of in-house tools and developing new ones on a case-by-case basis. Providing a comprehensive overview of their past projects is beyond

the scope of this thesis, as our focus will be on a specific use case related to PACBASE migration [Rai18], explained in detail further in this chapter.

Raincode Labs, as a branch of Raincode, also strives to maintain its presence and engagement in the academic realm, both for visibility and to stay up-to-date with the state of the art. This is why they have partnered with universities such as UCLouvain, VUB, KULeuven, ULB, UvA and UTwente on various master theses, PhD and research projects, seeking to collaborate and address various challenges they encounter in their processes. In our particular case, we addressed concerns related to code differencing, specifically in the context of PACBASE migration.

4.2 PACBASE

We will now shortly describe what PACBASE was at its creation as well as its status today. Since this technology is the reason for the existence of our use case for this entire thesis, it was important to give some context to the reader.

PACBASE is a language and 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. This 4GL was widely used since its creation and throughout its life cycle [Alp87], meaning that still to this day, many companies depend on it.

The "Compagnie Générale d'Informatique", which developed PACBASE, was absorbed by IBM in 1999. In 2000, PACBASE itself was modernised and rewritten in Java [Rém00], 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 announced in 2015 [Hew12]. 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 undertaken many projects of PACBASE-generated COBOL to plain COBOL migration, one example being the case of the insurance broker DVV Verzekeringen reported by Bernardy [Ber02]. The experts take PACBASE-generated COBOL code and refactor it into a shorter, more readable equivalent that can be maintained manually [Rai18].

It is important to note that the main focus of this chapter is not on how PACBASE works but on the migration performed by Raincode Labs and on how advanced differencing techniques can improve that process. However, more details on PACBASE and its documentation can be found online [IBM23c].

```
F05DC.
    MOVE        1       TO ICATR.
    GO TO       F05DC-B.
F05DC-A.
    ADD         1       TO ICATR.
F05DC-B.
    IF          10      < ICATR THEN
      GO TO     F05DC-FN
    END-IF.
    IF      CATX(ICATR) = '0' THEN
      NEXT SENTENCE
    ELSE
      GO TO F05DC-C
    END-IF.
    MOVE 'X' TO CATM(ICATR).
F05DC-C.
    IF      CATX(ICATR) NOT = '0' THEN
      NEXT SENTENCE
    ELSE
      GO TO F05DC-D
    END-IF.
    MOVE 'Y' TO CATM(ICATR).
F05DC-D.
    GO TO F05DC-A.
F05DC-FN.
    EXIT.
```

```
PERFORM
    VARYING ICATR   FROM 1
                    BY 1
    UNTIL 10 < ICATR
    IF CATX(ICATR) = '0' THEN
      MOVE 'X' TO CATM(ICATR)
    ELSE
      MOVE 'Y' TO CATM(ICATR)
    END-IF.
END-PERFORM.
```

Figure 4.1: Example of a migration from PACBASE-generated COBOL to equivalent COBOL code that is more readable and maintainable.

4.3 The PACBASE migration process

The goal of a PACBASE migration project is to allow the client to get rid of its dependency on PACBASE and allow them to switch to pure COBOL code. PACBASE does generate COBOL, but it is not usable as is, as can be seen on the left-hand side of Figure 4.1. After the migration, the client receives a new version of the codebase that has been refactored to be more readable and maintainable, as can be seen on the right-hand side of Figure 4.1.

The PACBASE migration process is performed through a set of about 140 transformation rules that Raincode Labs developed and refined over the last 20 years. Each individual transformation rule can be seen as a local automated transformation/rewriting/refactoring designed to be simple enough to not change the semantics of the code; yet making it just a bit more concise, readable or maintainable.

Most rules alter the statements of the code, while others alter only the appearance of it, namely ones of formatting. An example of a statement-altering rule would be *Merge IFs*. As suggested by the name, it will merge IF statements if possible. An example of this is shown in Listing 4.1. Since the body of the first if contains only the body of the second, it would be better to refactor them into a single if, joining their conditions with an AND as shown in Listing 4.2. This is a fairly simple but representative example. Other rules like this include the refactoring of GO TO into loops (explained in subsection 4.3.3)

and simpler rules like *Remove Useless DOT* that remove the superfluous . that are overly-present in PACBASE-generated COBOL.

Most formatting-only rules are optional and designed to make the output best fit the client's coding standards. Those rules include *Comment Line Before/After Section* or *Blank line around Loops and IF*.

Listing 4.1: An example of code to refactor

```
1 IF A = 0
2   IF B = 0
3     *statements here
```

Listing 4.2: The refactored version of the example

```
1 IF A = 0 AND B = 0
2   *statements here
```

It is important to note that, while some of those transformations have been formally proven to maintain the behaviour of the code, most were just developed to fit clients' needs and tested internally at Raincode Labs. The rules that have been manually proven are some of the ones that handle conditional logic, i.e. the grouping of IF shown above. They only represent a very small percentage of the total existing rules. This lack of formal proof that no bugs can be introduced by the process tends to worry code owners as we will see later.

PACBASE migration projects can be described by using several phases, some involving both clients and experts, some only concerning experts. A schema of the process and who is involved when can be seen on Figure 4.2: first, there is a preliminary step where the client picks the set of rules and the experts generate a first output to test. Then the migration is done exclusively by experts and finally the post-project adjustments require feedback from the clients. We will explain all the parts of this process in the following sections. The artifacts needed and created in this process are detailed in Figure 4.3.

4.3.1 Preliminary steps

The first preliminary step of each migration project starts with meetings, where the existing 140 migration rules are presented by Raincode's experts and selected by the client, to create a custom set for each project. In such meetings, the client sends both managers and more technical employees.

As mentioned earlier, the available transformation rules include examples such as the universally appreciated and always selected 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.

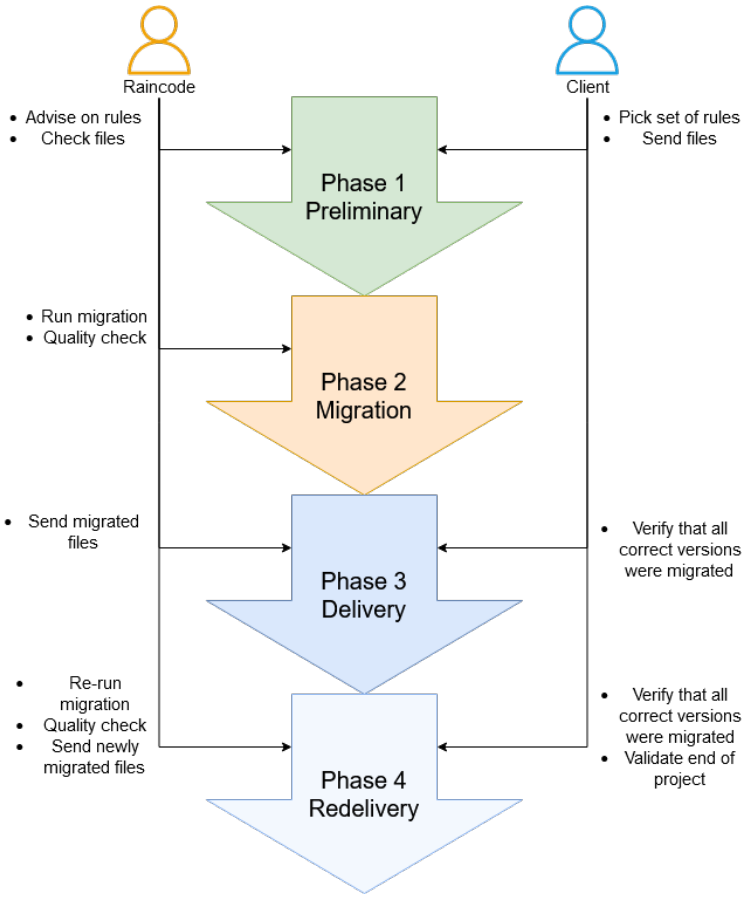


Figure 4.2: Summary of the steps of the migration process, showing who does which task

Raincode Labs' migration engineers coach the client in choosing which rules to include, give suggestions about what would work best, and then provide a sample of a few programs migrated using the current set of rules. This allows the client to concretely see the effects of the transformations and adjust the rules if needed. Usually, a few iterations of this process take place.

Once the set is decided, another preliminary step starts, although it is mainly meant to increase the trust of the client in the migration. While Raincode's engineers are accustomed to the process and confident in its capabilities, the code to be migrated is quite often part of the core business of a client, making it critical to ensure that no bugs were introduced before putting anything in production. For this, the migration is run for the first time, so the experts can determine how best to convince the client that all functionalities

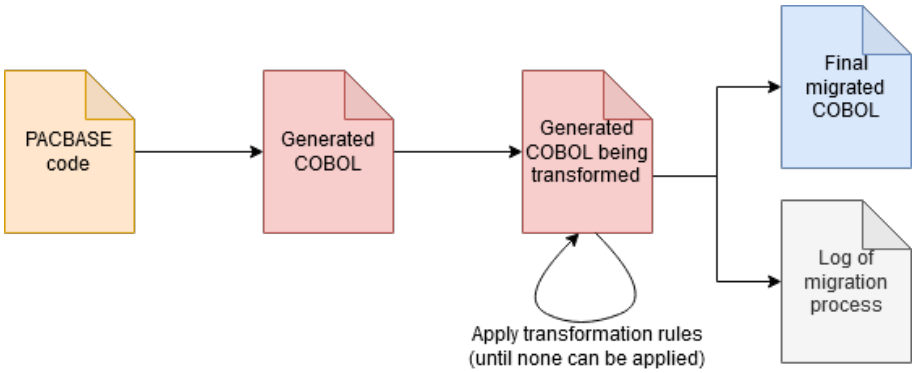


Figure 4.3: Summary of the migration process, showing the different artifacts and their transformations

are maintained. Based on this first run, the codebase is then partitioned into three:

- 10–30 critical programs to be tested exhaustively (0.3–1%);
- 80–100 programs to be tested thoroughly with unit and integration tests (2.6–3.3%);
- the rest of the programs, to be tested for integration or not tested at all.

It is verified that in the first partition, all transformation rules that were triggered in the migration process are applied at least once, ensuring that all rules get manually tested by the customer. However, since the testing that needs to be performed is partly manual, the customer will often want to negotiate this partitioning. Some programs might be too big to test manually, some might already have a robust test suite, and so on. In practice, most clients give the green light to go ahead with the real migration without being completely certain of its effect.

When all parties are in agreement, the customer's entire PACBASE-generated COBOL codebase is sent to Raincode engineers, and the project moves on to the next phase.

4.3.2 The migration process in detail

Once all the files are received, they first go through a *manual analysis* to make sure that everything needed is there, then the migration itself is run, and finally, a *quality check* is performed by Raincode engineers. All in all, this process takes from 2 to 4 weeks depending on the size of the codebase.

During that time, it is common that the client keeps working on the code and requests the new versions of some files be migrated as well. This is called a *redelivery* and will be detailed later.

4.3.2.1 Data collection

This first phase of a migration (shown in Figure 4.4) exists mainly to check that everyone agrees on what needs to be migrated and that all needed files are there. Experts perform a cursory analysis of the codebase since it is common, due to the scale of the codebase (typically 10–200 MLOC), that the received files contain uncompilable code or non-code artifacts, or even missing pieces (often, the missing files are *copybooks*, files that are imported in the main code and are needed to compile it). An automated script helps to show if anything is missing and if needed, emails are exchanged to ask the client for clarification or updated data.

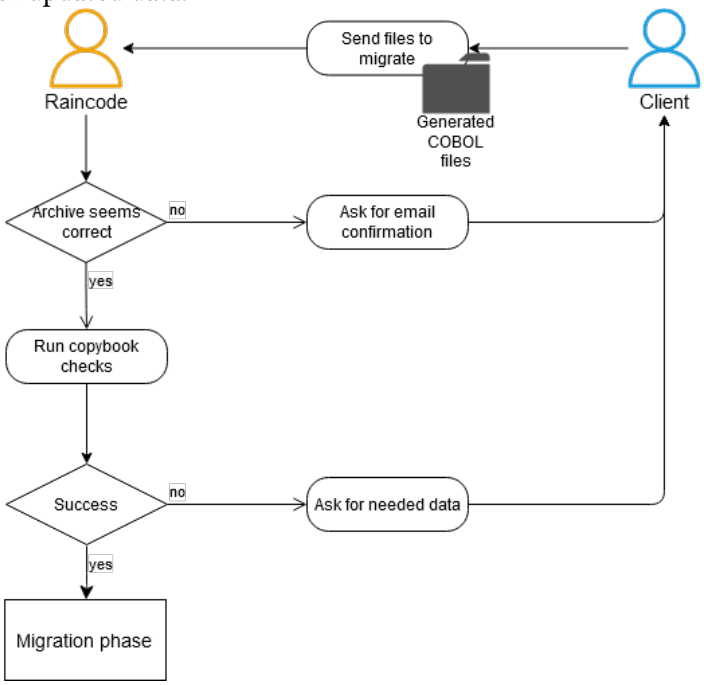


Figure 4.4: Data collection phase

4.3.2.2 Migration

When all the necessary files are available, the project moves on to the migration phase depicted in Figure 4.5. It is run on all provided files, outputting the migrated files, a log of the process, but also a summary constructed from the

logs that contains all migration rules that were applied and any errors that occurred.

The logs themselves provide more details about where and when migration rules were applied as well as all errors or warnings, but are lengthy and hard to read through, which is why the summary is checked first. The automated migration process is itself divided into several independent parts called *rea*, *reb*, *rec*, *red* and *ref*. It starts with two preprocessing phases, then the main migration phase *rec*, then phases of clean-up and formatting.

In this phase, the engineers focus on the errors and warnings caused by the migration rules. Both are fairly common since the migration process is ever-evolving and every new project brings corner cases that were not yet encountered and accounted for. Any error or warning might cause bugs that would be due to the migration itself, so the engineers will manually go through the summary, fixing every error or warning and re-running the migration until no more arise during the process. When this is done, they double-check that no error was missed by performing a `grep` on the logs, expecting an empty result to move to the next phase.

4.3.2.3 Quality check

Finally, the project enters the quality-check phase shown on Figure 4.6. Now that no error or warning was flagged during migration, the engineers take the time to look at the output of the tool Meld [Sof23], a visual diff utility allowing them to look at changes between the previous and new versions of a few migrated programs. This is done manually and is a heavily subjective process. Armed with years of experience, the engineers check that, using their wording, "the migrated program looks good enough". After both having observed and listened to them doing this task and conducted rigorous interviews, we understood that they try to match the kind of transformations (loop rewriting, renames, deletions) they have seen in the logs to what they see now. If it doesn't look "good enough", they will again make an effort to understand why, fix the migration process, and re-run it again. An example of "not good enough" would be for them to find a piece of code still containing a lot of GO TO that could be transformed into a loop. They would then investigate why the migration rule rewriting that kind of structure could not be applied there and if possible, change it slightly to allow it to trigger.

When the engineers are confident that their migration didn't produce any unexpected transformations, they check the rest of the summary mentioned above. It should contain no more errors but contains a few metrics that will be sent over with the refactored files, i.e. the file size reduction in terms of LOC as well as the amount of GO TO deleted. Sending all these metrics and the refactored code is called a *delivery*.

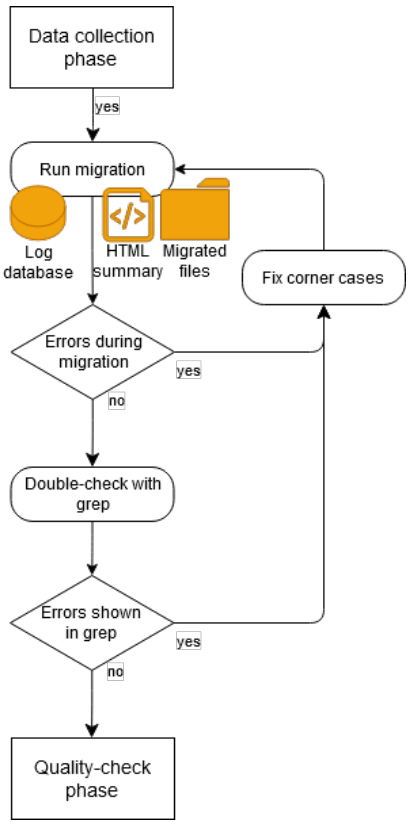


Figure 4.5: Migration phase

Since the three previous phases tend to take 2 to 4 weeks and the codebase is critical for the client, it is impossible for them to freeze all development for that long. It is thus likely that, while Raincode is performing the migration, at least part of the files have been modified on the client side. When the migrated code gets delivered, in the meantime the client often has a new version of a few programs with bug fixes or new features that they wish to be integrated as well. This triggers another phase of the project, called a *redelivery*.

4.3.2.4 Redelivery

When it is necessary, a redelivery is done in a very pragmatic way: rather than trying to integrate the client's changes into the migrated code, the new versions of the files are simply migrated again. Like a delivery, this process also takes time, meaning that in the meantime the client may have changed the codebase a bit more. This may then lead to another redelivery, hopefully with an even smaller set of changed programs. On average, migration projects contain 2 to 3 of such redeliveries.

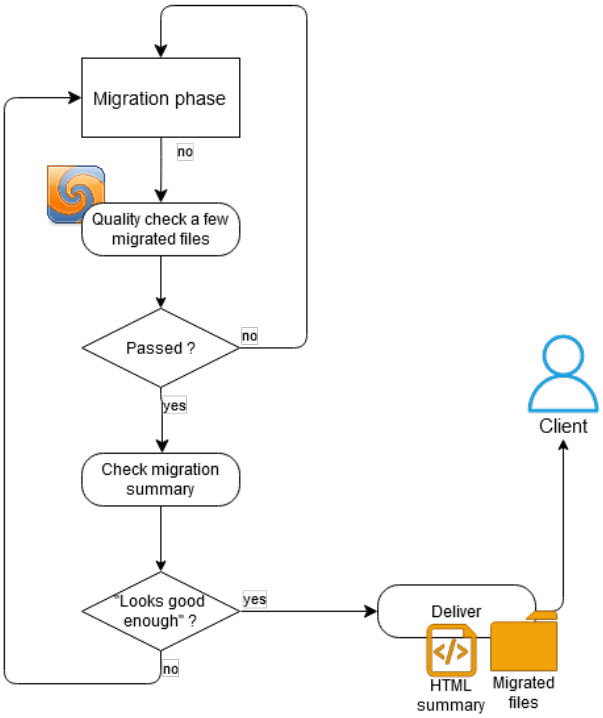


Figure 4.6: Quality-check phase

Since a redelivery is the same process as the first migration or delivery, it has the same phases. However, a bit more manual work is done to ensure that everything goes smoothly.

In the data collection phase, the experts check not only that they have all the necessary files, but also that all the files that they were given have indeed changed since the last iteration. For this first verification, the Raincode engineers use Meld again.

The migration phase happens exactly as before, but the experts go a bit deeper in the quality check when looking at the summary of the migration.

When doing the first migration and a delivery, since we are just after the first testing phases, all the migration rules triggered have been internally tested by the client. However, in a redelivery, it is possible that the new code triggers a rule that wasn't previously tested, and this information is shown in the summary. This usually creates issues since the client is often wary of putting untested code in production. An easy fix would be to thoroughly test the flagged file(s), but since testing is a lengthy and costly process, clients are often reluctant to go through a test phase again, potentially sending the project into an infinite loop of waiting, argumenting, and redelivering.

This is why the engineers often go a step further and perform an extensive

investigation phase shown in Figure 4.7. First, they manually sift through the logs to find exactly at which iteration of the migration and on which line the new migration rule was applied. Then, they use the migration tool to generate two intermediary versions of the program that need to be tested: a first version with the program migrated until just before the rule was triggered (and should therefore be correct) and a second version being the next iteration of the migration with the migration rule applied. The engineers are then able to pinpoint exactly the lines that are affected and may need to be tested by the client, significantly reducing the work needed to do so. In that case, the redelivery is sent with that extra explanation, and it is up to the client to test or not.

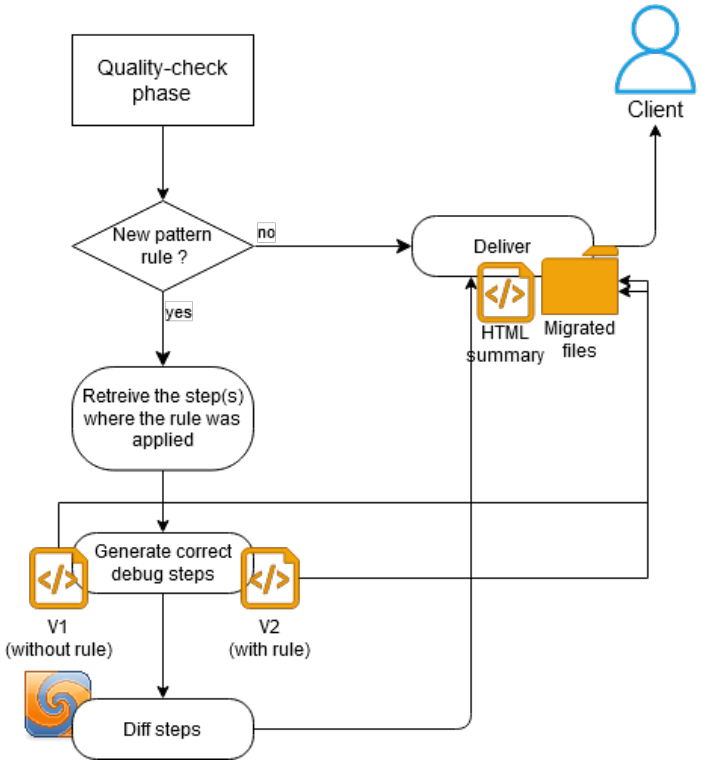
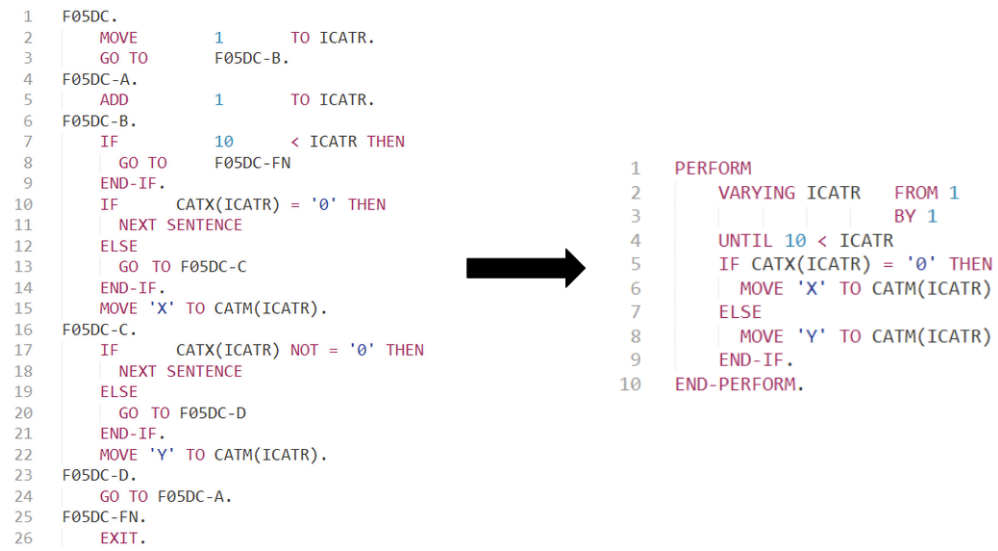


Figure 4.7: Investigation phase

4.3.3 Concrete example

An example of such a COBOL-to-COBOL program transformation is given in Figure 4.8 (it is an exact copy of Figure 4.1, included for ease of reading): 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



```
1  F05DC.
2      MOVE        1       TO ICATR.
3      GO TO       F05DC-B.
4  F05DC-A.
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 4.8: Copy of Figure 4.1: Example of a migration from PACBASE-generated COBOL to equivalent COBOL code that is more readable and maintainable.

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.

Transformation rules are applied iteratively. For example, it takes 33 intermediary steps to perform the migration from Figure 4.8. 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 with the instruction `CONTINUE`. As a reminder, `NEXT SENTENCE` jumps to the instruction after the next period (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 period is right after the end of the if statement, where execution naturally continues.

Some transformation rules remove artifacts that are no longer useful. An

example of such a rule would be **Remove Useless Dots**, which is applied four times to the code a few steps later. Indeed, since we removed our NEXT SENTENCE instructions, we do not need the periods signalling such sentences anymore. Therefore, the periods 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 4.8, may require applying quite a few intermediate transformation rules. To remain relatively concise, we will highlight only a few of them in Figure 4.9. 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 4.9a). Then, a transformation rule **Recognise Loop-like Patterns** is triggered twice in a row, and after some clean-up steps, we can see an if-else structure containing GO TOs that starts to resemble a loop on lines 9, 10 and 21 (Figure 4.9b). Quite a few more steps are needed, however, to bring all the conditions into the simple if-else clause that we get at the end (Figure 4.9c); before the **Replace GO TO by Loop Exit** transformation rule can finally create the PERFORM loop that we can see in Figure 4.9d. The final steps create the VARYING and performs some cleanups, resulting in the code on the right-hand side of Figure 4.8.

Note that the example that we presented is handcrafted. It is made to be understandable and not representative of a real PACBASE-generated COBOL file. The scale of the PACBASE projects that Raincode Labs undertakes varies greatly. For the purpose of this thesis, we were given an example of such a project, which the company considers to be of small-to-middle size. It is composed of 3014 different programs, ranging from 120 LOC to more than 20 KLOC (average is around 2500). Running the migration for the entire project takes around a week-end when performed on the servers available in the company. This project had a delivery, then two redeliveries of 39 and 7 files, respectively.

4.4 Conclusion

In this chapter, we have presented our partner company Raincode Labs and the context in which it operates in the industrial world which is legacy migration. A part of this is refactoring non-human-understandable generated code into a maintainable version of itself, in order to get rid of the dependency on the generator. We have also detailed the industrial process that will be the use case for both the tools presented in this thesis, PACBASE migration. We

explained every step of such projects, highlighting both their strengths and weaknesses.

```
F05DC-A.
    ADD 1 TO ICATR.

F05DC-B.
    IF 10 < ICATR
      GO TO F05DC-FN
    END-IF
    IF CATX(ICATR) NOT = '0'
      GO TO F05DC-C
    END-IF
    MOVE 'X' TO CATM(ICATR).
F05DC-C.
    IF CATX(ICATR) = '0'
     GO TO F05DC-D
    END-IF
    MOVE 'Y' TO CATM(ICATR).
F05DC-D.
    GO TO F05DC-A.

F05DC-FN.
    EXIT.
```

(a) Flipped conditions without `CONTINUE`

```
F05DC-A.
    ADD 1 TO ICATR.

F05DC-B.
    IF 10 < ICATR
      GO TO F05DC-FN
    ELSE
      IF CATX(ICATR) = '0'
        MOVE 'X' TO CATM(ICATR)
      END-IF
    END-IF
* Raincode removed Label  F05DC-C.
    IF CATX(ICATR) NOT = '0'
      MOVE 'Y' TO CATM(ICATR)
    END-IF
* Raincode removed Label  F05DC-D.
    GO TO F05DC-A.

F05DC-FN.
    EXIT.
```

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

```
* Raincode removed Label  F05DC.
    MOVE 1 TO ICATR
    .


F05DC-B.
    IF 10 < ICATR
      GO TO F05DC-FN
    ELSE
* Raincode removed Label  F05DC-C.
    IF CATX(ICATR) = '0'
    MOVE 'X' TO CATM(ICATR)
    ELSE
    MOVE 'Y' TO CATM(ICATR)
    END-IF
    END-IF
* Raincode removed Label  F05DC-D.
* Raincode removed Label   F05DC-A.
    ADD 1 TO ICATR.
    GO TO F05DC-B.

F05DC-FN.
    EXIT.
```

(c) Simplified if/else clause doing the MOVEs

```
* Raincode removed Label  F05DC.
    MOVE 1 TO ICATR
    .


F05DC-B.
    PERFORM UNTIL FALSE
     IF 10 < ICATR
       GO TO F05DC-FN
    ELSE
* Raincode removed Label  F05DC-C.
    IF CATX(ICATR) = '0'
    MOVE 'X' TO CATM(ICATR)
    ELSE
    MOVE 'Y' TO CATM(ICATR)
    END-IF
    END-IF
* Raincode removed Label  F05DC-D.
* Raincode removed Label   F05DC-A.
    ADD 1 TO ICATR
    END-PERFORM.

F05DC-FN.
    EXIT.
```

(d) Creation of the `PERFORM`

Figure 4.9: Snapshots of the loop-creation process

Part II

Analysis of the refactoring process

This part of this thesis is dedicated to enhancing the comprehension of the migration process itself. The primary objective is to investigate the interplay of rules during the migration process, notably helping the explanation of how changes to the input or the configuration file can affect the exact set of rules being applied during the process. In the context of changing the configuration file, the removal of a single rule can lead to the non-application of other rules during refactoring due to the removed rule's role in establishing preconditions for other rule executions. This is really confusing to clients since it is impossible for experts to foresee and explain every possible interaction when adding or removing a rule from a configuration file.

To analyse this, we shift the focus away from the code and instead concentrate on the logs generated during the migration process. We aim to discern and comprehend the effects resulting from the alterations in either the input file or the configuration file.

The initial step involves processing the logs generated by the migration process. We automatically extract pertinent information, such as the names of executed rules, while disregarding any error or warning messages since they are not relevant at this stage of the process. We then translate those logs into a Finite State Automata (FSA) using an external tool.

Subsequently, we employ a log-based behavioural difference algorithm that takes as input the FSAs and compares them, enabling a comprehensive depiction of the entire migration process and the observed changes.

The next phase centers on the visualisation of the identified changes in an interactive manner. As part of this endeavour, we identify limitations in a previously proposed visualisation approach. We then address these limitations, accommodate the complexities of our industrial scenario and propose a more scalable solution.

Finally, we validate our process, both in terms of its time efficiency and in terms of the practicality of our visualisation tool. We aim to prove that the changes to the visualisation we propose do solve the issues that we discovered and engage Raincode experts in the evaluation process, conducting semi-formal interviews.

Log-based behavioural differencing

5

5.1 Context

This chapter focuses on the first differencing technique applied in this PhD, log-based behavioural differencing. Our goal is to show how the migration process evolves when changing the configuration file (the list of available refactoring rules), while keeping the input the same. Here, we look only at the log of said process, not at the COBOL code. We use an already-available tool to produce Finite State Automatons from the logs, then compare the models using a log-based behavioural differencing algorithm. We finally provide Raincode Labs experts with a visualisation tool allowing them to show the process evolution to their client.

The purpose imagined for our tool is to help experts in the early stages of the migration projects, when they have to explain to their clients why a configuration is superior to another or why removing a specific rule might have a broader effect than expected. We mean for our tool to help fulfil their biggest demand: bridge the gap between them and their client by helping them show, in an accessible way, the knowledge they have acquired over the years. As will be demonstrated in chapter 6, other uses were also considered by Raincode experts when they were presented with the tool.

This strengthens our confidence that our work can be used in various contexts and not only within our partner company. Since it compares how a complicated process evolves, it could be used to make sure that maintenance tasks do not change the behaviour of the process or to help optimise it, for example by showing which parts of the process are most often executed and would bring the biggest performance impact. These use cases, along with the idea of helping people get a better understanding of how changes to a configuration can affect a specific process, are applicable in all contexts where a process producing textual logs exists.

5.2 State of the art and background

In this section, we present the base algorithm that we used to compare the FSAs that we generated from the logs of the migration process.

In order to help Raincode engineers show their customers why they should or should not include a certain refactoring rule, we want to offer a tool that supports the analysis of different variants of the refactoring process, in order to gain more insights on what and why changes occur when activating or deactivating a rule. While many tools exist to compare versions of code artifacts, we want to focus on understanding the evolution of the migration itself through its logs.

With that in mind, we based ourselves on Goldstein et al.'s *log-based behavioural differencing* algorithm [GRS17]. This algorithm takes two FSAs models as input, each one having an *INITIAL* and a *TERMINAL* node, and compares the models. We define *paths* in those models as a sequence of linked nodes starting at the *INITIAL* and ending at the *TERMINAL* node. *Common paths* are paths that have exactly the same labels and are present in both models. *Common nodes* are then nodes that have the exact same label and are part of at least one common path. Finally, since we know the *common* nodes, we can derive the *added* and *removed* ones and colour-code the graph. All of the concept described above are formally defined in section IIB of the original paper. Here, we only give the intuition of the differencing algorithm: its goal is to highlight changes in two (consecutive) executions of a process. If p_1 and p_2 are two log files representing a process, the result of the differencing algorithm applied to p_1 and p_2 answers the following questions:

- What step(s), if any, happened in p_1 but not in p_2?
- What step(s), if any, happened in p_2 but not in p_1?
- What step(s) are common to p_1 and p_2?
- How has the execution of p_2 changed with respect to p_1, i.e. are some steps present in p_1 happening more or less often in p_2?

This allows us to provide a clear representation of the changes between two executions and to identify some of those changes as symptoms of issues or bugs. To support the idea of finding problem-inducing changes, the first execution p_1 is assumed to be a known normal, stable or bug-free version; while p_2 has not yet been cleared as correct and needs to be analysed.

As mentioned, such a tool can be helpful in any situation where one wishes to compare two consecutive executions of a process in order to analyse the changes between them. Many such instances can be imagined. For example, in the context of analysing an evolving online website, p_1 could represent the old version of the website, while p_2 would represent the website including its new features. The goal of the differencing algorithm is then to highlight what new paths (if any) exist in the navigation to see if the new features are well-comprehended and used.

The approach proposed by Goldstein et al. presented a way to use both versions (p_1 and p_2) of the log messages and create Finite State Automata representing their respective behaviour. Both models are then compared, allowing to identify outliers or behaviours that are different and therefore considered abnormal.

In our industrial setting, we apply this algorithm to visualise differences in how and when refactoring rules were applied before (p_1) and after (p_2) a change to the rule set, thus giving insights in how or why a change affected the refactored files.

An illustration of the results obtained by Goldstein et al. on the example of the usage of a website is shown in Figure 5.1. 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 of going from the first INITIAL node to the *Credentials* node remains unchanged while the probability of going to the new node *PwdReset* evolves from 0 to 0.33.

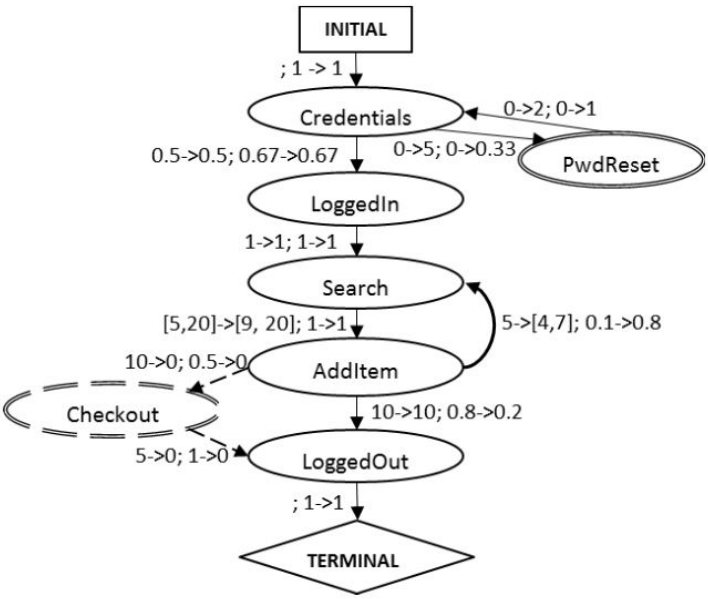


Figure 5.1: Result of log differencing for the shopping process

As we will see in the next section 5.3, translating this example to our spe-

cific context is fairly simple. Nodes will correspond to log lines (representing a migration rule), and the edges and their probabilities will model the iterative process of the migration.

The use of FSAs in this context has two justifications: first, those models were used in the original paper and there are available tools to produce them from the logs. Second, the FSAs states represent every migration rules, and the links between the states represent the order in which those rules were executed. The order of the execution is key to understand how the configuration of the migration might affect its execution: we want to visualise the effect of adding or removing rules and answer questions such as: "*Does this change affect the execution of other rules?*". To answer this question, we will look at both sequences of rules and see if the addition or removal of a rule had side effects, affecting the order of other rules.

5.3 Our approach

5.3.1 Preparing the log files and getting FSA models

The first step of Goldstein et al.'s method is to extract FSA models from the logs. This is done in two phases: normalising the logs and then extracting relevant information from them.

The content of logs can vary immensely from one process to another: they can contain debug messages, errors, timestamps, hardware information, etc. It is important to think carefully about what exactly we want to visualise. Differentiating relevant information from uninteresting information requires expertise and manual effort.

Reconsider the previous example of logs from a website. They can contain debug messages, information about the clients (language preference, country, ...) and navigation details such as the pages and products visited. Each log line has a timestamp. We want to analyse the flow of the website to see what pages or products are visited most by clients, if clients get stuck anywhere because of a bad design, as well as make sure that page loading times are within expected limits.

Reaching this goal is possible by using only the part of the logs that contains the timestamps and the page visits, while dropping the rest of the data to ensure having the simplest possible model. Figure 5.2 shows a truncated example of the normalised log files obtained this way.

In our case, we analysed the data provided by Raincode Labs for the pilot study of a specific migration project. Since Raincode Labs does not keep track of the various iterations that they perform to get the final configuration files, we could not simulate exactly our target use case of seeing how different configurations affect the migration process. However, we can use another

```
1 Credentials Page 13:53:37.281
2 LoggedIn Homepage 13:54:27.841
3 Search Page 13:55:02.182
4 AddItem Page 13:55:57.283
5 ...
6 Checkout Page 14:15:29.724
7 Logout Page 14:16::37.528
```

(a) Log file p_1

```
1 Credentials Page 10:32:02.356
2 PswReset Page 10:33:05.628
3 LoggedIn Homepage 10:35:18.724
4 Search Page 10:35:02.827
5 AddItem Page 10:36:03.924
6 ...
7 Logout Page 10:50:12.656
```

(b) Log file p_2

Figure 5.2: Log files for two executions of the shopping process, example from [GRS17]

source to compare the logs: the redeliveries. We can compare the execution of the migration process on a file in the first delivery, and then again when it gets redelivered. In this case, the configuration stays the same, and the analysis ports on what the changes within the file caused to change in the migration. The migration process we analysed concerned about 3000 artifacts in total, and two redeliveries were performed. The first concerned 47 files, the second only 8.

Since we are interested in modelling and understanding the refactoring process itself, we filter out any errors or warnings from the logs, and from the other log lines, we simply keep the name of the refactoring rule that got triggered. We discard the idea of keeping track of the transition times since an analysis of the time needed for the refactoring is something that is already available internally at Raincode but very rarely used in practice.

The log files generated by the migration process were quite substantial with an average of around 1200 lines before cleaning up all unneeded information (errors and warnings) and around 900 useful log lines after cleanup. Even if we knew from Goldstein et al.'s experience that this amount could be reduced further by translating to graphs, we made a design choice early on to ensure having something small enough that we could analyse.

This choice is that we decided to divide the logs into the main phases: `rea`, `reb`, `rec`, `red` and `ref`. These correspond to naturally independent phases

(preprocessing, clean-up, etc) which have always been analysed separately by Raincode Labs engineers. We decided, after discussing this with our experts, to focus our analysis on the main migration phase, rec, which still contains upwards of 400 lines, plenty of data to provide an interesting analysis.

In the original paper, FSA models are then extracted from the normalised logs using the *kTails* [BF72] algorithm[1]. Many tools applying this algorithm already exist, and Goldstein et al. chose Perfume [Bes+21a] for its relatively concise output. The tool takes a normalised log as input, and calculates the corresponding Finite State Automaton, outputting it to a file in the standard `.dot` format accepted by Graphviz [Ell+22]. The file representing an FSA is simply a list of node IDs and their labels, followed by a list of edges between those nodes, which allows one to easily recreate a graph representing the extracted behaviour. However, since Perfume has become abandoned and impossible to run, we used a substitute called Synoptic [Bes+21b], created by the same team.

Figure 5.3 shows a graphical example of such a FSA for each of the two logs of the previously mentioned website example. The first word on each log line is used as node ID, subsequent lines represent transitions from one node to the next one, and the timestamps are used to calculate the time required to perform a transition. If this time varies, minimum and maximum values are shown using brackets. This execution time can be seen on the edges, before the semicolon. After the semicolon, we see the transition probability, i.e. the probability to go from the start node to the target node the edge points to.

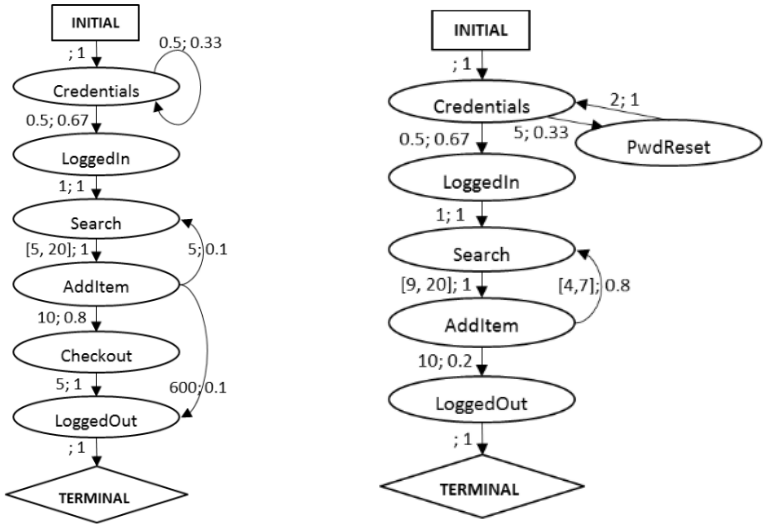
Once both FSA models have been calculated, we can compute the differences between them. The idea is as follows: we want to keep all nodes from both models (marked either as common, added or removed), but not all edges, since otherwise most of them would be duplicated. We do keep all edges from the second model, to highlight how things changed while keeping only some from the first model. All edges are still adorned with the metadata, edited to show the evolution in execution time and transition probabilities.

5.3.2 Applying the base differencing algorithm

To apply the algorithm described in section 5.2 to our FSA models, we reimplemented it using Pharo [Ber+13; Pha21]. The latter choice was mainly motivated by its companion visualisation framework Roassal [Ber16; Ber+21] that we wanted to use afterward.

Concerning visualisation, we stayed relatively close to what was proposed in Goldstein et al.'s original paper (as shown in Figure 5.1), with nodes representing rules linked together by arrows representing the probability of ex-

[1] A detailed discussion of what *kTails* is and why it was chosen, along with pseudo-code describing the differencing algorithm can be found in Goldstein et al.'s paper [GRS17].



(a) FSA for log file p_1 **(b) FSA for log file p_2**

Figure 5.3: FSA graph representation of both logs to compare

ecuting one rule after the next one. We use colour rather than line style to differentiate the status of the nodes, following the usual colour code of red/orange/green for a removed/modified/added rule execution, respectively. To ease readability, we visualise probability changes with line thickness. This removes cluttering text from the view while allowing for seeing the exact value of the probability via mouse hover.

The visualisation resulting from this implementation is shown in Figure 5.4. In order to be able to show all the nodes on a single image, thus giving an overall impression of the total size of the resulting graph, we used a force-based layout[2]. This layout differs from the linear layout used by Goldstein et al. which is arguably more readable as it follows a natural top-to-bottom reading path, but would not scale up to these sizes of graphs as it would not fit on a single page or screen.

5.3.3 Limitations of the visualisation

It becomes clear from looking at Figure 5.4, that our data is of a different nature than what was presented by Goldstein et al. [GRS17]. First, due to the iterative nature of the refactoring process, our graphs are sparsely connected, with very few or very small cycles. They are also considerably larger than the ones depicted by Goldstein et al. With a subset of the 140 available refactor-

[2]The Roassal visualisation framework is sufficiently versatile to support several other layouts, but none was entirely satisfactory due to the graph size.

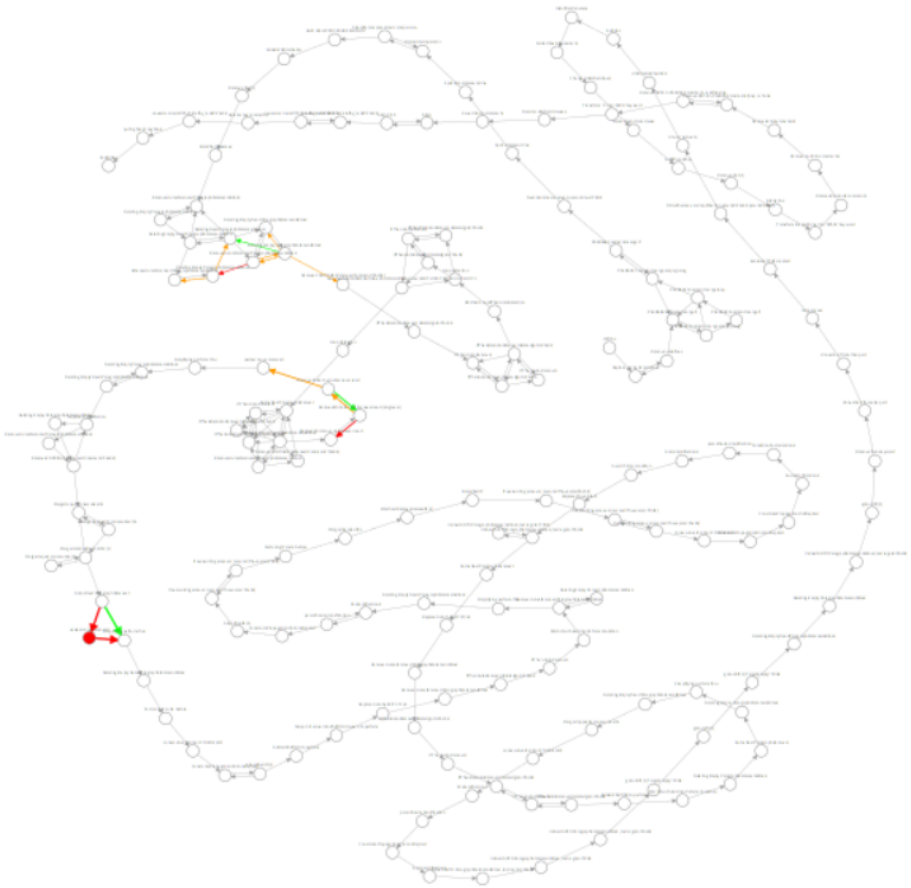


Figure 5.4: First differencing output for one program

ing rules that can be triggered multiple times, even when simplified into an automaton, our average output graph contains around 120 nodes.

While the nature of our graphs was not an obstacle to the execution of the algorithm in terms of processing time, their size made them overwhelming and hard to interpret visually, forcing a user to scroll through a zoomed-in version and thus quickly lose the bigger picture.

When analysing the nature of the graphs produced for our industrial case, to see how they could be improved, we made two important observations. First, our graphs are extremely linear, often displaying long chains of nodes connected by a single edge with a transition probability of 1, or several edges with no changes in the transition probabilities, e.g., as shown in Figure 5.5. Again, this is caused by the iterative step-by-step nature of the refactoring process: all rules were created with this process in mind, refactoring the code

by a small step every time one rule is applied. In some cases, the previous rule prepares the code for the next rule, which means that we can expect to see sets of rules being applied in a specific order. While this is an interesting finding, those fine-grained rules do not provide a high-level understanding of what changed in the overall migration yet they take up most of the space on the graph.

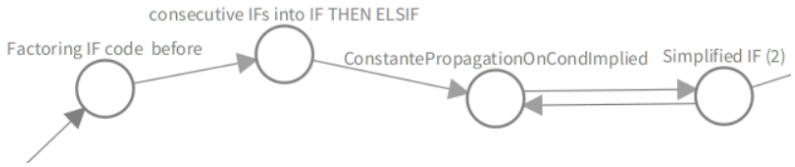


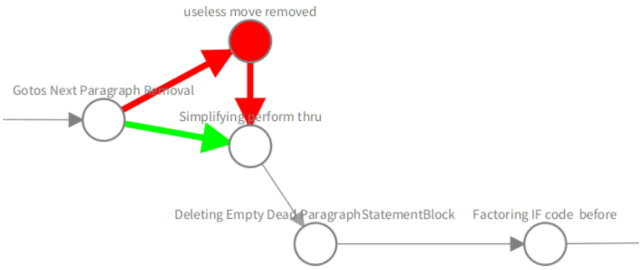
Figure 5.5: A chain of nodes with no changes in the transition probabilities

The second peculiarity we noticed is that changes often tend to happen in clusters. We identified two kinds of such clusters. Either a (few) new rule(s) are triggered or a (few) old one(s) are not appearing anymore, after which the execution trace returns back to its mostly linear nature, as shown as an example in Figure 5.6a. The second kind of change, like the one depicted in Figure 5.6b, is more complex. It presents itself as a cluster of highly connected nodes with mostly transition probability changes, along with a few added or removed nodes and edges, after which the graph again goes back to its linear execution as described previously. In both cases, these clusters are precisely the kinds of structures we would like to be highlighted when looking at our visualisation.

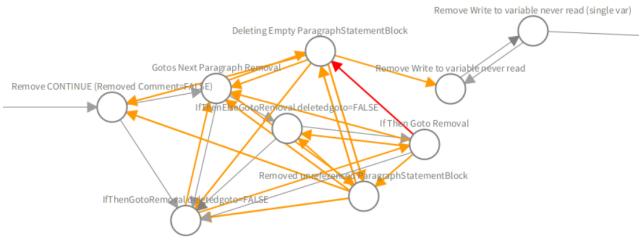
5.3.4 Adapting the visualisation

Based on these observations, we decided to address the readability issue by creating a fusing algorithm to apply after the computation of the original differencing algorithm. The fusing will collapse long 'unimportant' chains into single nodes, causing more interesting clusters of changes (as described above) to become more prominently present. The algorithm is quite simple and is applied iteratively throughout the graph until there are no valid nodes left to fuse. Two linked nodes are fused if they fit the following criteria:

- Both nodes are common (neither added or removed, i.e. not in green or red);
- No transition edge from or to those nodes is added or removed;
- No transition edge to or from those nodes has seen a probability change between the two variants of the process (i.e. in orange).



(a) An old rule being removed, then return to previous execution



(b) Cluster of modifications concerning the removal of GO TOs

Figure 5.6: Two types of clustered changes

Consequently, the algorithm takes the graphs outputted by the log-differencing algorithm as input and returns the same graphs, without every node corresponding to the previously mentioned criteria (common and with no coloured edge in or out). New nodes are added to the graphs to fill in the gaps created by the deleted nodes and annotated to show how many deleted nodes they replace (at minimum, 2).

Since we are using graphs, the fusing algorithm can be defined recursively as follows:

```
1  fuse(graph){
2     do {
3        // Keep track of whether or not we fused
4        beingFused = false
5        // Keep track of visited nodes
6        visitedNodes = new Array()
7        // Start at the top
8        fuseRec(graph.initialNode(), visitedNodes)
9     } while(beingFused)
10 }
11
12 fuseRec(currentNode, visitedNodes){
13    // Fuse anything that we can in the current node
14    foreach(child : currentNode.children()){
15       if(canFuse(currentNode, child)){
16          fuseNodes(currentNode, child)
17          beingFused = true
18       }
19    }
20    //Add the current node to the visited ones
21    visitedNodes.add(currentNode)
22    foreach(child : currentNode.children()){
23       if(child not in visitedNodes){
24          //Recursively go through each unexplored node
25          fuseRec(child, visitedNode)
26       }
27    }
28 }
```

The fuse conditions ensure that no key insight will be hidden from the user when looking at the resulting graph while minimising the amount of information shown at once. For example, all nodes in Figure 5.5 would be collapsed into one single fused node, while only the two last nodes of Figure 5.6a would be fused, leaving the *Simplifying perform thru* node visible because it has two incoming links: one removed and the other added.

Recall the initial example of Figure 5.1: our algorithm would not fuse a single node in that graph, since each node is either connected to an added or removed node or one of its edges has seen a probability change.

We decided to differentiate the fused nodes visually by showing them in grey rather than white as was the case for the normal nodes. As labels, we simply give them a number describing how many nodes were collapsed into them and show that label inside the node rather than under it. Finally, a fused node's size is proportional to the amount of nodes it contains, enabling a more "at-a-glance" analysis.

Our fusing algorithm helped us shrink the graph from an average of 120

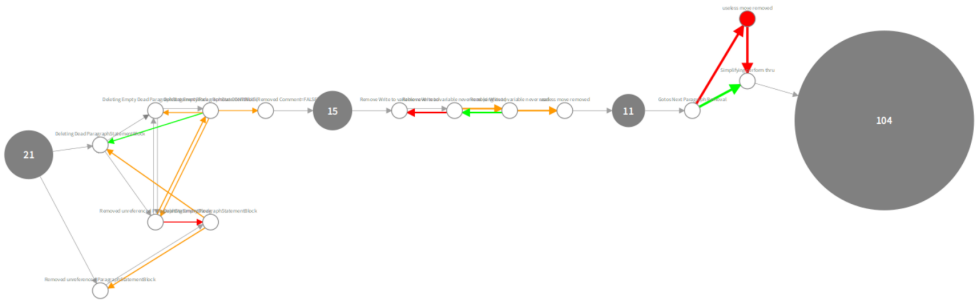


Figure 5.7: Adapted visualisation after fusing nodes, for the same input as Figure 5.4

to 30 nodes (more in-depth metrics are presented in chapter 6). As described, only plain white unchanged nodes were fused, and they still appear if they are linked to a change, allowing a user to look for the context or result of a modification. If needed, a fused node's internal details can still be inspected in a new window when clicking on it. As an example, the result of our fusing algorithm on the graph of Figure 5.4 can be seen in Figure 5.7.

5.4 Generalisability

As mentioned previously, the log-differencing technique presented in this chapter is applicable in any context where there is a process producing textual logs, for example the use of a website. A recent survey [Yan+23] suggests that the issue of analysing and understanding textual logs is very common.

The paper explores the use of logs in a very different context than the one in Raincode Labs: embedded systems. Such systems come with a few challenges that do not exist in the use case we explored, for example the cost of generating the logs being high and the fact that those systems are concurrent. Nevertheless, the authors identify a similar use of logs as what we see within Raincode Labs: the logs are analysed using `grep` or self-made scripts due to a lack of proper tools. The authors of the survey also found that the developers using the logs had the same goals as our Raincode Labs experts: getting context about what happens at a specific point of the project and using the logs to better understand a process that they see as a black box.

While particular attention to the concurrency aspect would be necessary, the technique presented in this chapter should be able to provide key information in the context of embedded system analysis or any other domain where textual logs are available that describe a process in some level of detail,

strengthening our claim that it is general.

5.5 Conclusion

In this chapter, we focused on the comparison of two consecutive executions of a process, to see how the change in input or configuration affected the process itself. We used the log of the said process as input, sanitised those logs and transformed them into FSAs using an external tool. We then compared those models using an existing algorithm and visualised the results.

Finally, we identified some limitations in the original visualisation, causing it to not scale when applied to our industrial scenario. We then proposed solutions to those limitations and implemented them in our visualisation tool.

The work about the extraction of information from the logs and the differencing itself is a direct application of Goldstein et al. [GRS17] with no change to the algorithm. The visualisation as well as the fusing algorithm we created are new contributions, allowing the original technique to scale to industrial-size data. The analysis of logs, in general, has been identified as highly relevant [Yan+23], which means that several other tools exist, some among them being focused on the visualisation of log differences. This is the case of SnortView [KO04], created to detect network intrusions, and ProcessProfiler3D [Wyn+17], a log-differencing visualisation framework that is geared towards analysing how different cohorts of people use a process and how their use could affect their performances.

Validation of the log visualisation tool

6

In this chapter, we first present a few metrics regarding the results produced by our visualisation tool. Second, we analyse the execution time of the model creation and the differencing and fusing algorithms, validating that the original algorithm does apply to our industrial-scale data. Third, we discuss the size of the input logs and graphs and compare the size of the resulting graphs, for both the original and fused graphs. Fourth, we present how we validated our visualisation tool with two Raincode Labs engineers involved in PACBASE migration projects, and analyse the results of this user study.

6.1 Metrics

As mentioned, since Raincode Labs did not have an ongoing migration project, we had little or no data from the rule selection step available. Therefore, the metrics of our algorithm shown in this section were applied to the redelivery step instead.

Goldstein et al. stated that the creation of the `.dot` FSA models is the most time-consuming step of their algorithm. With normalised logs (see subsection 5.3.1) containing hundreds of nodes, Synoptic can take hours to generate the corresponding models on an Intel i7-9750H processor with 16GB of RAM, requiring around 20 hours to compute the models for both versions of 39 migration logs of an example redelivery. Note that this model-generation step would require companies to have a server that can run the script outside of office hours for more efficiency (Raincode Labs does have suitable machines).

Once the models have been created, the time required to obtain the diff graph is minimal: on the same processor, calculating output graphs for our example redelivery of 39 files takes only 0.77 second, with only an additional 0.58 second needed to apply the fusing algorithm, increasing the total execution time to 1.35 second in that case.

Regarding scalability, the model-creation step could become a bottleneck because its execution time and memory consumption grow superlinear with input size. With our biggest log of almost 800 lines, we need 12 out of the 16 GB of memory of the machine we used. Therefore, to scale up even more than

what we have now (which was sufficient for the industrial case at hand), we would need to create a more efficient model-creation tool, instead of reusing the existing solution.

In terms of size, the transformation to FSA models already shrinks the logs quite a bit: we go from an average of 282 lines (see Table 6.1) to only 119 nodes in our input graphs.

	Log lines	Graph nodes
Mean	282	119
Min	72	61
Max	779	198

Table 6.1: Size of inputs (logs and graphs) for all 39 programs

Applying the differencing algorithm to show information from both variants of the process increases the number of nodes by just a bit, from an average of 119 to one of 123. As illustrated by Figure 5.4, this amount of nodes is overwhelming to be visually analysed. The second column of Table 6.2 shows the improvements gained by the fusing algorithm.

	Full graphs	Fused graphs
Mean	123	33
Min	74	1
Max	201	80

Table 6.2: Graph size in nodes

Fusing reduces the average amount of nodes by around 75%, resulting in much more readable graphs. The minimum graph size consisting of a single fused node is also an interesting find. In fact, this file slipped through the first analysis of the engineers: it was refactored even though it had not changed. This explains the single fused node representing no changes.

6.2 Interview with the engineers

Given the size and style of the company, only two people are working on PACBASE migration at Raincode Labs. One of them is more customer-oriented, having as main responsibility to assist in sales and accompany customers in the first step of selecting the refactoring rules. The second person is more technical and the main developer of the entire refactoring process. For our validation, we interviewed both engineers, resulting in varying answers due to their different focus and backgrounds. In the remainder of the text, we will refer to them as participants P_C and P_T respectively, for customer- and

technical-oriented. In what follows we will first present our validation methodology, then discuss the different points on which both engineers agreed, followed by the ones on which they have a diverging opinion.

6.2.1 Interview set-up

Since we had only two engineers available for our study, we conducted a semi-structured interview [AST13; Ada15] with each of them individually (a full interview taking around 45 minutes).[1] Before the interview, they received a one-page manual describing each component in our visualisation tool and its meaning. The engineers could refer to this manual at any time during the meeting. During the interview, we first showed them our tool and let them explore it for a bit (P_C: 6 minutes; P_T: 5 minutes) while answering any questions they had regarding the use of the tool or the inputs it takes. Once they got situated, we asked them to give us their interpretation of the graphs they were shown. We then had a semi-structured discussion with them, driven by a set of guiding questions that we prepared beforehand. We included some open questions to provide them the space to present us with their own suggestions along with their opinions. Our accompanying repository [Dek21] contains the manual that we presented to the engineers, our list of guiding questions to structure the interviews, along with our analysis of the reactions and answers of the engineers. The user manual is also included in Appendix A and the question summary is in Appendix B.

6.2.2 Methodology

To structure the presentation of our results, we took inspiration from the work of Sillito et al.[SMD08]. In addition to examining the participants' remarks and answers to our questions, we also analysed the questions they were asking themselves while thinking out loud. We wanted to study not only their opinions on the tool, but also how well our tool supports the actions they had to perform for the tasks we gave them. We classified these questions into four categories:

- Questions about the **domain** (input, output, inner representation of data in the tool);
- Questions about **tool usage** (often the starting point of a series of concrete actions they undertook to answer that question, without requiring further help from us);

[1] Due to restrictions imposed by the COVID-19 pandemic, we necessarily had to conduct the interviews virtually using screen sharing.

- Questions about the **meaning** of a visual component (they either asked us directly or used the one-page manual to look for the answer);
- Questions leading to ideas for **future work** (when they asked if some feature would be possible to implement).

We adopted this question-oriented format for three reasons. First, we wanted to get as much feedback as possible from our two users, hence the open questions. Second, we wanted to highlight the strong and weak points of our tool: what part of the interface is so intuitive that it does not raise any questions? What part is less easy to understand (gives rise to more questions) and would thus require improvement of the tool or better training? Finally, to validate if our fusing algorithm is an improvement over the non-fused graphs we asked them direct questions about this aspect.

6.2.3 Results of the study

We refer to the file `InterviewCommonQuestions.csv` in our accompanying repository [Dek21] and Appendix B for all questions highlighted in the following descriptions. We limit ourselves to presenting the most significant items here.

Domain

There were only a few questions in this category, yet they were quite specific and common to both participants. Both of them asked: *"What does this tool take as input?"* and *"Can a node be repeated in the graph?"*

Tool usage

Since we asked the engineers to think aloud while exploring the tool, questions regarding tool usage were the most common. This category of questions showed us that the most performed actions were moving nodes and zooming into the graphs. Every time any of our engineers opened a new graph, their first reaction was to zoom in, then sometimes move nodes. Those actions were translated by questions like *"How can I see/arrange this better?"*

Meaning

The second-most frequent category of questions was about the meaning of specific visual components. Two questions were common to both participants: *"Can you clarify the meaning of the orange/gray nodes?"*

Future work

Both participants provided concrete ideas about where in the migration process they would apply our tool, gave a few suggestions about our layouts, and presented at least one proposal for new features.

Open questions

For the open questions we asked our participants, we were mostly interested in knowing whether they perceived our fused graphs as more useful than the

full graphs. For this, the participants were requested to open a non-fused version of a graph of which they had previously analysed the fused version. We then asked them questions regarding the similarities and differences of the information shown in this graph with respect to the previous one.

6.2.4 Analysis of the results

Domain

The first thing that we observed is the fact that the engineers did not pose many domain questions. This is not surprising since they have been working on PACBASE migration projects for over 15 years and are therefore experts in this domain.

This is confirmed by the fact that they both asked the same detailed domain question very early during the interview. Indeed, both participants are used to working with and looking at the logs and wanted to know exactly what parts of it we selected before proceeding with the analysis we asked them to perform. Once they had this information, they had no trouble recognising the things they are so used to work with.

Still concerning the domain, both engineers asked us if it was possible to have a node repeated in a graph. Both of them seemed to have trouble visualising and understanding that. They felt it might be confusing for an end user. P_T said that *"it would be less useful that way"*. Having no concrete example of a graph with a repeated node at hand during the interview, it was impossible for us to show them and get further feedback on this issue. We keep this in mind for future work: while we feel that a repeated node should not threaten the usability of our graphs, we should test this further. It may be necessary to adapt the visualisation so that repeated nodes occur together, and analyse whether that yields a better visualisation or not.

Tool usage

We observed that the most performed actions were moving nodes and zooming into the graphs. This may suggest that despite our fuse algorithm, the generated graphs are still not easy to analyse to understand "at a glance", requiring engineers to zoom in on a specific section at a time, then move over to the next one. Their systematic moving around of the nodes may also suggest that the default layout we chose might not be ideal for this specific use case.

However, when asked about this, the engineers assured us that, while the overall placement of the nodes was certainly not perfect, the tool itself remained usable. Moreover, some of the layout issues were due to the length of the labels of the nodes, and are inherent to Raincode Labs' use case. For example, a particularly lengthy and fairly common label is `IfThenElseGotoRemoval deletedgoto=FALSE`. This label-length issue could easily be fine-tuned by changing the logs themselves and is something that the engineers would be

willing to do when using our tool on a regular basis.

Meaning

Questions regarding the meaning of visual components arose mostly at the beginning of the interview when the engineers were still unfamiliar with the visualisation or unsure about how to do something. Yet they were able to answer most, though not all, of those questions by themselves by referring to the one-page manual we provided. This means that the most unclear components should probably be explained better when presenting the tool to new users. We found that the meaning of the orange links is the least easy to understand. This is probably because its description in the manual remains too high-level, referencing varying probabilities, causing both engineers to need clarification of this description. Second, the use of the grey nodes might not be very intuitive either: one engineer did not click on it to open the inside representation, and the second one needed us to tell him how to do so, even though it is mentioned in the manual.

Future work

As described previously, the selection of what to keep from the logs is critical to having useful outputs. In this case, we chose to keep only the rules names, which seemed sufficient to the engineers. We also choose to focus on the most fine-grained level of the logs, while we could summarise some of the information and present the visualisation on another level of detail. This was noted as something that could interest P_C. Another round of interviews on a more coarse-grained level would have been useful to compare which level would be more suited for which analysis, or simply preferred by the participants and is an avenue for future work.

As for feature requests, we want to highlight two that seem quite interesting. P_C would like to have, attached to a fused node, details on where in the process changes start to happen. For example, he would want a label on the right-hand side of the node, noting that he can look at log line number 42. P_T had a simpler request: he would like to highlight a node that is new to a file. For example, if rule A was not triggered in the first execution of the migration but is in the next, he would like it to be not in green for "added", but in yellow for “added and new to this file".

Open questions

We focus here on the comparison between the non-fused and fused graphs. When shown a non-fused version of a fused graph they had analysed before, we asked the engineers if, at first glance, this was something they had seen previously. Both of them said they thought they recognised it, though without certainty. We then confirmed that it was indeed the same thing, visualised differently, and asked them if they could find the same information as was presented in the previous graph they analysed before. They agreed that they could but that it would take them longer. P_C stated *“Yes, I could find the*

same information, but I don't really care about all those white nodes, they don't give me any interesting information". P_T disagreed that the white nodes were uninteresting. He liked the idea of being able to look at a broader context if needed, but did agree that this did not necessarily require the full graph, instead having the option of opening the inside of a fused node would be sufficient for his needs. When asked if they thought that the used graphs were an improvement over the non-fused ones, they both agreed that the fused graphs were a strict improvement. They also said that no features from unfused graphs would be missed.

6.2.5 Discussions

To conclude, we observed that, even if they had fairly different points of view due to their backgrounds, both engineers used the tool in a very similar fashion (excluding the idiosyncrasies of who prefers scrolling and who prefers dragging). When presented with our visualisation, they looked at the graph from left to right, stopping on clusters of changes and guessing at what might have caused that change. It is in their interpretation of the changes that they differ. P_C views everything in terms of "what did the customer do to initiate this change?", while P_T goes directly to reflect on the interaction between the refactoring rules themselves.

Because of those different inclinations, they also have a different idea of how they would use our tool. P_C, due to his very frequent interactions with the customer, wishes to use it as an aid in a conversation to convince his customer or to help him understand why a rule should be picked over another. He does not, however, see how the tool could be used in the context of working on the refactoring process itself. On the other hand, P_T said that our visualisation might help him when he is maintaining the rules and the process: he would like to see if the changes he made to a refactoring rule influence the execution of others, to ensure nothing gets obsolete.

Later in the thesis, another use for our tool was proposed by another Raincode Labs employee who was not part of the validation described here. He wished to optimise the migration process, which currently tries to apply every rule in order from top to bottom until one that is applicable is found, then goes back to the very first rule and tries again in the same order. While this is an easy and reliable implementation, it does not take advantage from the fact that some rules are preconditions to others and that when a rule is triggered, there is a small set of other "candidate next rules" that are very likely to be next. This causes significant time loss, which could be fixed with a smarter algorithm to pick the next rule to try. This employee heard of our tool and came to us to see if we could collaborate and help him with his task. In this case, the visualisation would be less necessary, but the log-based behavioural

differencing algorithm could easily be modified to output the rules occurring most often together, providing him with a valuable basis to implement the new rule-picking strategy.

This suggests that our tool is quite versatile: depending on the inclination of the person who is looking at it, it can prove useful in several different ways. We validated this claim further by presenting this work at conferences, where we got interest from researchers in other domains, who came up with use cases for this tool completely outside of the context of Raincode Labs. In particular, the use case of comparing dependencies in Unix installation was proposed: comparing logs from the booting process of these OS could help show if it would be possible to migrate from one distribution to another without losing compatibility with certain tools.

Finally, we want to note that our fusing algorithm seems satisfactory: neither of our participants missed any information from the previous graphs, and both agreed that they would rather work only with the fused graphs.

6.3 Conclusion

In this chapter, we presented our methodology and observations in regards to the validation of the tool presented in chapter 5. We showed that it scales both in terms of time of execution and with the new version of the visualisation. We validated our created tool with Raincode Labs engineers, who gave positive feedback as well as came up with other use cases for our tool, proving that the work shown here can be applied to many different scenarios, both inside our partner company and in other industrial contexts.

Part III

Analysis of the effects of the refactoring process

In this next part of the thesis, we aim to approach the issue of understanding the migration process from a different perspective. While previous analyses have focused on explaining exactly how the migration rules interact with one another through the logs of the process, this study intends to delve into the specific details of what precisely occurred during the migration.

For this part of the work, the COBOL code itself serves as the input for our analysis. Specifically, we compare the version generated by PACBASE with the refactored version resulting from the migration process. However, due to the substantial differences between the two versions, a standard textual diff proves inadequate for comparison. To give concrete numbers, a file pre-migration is on average 2680 LOC, while it gets down to 1719 LOC after the migration. A standard *diff* outputs on average 4402 LOC, which means that almost all lines of the file are modified. Furthermore, the automated refactoring is not applied to a single file but to many (3014 different files in our use case data), rendering manual analysis of all the files impossible. As a result, our primary effort is directed towards extracting the most relevant parts of the code to reduce the amount of data to compare and make it more manageable as well as allow the comparison to be partially automated and provide a summarised version of the information.

Once the important elements have been extracted, they then have to be arranged into a model. In this part, we will have two separate models: first, Control-Flow Graphs that will be used in a visualisation. We chose them for their user-friendliness, because they are familiar to developers and therefore easy to read and understand. However, CFGs are not suitable for our comparison, which is why we will present a second model, Labelled Transition Systems. LTSs are simpler in nature than CFGs and therefore compatible with comparison algorithms, but much less destined to be analysed by humans. While the use of both models adds a bit of overhead from translating one to the other, it allows us to both have automatic analysis and present human-readable data.

The final element of this part of the thesis is the trace equivalence algorithm that allows us to match the various paths present in the LTS models before and after refactoring, it allows to further sort the data to get to the most important parts: all of the structures that can be proven to be trace equivalent do not need to be manually analysed, leaving experts with the task of only looking at the edge cases where the automatic analysis could not prove equivalence.

Semi-parsing

7

7.1 Context

As mentioned, the crucial aspect of this research is to provide evidence that the migration process maintains the program's behaviour, ensuring that the two versions remain semantically equivalent after refactoring. To achieve this, certain assumptions will be made, and implementation choices will be discussed and justified based on expert consultations.

In light of the migration's nature and after a long discussion with Raincode Labs experts (detailed in subsection 7.3.1), we have identified the structural and conditional aspects of the code as the most significant elements to consider. Consequently, we opt to abstract all the statements away, focusing solely on the more crucial components. Once we have successfully extracted the pertinent parts of the code, we proceed to transform them into a model representation using Control-Flow Graphs (CFGs).

In this chapter, we delve into the first step of our analysis, where we extract the relevant information using semi-parsing and generate the CFG models, while chapter 8 validates our approach. The subsequent chapter will focus on the comparison of those models on an algorithmical level and the next one will provide validation of the comparison. Finally, chapter 11 presents the visual tool that we created, and chapter 12 presents the validation that we performed with the help of our experts.

We will also argue that the ideas and methodologies presented in this chapter can be generalised beyond the specific context of our partner company. Demonstrating that a program retains its behaviour after undergoing refactoring is pertinent in numerous scenarios where code is continuously developed and maintained over extended periods by various individuals. This is a prevalent situation in both industrial and academic settings. While caution must be exercised in analysing each unique situation and the associated assumptions, the ultimate objective remains universally applicable.

Special attention will be given to explaining the rationale behind the decisions made throughout this project. The clarity of these explanations is of utmost importance to facilitate the understanding of this work by other researchers, should they wish to conduct similar investigations. We will be transparent by explaining exactly why this is applicable to our use case, so

others can determine if it would also be useful for their situation.

7.2 State of the art and background

In this section, we present the background knowledge necessary to comprehend this chapter as well as some related work. We start by presenting the general concepts of parsing and semi-parsing and explain which technique exactly we selected as well as why it is a good fit. Secondly, we talk about the model we chose as output of our semi-parser, Control-Flow Graphs. Our choice was motivated by the user-friendliness of said model since it will later be used as a base for our visualisation.

7.2.1 Parsing

Parsing as the analysis of the syntax of computer programs has been an integral part of automated compilers since their beginning. In a broader sense, it can be understood as the process of eliciting the structure that was expected to be found in a software artifact and making this structure explicit as preparation for further processing [ZB14]. The syntactic commitments of an artifact can be as strict or as loose as the situation dictates, and the expectations are usually expressed in some explicit form: a grammar, a regular expression, a type, a schema, etc., sometimes collectively being referred to as "grammars in a broad sense" [KLV05]. Grammars can have vocabularies ranging from several distinct symbols to over 5000 of them [Zay15]. When the syntactic expectations are expressed in a form close to that of a context-free grammar, many parsing algorithms are readily available [GJ08]. Most of them expect the program either to comply to these expectations fully, or be declared invalid.

Parsers are metaprograms that typically take a program in textual form as input and produce a more structured representation of the program in the form of a tree, a graph, a table, and so on. There is a spectrum of such metaprograms, but we can roughly separate them in two groups: full parsers and semi-parsers.

Full parsers [GJ08] have a complete overview of the syntactic expectations of a language, and expect the input to fully conform to all of them. In exchange, they produce a fully structured ("parsed") version of the input. It is common for abstract syntax trees to contain information about variable types, name scopes and other complex matters useful for figuring out the semantics of the parsed program afterwards.

Semi-parsers [Dea+03; Zay14] admit to having only a partial overview and embrace it by handling uncertain portions of the input tolerantly or not at all, and yield only an estimation of the full structured representation. In the

industrial practice of language processing, one often comes across situations where a full definition of a legacy language is unobtainable or unreliable, or when the investment in the development of a full parser cannot be justified. Semi-parsing offers an interesting alternative. Depending on particularities, the output from semi-parsers can be a normal full tree/graph constructed under a set of assumptions (e.g., "a semi-colon is expected, so let us assume it" or "a variable is undefined, but it seems like its type could be string"), or a partial tree with explicit gaps (e.g., one does not need to fully understand HTML to be able to extract all CSS snippets from it). Fact extractors [LHM03] can be seen as extreme forms of semi-parsers that only collect imported module names to construct a dependence graph, or only handle comments and line continuations to count the logical lines of code.

As wonderful and omnipresent as full parsers can be, they have at least three disadvantages in the context of software modernization. Firstly, they are very complex to write, and literally take years to complete [Mas98]. Many legacy codebases contain a language cocktail, and it would be a major investment to develop a full high-quality parser each time a new fourth-generation language (4GL) enters the scene. Also, handling hard-to-parse exotic 3GLs like REXX or RPG can in many contexts be inexpedient. It has been estimated by one of the experts on the Y2K problem that in 1998 there were at least 200 proprietary 4GLs in active use [Jon98], and many of them are still active today. Secondly, in a world where just handling COBOL (the most used legacy language, and one of the 700 that can be found on mainframes) means being ready to encounter up to 300 different COBOL dialects [LV01] and to deal with various versions of competing compilers that accept diverging syntax, strict adherence to the syntactic expectations is not only undesirable, it is highly unrealistic. There are always "uninteresting" syntactic deviations in spelling keywords or using the preprocessor to handle non-existing statements as extensions. On top of that, modernization engineers often need to deal with obfuscated code (sometimes up to the point of it not being executable) or partial code (e.g., COPY books in COBOL serve the same function as libraries in other languages, but with lexical parameterization — such a COPY book is never a fully parsable compilation unit). Thirdly, language embedding [Ren10] is still an open problem, especially in technology-defined grammar classes. Legacy languages like COBOL, PL/I, FORTRAN or 4GLs, often contain fragments of embedded SQL queries, JCL requests or CICS commands, and we indeed see that appear in our industrial use case. Each such language boundary must be manually programmed and thoroughly tested, and even then without a full parser of the embedded language, the EXEC block containing the embedded queries will stay unparsed. Interestingly, this is a well-known phenomenon in semi-parsing, which refers to such unparsable regions as "lakes" [Zay14].

Semi-parsers have a number of disadvantages as well, such as overapprox-

imating the input language — we will address them concretely in the following sections. The industrial state-of-the-art attitude is to accept semi-parsers in program analysis, but not in program transformation [Bla16]. However, they have three advantages that are worth mentioning up front: (1) since their development effort is much lower, it is possible to create them in an agile way, catering fully to customers' needs and using them in early stages of the project during joint workshops and feasibility studies; (2) due to their built-in tolerance, semi-parsers are often naturally applicable to a range of language dialects, and have the capacity to skip over embedded fragments just as easily as they skip over other "uninteresting" input parts; (3) it is often possible to overstep the formal limitations of language composition and grammar modularity, and develop a bespoke "grammar" (in a broad sense) for each scenario. Semi-parsers are uniformly accepted also in interaction situations where responsiveness is more important than precision — such as syntax highlighting or contextual code completion in an IDE.

The core of the particular algorithm we use in this part of the thesis is based on the idea of *fuzzy parsing* [KOP97] with anchor tokens. In short, the semi-parser scans the input in search of one of the anchors it knows (e.g., `EXEC`, `IF`, etc.) and skips all other tokens. Once the anchor matches, the actual parsing begins. Formally speaking, this is a form of event-based parsing [Zay19] where parse actions are executed reactively after witnessing something specific in the input, and not dictated by the grammar. For our approach, which will be described in full in section 7.3, the "actual parsing" can mean one of two things, depending on the anchor: it could lead to an invocation of an external full parser (for embedding SQL and incorporating full parse trees of each query into the result); or to a modification to a control-flow graph that we are constructing instead of the parse tree. Thus, it is a bespoke setup combining semi-parsing [Zay14] ideas of fuzzy parsing [KOP97] with parsing in a broad sense [ZB14]. We decided to pick fuzzy parsing for its high flexibility: since every anchor token can be handled separately, it is easy to implement a prototype and then add to it as needs arise. It also works the other way around: if we wanted to remove an anchor, it would be as simple as deleting it from the list in the configuration file.

7.2.2 Control-Flow Graphs

Control-Flow Graphs (CFGs) are a representation of all the paths that could be followed during the execution of a program [All70]. Each node of a CFG represents a code statement, and edges connecting nodes represent steps in the program's control-flow. Such graphs can be useful when one needs to understand a program at a glance, or when the language that was used to create the program is not well known to the user. In the context of our industrial

use case, the focus is less on understanding the entirety of a legacy program, but more on allowing the user to zoom in on and compare the control-flow around interest points. This can be achieved more easily using a graph than through highlighted code since when generating that graph, we can pick and choose what to show or not, i.e. keeping only the most important parts of the control-flow to shorten the output. This can be seen as a kind of intentional slicing [Wei84].

The most generic way of creating a CFG is by using a full parser: given the parse tree of a piece of code, generating its corresponding control-flow graph is a relatively straightforward traversal of the tree, creating nodes and connecting them as needed. Another way to generate CFGs is to use one of the many existing tools. The issue with that approach is that those tools are only available for some languages, mostly widely used ones.

Popular CFG-handling tools include GrammaTech's *CodeSonar* (previously known as CodeSurfer) [Gra20] and *Ghidra* (and the entire GTIRB-based ecosystem) [Nea19], UCLA's *Aurora* [UCL03] and UniLeiden's *JShowFlow* [Mui09], Eclipse plugins *EclipseCFG* [Ali09] and *Atlas SDK* [EnS22], Python modules *StatiCFG* [Coe+18] and *PyCFG* [Gop17], etc.

Using a ready-made tool presents the advantage of being a very fast way to obtain a CFG, but may limit the ability to tune the tool's output to one's particular needs. Also, when dealing with less widely known languages, and legacy languages with multiple dialects, it is possible that no quality tools exist that are freely available. This is the case for COBOL. For some more confidential languages, it even happens that no parser is available at all.

Yet another way to generate CFGs is to use a language workbench with a meta-programming language like *Spoofax* [KV10] or *Rascal* [KSV09]. They come equipped with their own way of analysing the control-flow of code, for these examples called *FlowSpec* [SV17] and *DCFlow* [Hil14]. While these are generally of higher quality and abstraction level, all language workbenches require a good deal of background knowledge and often a steep learning curve from their users, which in the context of this project was not sufficiently available within Raincode Labs. While the researcher could have learned one of the described technologies, our ultimate goal is to have a tool that can be used internally in the partner company so we discarded this option.

While each of the above techniques is interesting, we chose an approach that did not require any meta-programming or DSL, and chose to go with a semi-parsing technique since it is both lightweight and accessible, while providing the advantage of allowing to generate CFGs of the desired structure. We discuss further the advantages and drawbacks of our CFGs in section 8.1.

7.3 Creation of a configurable fuzzy parser

In this section, we describe in detail the approach we used to create our fuzzy parser for extracting CFGs from our COBOL files. To allow it to be flexible, we divided it into two parts: the parsing logic and a configuration file, with the idea being that in most cases users of the parser should only edit the configuration file to be able to adapt the parser to their needs.

The semi-parser is the first step in our endeavour to show that two pieces of COBOL code are semantically equivalent before and after refactoring. The first step is to identify, through discussion with the experts and analysis of the rules, the code constructs that we want to compare. Next, the parser needs to be created to extract them from the code. The secondary purpose of our parser is to arrange those structures into CFGs, the model we have chosen to help visualise the structures we extracted. The parser takes as input a piece of COBOL as well as a configuration file and outputs a CFG containing all the constructs specified in the configuration that were found in the inputted code.

In this section, we first present and justify our choice of constructions to extract, then the parser configuration, followed by a preprocessing phase, the intuition of how the parsing algorithm works and finally we explore some results we obtained by applying it to a few small handcrafted examples.

7.3.1 Choice of constructs to extract from the code

Since the code it outputs was never really meant to be manually altered, the PACBASE generator has no consideration to make pretty or even readable code. Everything is dumped into paragraphs and the various pieces of code are not written in any particular order, causing `GO TO` to be not only numerous, but to jump all over the file.

Raincode experts explained during a brainstorming session that, due to this generated nature of the input, the migration rules were created to mostly *restructure* the code: removing useless empty paragraphs and unnecessary jumps, grouping things together, going towards more user-friendly `PERFORM VARYING` loops and restructuring heavily-nested `IF` structures. The process does alter some statements, but far less than it does alter structure. Furthermore, when statements are altered, the changes are small, localised and fairly easy to explain. One rule, for example, groups variable declarations into a single line if possible, so instead of using 5 lines to declare 5 different variables containing integers, the code uses only one and declares them all at the same time.

After our discussion with experts, we were given an exhaustive list of all 140 migration rules and analysed them. We came to the same conclusion as

the experts, finding only 17 out of the 140 rules that impact statements directly. Moreso, most of them handle the removal of dead code or perform renamings. Consequently, we opt to abstract all the statements away, focusing solely on the more crucial components of GO TO, loops and IF structures.

7.3.2 Parser configuration

The configuration is a Python file that contains a set of arrays, each defining a type of language structure we are looking for in the code. Each element of those arrays is a specific definition of that structure, e.g. an IF statement is a specific definition of a condition as illustrated by Listing 7.1. As a reminder, the syntax for an if statement in COBOL is `IF condition (THEN) body (ELSE body) END-IF`.

Listing 7.1: Extract of our parser configuration file for COBOL.

```
conditions = [
   {"start": r"\sIF(\s)+",
    "condition_delimiter": r"\sTHEN(\s)+",
    "mandatory_delimiter": False,
    "single_branch": r"ELSE\s", "end": r"\sEND-IF"},
   ...
   ]
```

Using this file, users of the semi-parser can both pick exactly what structures they are interested in seeing in the output control-flow graphs, and define what is the syntax of those structures. The content of this configuration file would therefore change mostly in function of the language that is being parsed, but one could also omit a certain type of structure if they do not want to have them in the output. Whereas such omissions can result in CFGs that do not fully reflect the language semantics yet, it does help in building the fuzzy parser incrementally, by adding one language construct at a time, until reaching the full language semantics that one wants to cover.

After our discussion with experts and analysis of the migration rules, we chose to extract the following structures from our industrial COBOL code:

Condition The COBOL statements IF/ELSE and EVALUATE ... WHEN.

Loop The COBOL statements GO TO, PERFORM, PERFORM THRU, PERFORM VARYING and NEXT SENTENCE. As explained in section 2.2, most of these are not conventional loops, but behave more like jumps through the code that either return to where they started (PERFORMs) or do not return at all (GO TO and NEXT SENTENCE). The only exception is PERFORM VARYING, which is equivalent to a standard Java for-loop. It is present only in the post-migration files, but its inclusion was requested by Raincode Labs engineers.

Parsable This structure is used to define program fragments to be analysed by an external parser, in our case EXEC SQL, since we want to delegate the parsing of embedded SQL fragments to a separate parser.

Ignorable structures that we do not want to see in the generated CFGs, but that are sometimes needed to disambiguate the rest. In our case, strings and comments. It is important to specify those two because it is possible (and even common) to find a comment or string containing one of the structures we are interested in, e.g. a string containing the word "IF". If we did not specify this to our parser, we risk falsely identifying pieces of comment or string as actual code structures and result in false CFGs.

Special This structure groups special language-specific cases that need to be dealt with. In the case of COBOL, we need to handle the special period statement (`.`) which closes all open statements. In the context of our industrial case itself, we decided to parse only the procedure division of COBOL programs and therefore added a special anchor to skip to that division, which helps us save execution time.

Every element of the arrays found in the configuration file can be edited or removed, and new ones can be added. In case of adding a new element, some edits to the actual parsing code may be necessary to implement the new structure's logic.

A concrete example of such a configuration file for COBOL can be found on GitHub [Dek22]. To define the syntax of the statements we are looking for, we decided to use Python's regular expression module `re` [Pyt22]. This choice was motivated both by its ease of use and by COBOL's inner complexity. Indeed, some statements can have multiple correct syntaxes, like the well-known GO TO which can also be written as just GO while having exactly the same semantics.

7.3.3 Preprocessing

Before being able to parse our COBOL files, we need to preprocess them. Not in the sense of using the COBOL preprocessor, but rather altering the input of our parser. Since the files used in our use case are generated by PACBASE, they contain line numbers and program identifiers, which are a part of the line-by-line structure, but not a part of the core syntax. An example of what a PACBASE-generated file looks like prior to such cleaning has been shown in section 2.2. Technically columns 1–7 and 73–80 can be ignored by the fuzzy parser itself reacting to pseudotokens (similar to how most Python parsers work with indent/dedent pseudotokens), but that would bias our parser too strongly towards position-based languages which often

require drastically different parsing techniques anyway [Zay17]. Thus, we opt for a gentle preprocessor that replaces everything outside Areas A and B with spaces and keeps the COBOL code syntactically correct and semantically equivalent. This also allows us to deal with line continuations and comment markers at column 7.

7.3.4 Parsing the SQL

Since we are relying on semi-parsing (see subsection 7.2.1), the core of our algorithm is relatively simple. First, when our parser is instantiated, it goes through the configuration file to define the anchors. To simplify the parser's code, the different regexes are *compiled* and passed to another object that will keep track of which ones are active (i.e. still present in the rest of the input). The parser only handles the actual matches of the regex, keeping both a list of the next match for each anchor and one with iterators that allow us to retrieve the next matches when needed.

The parser moves through the input considering it as a long string with an index, starting at zero and jumping from anchor to anchor. At each step, our parser finds the anchor that has the earliest match in the input and passes it to a handler that processes it differently according to its type (Condition, Loop, Parsable, Ignorable or Special, see subsection 7.3.2) and according to its exact match. This process creates Node objects that will form the final CFG, and analyses any needed additional information like an `IF`'s condition.

It is important to note that our Ignorable (comments and strings) matches are used only when they are needed to disambiguate. When looking for the next anchor, if we find a match for an `IF`, but find that an Ignorable anchor overlaps it, we pick the string or comment instead of the if and do not add any node to the graph.

The only other special case worth mentioning regarding the computation of additional details about a node is the case of Parsable nodes. To parse the embedded SQL statements it encounters, our parser simply calls the SQL parser generated from an open-source ANTLR grammar [KK15].

Our Parsable node objects have access to the SQL grammar and a function allowing them to parse a string using the `antlr4` Python module. After retrieving the SQL code, conveniently contained between the EXEC SQL and END-EXEC statements, they pass it to ANTLR to create a parse tree that is stored inside the Node object for later use.

The parser's iterative processing of the input, anchor by anchor, continues until either the end of the file is reached or no more matches are found for any of the anchors.

To finish fully constructing the final CFG, we make a pass through our loop nodes to make sure that they point towards the right children. Indeed,

given the jumpy nature of COBOL code, it is common to have GO TO or PERFORM statements pointing towards a label that is further down the code, and that we did not have knowledge about yet at the moment of parsing the loop itself. For simplicity, we resolve all of these after the first parsing phase, when we are guaranteed to have found all the correct labels to point to.

At the end, we obtain an object representation of a CFG corresponding to the given file. It can be serialised into a file to be visualised later, although we do not use those files in our final tool. For this, we use the GraphViz [Ell+22] library, which has the advantage of being directly connected to Python and easy to use to produce a visualisation of such graphs.

7.3.5 Example

We now illustrate concretely how the parser works on the small handcrafted COBOL example of Listing 7.2.

Listing 7.2: A small handcrafted COBOL code example

```
1       IDENTIFICATION DIVISION.
2       PROGRAM-ID. Example1.
3       ENVIRONMENT DIVISION.
4       DATA DIVISION.
5       PROCEDURE DIVISION.
6           IF A > 0
7             NEXT SENTENCE
8           ELSE
9             DISPLAY "First if branch"
10          END-IF.
11          IF B = 0
12            DISPLAY "Second if"
13          END-IF.
```

After loading all anchors and searching for all their first occurrences, the fuzzy parser will be left with only the following: the *special* PROCEDURE DIVISION anchor on line 5 to start the parsing, the *condition* anchors IF, ELSE and END-IF, the *loop* NEXT SENTENCE anchor and finally the *ignorable* string anchor.

Let us analyse the parser operations line by line:

- Even though we start at line 1, there is nothing of interest to parse until line 5, where we find the first PROCEDURE DIVISION anchor. This does not produce any node, but advances the parser's index up to the start of line 6.
- On line 6, we find a match for one of the *condition* anchors. A node denoting the start of an IF is created, and the parser also takes note of its condition A > 0. To allow us to keep track of how the nodes are

embedded, we also increment a *depth* variable. All nodes created after this point will be inside the `IF`, at a depth of 1.

- On line 7, a match is found for a *loop* anchor. A node is created, and a new anchor is added to the list: the parser now looks for a period (`.`), which will indicate the end point of the loop.
- On line 8, there is again a match for one of the *condition* anchors, this time for a branch. No new node is created in the graph, we simply mark the "True" branch of the first condition node as closed (new nodes that we parse next will no longer be added as its children).
- Line 9 does contain a string, but it does not overlap with another anchor, so it is ignored by the parser.
- On line 10, there are two separate entities to parse: first, the `END-IF` which is a match for one of the *condition* anchors and will result in the closing of the first condition node. The internal *depth* variable is decreased by one and this closure is matched with the nearest open condition of the same depth. The second match concerns the period, which is processed to act as the end point of the loop node, adding a temporary node in the graph to mark its position.
- Finally, another `IF` node with the condition `B = 0` is created for line 11, line 12 is ignored and line 13 closes the second condition node.

As described above, after this first parsing phase is done, we make sure all loop nodes present in the graph point to the correct place. In our example, we suppress the previously created period node, pointing the `NEXT SENTENCE` to where that node was, and the graph is complete.

The CFG produced by parsing the program of Listing 7.2 is shown on the left side of Figure 7.1. We represent the start and end points with a diamond shape, and show the COBOL statements in ellipses, with some edge annotations.

Note that due to the syntax of COBOL, very slight changes to a COBOL program can change its CFG greatly. For example, if we transfer the period seen on line 10 to the end of line 13 (i.e. after the second `END-IF` and not the first), we change where the `NEXT SENTENCE` jumps to, and therefore the CFG itself. The CFG output corresponding to this variation of code Listing 7.2 is shown on the right side of Figure 7.1. The parsing itself is almost the same, except for the step at which the period is parsed, which is now the last one.

The last feature of our fuzzy parser is its ability to make use of an external parser to analyse an embedded language, in our case the SQL code present in between an `EXEC SQL` and `END-EXEC` statement in a COBOL program. Consider the code of Listing 7.3.

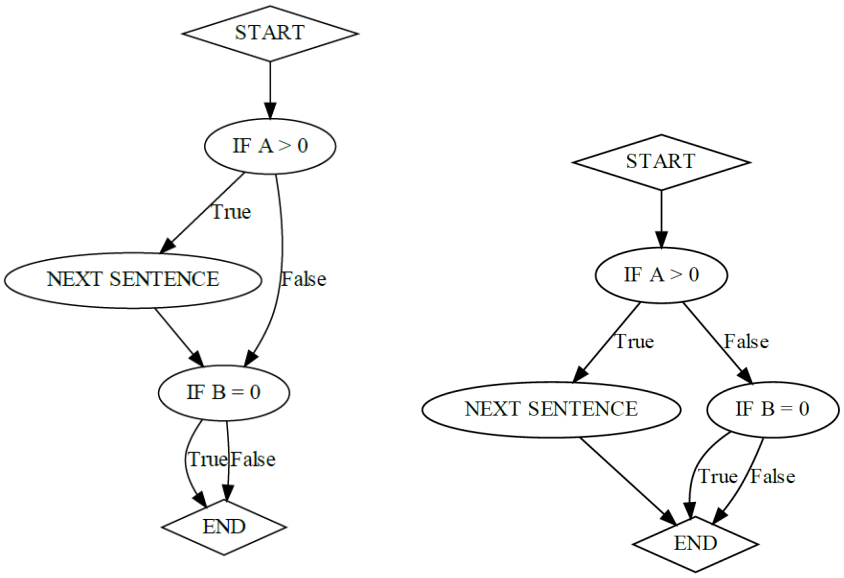


Figure 7.1: Generated CFGs for code example from Listing 7.2 and its variation with the DOT only on line 13

Listing 7.3: A small COBOL program with embedded SQL code

```
1       IDENTIFICATION DIVISION.
2       PROGRAM-ID. Example2.
3       ENVIRONMENT DIVISION.
4       DATA DIVISION.
5       PROCEDURE DIVISION.
6           IF A > 0
7             EXEC SQL SELECT * FROM TABLE END-EXEC
8           END-IF.
```

The parsing happens in a very similar way as described earlier with the addition of a *Parsable* anchor in the anchors found present in the program (on line 7, instead of the loop anchor found previously on that line in Listing 7.2). As before, the parsing starts with the `PROCEDURE DIVISION` and the `IF`. When arriving at line 7, the handler for the Parsable anchor creates a node and extracts the SQL code between the delimiting COBOL statements. This SQL is stored in the node as plain text for ease of representation with Graphviz, but it is also passed to an ANTLR parser, which creates a parse tree of the code that we will be able to use when we need to compare the SQL code pre- and post-migration of a COBOL program. The parsing is finished with the processing of the `END-IF` at line 8, and for this program we do not have any loops to clean up after the first phase. Figure 7.2 shows the resulting CFG for

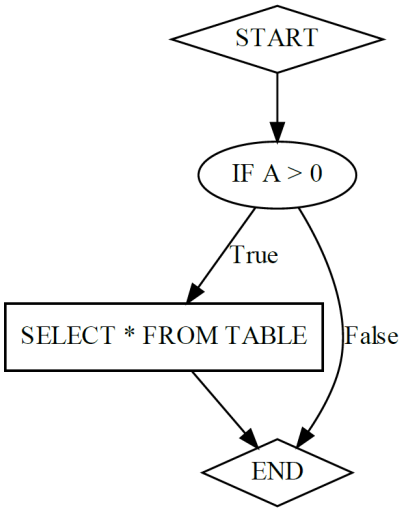


Figure 7.2: Generated CFG for the code example from Listing 7.3

the code of Listing 7.3, with the embedded SQL shown in boxes to visually differentiate it from COBOL.

7.4 Generalisability

We designed our semi-parser to be as generic as possible, achieving this through the configuration file. For COBOL itself, the flexibility of using regular expressions allows us to handle the slight differences that exist within the various dialects (of which there are over 400), which is one of our main motivations for our choice of using a custom fuzzy parser and not a full parser.

Furthermore, handling other languages that are close to COBOL conceptually (i.e. that are imperative and procedural) should be light work, adapting the regular expressions to match the syntax of the new target language. In the case of FORTRAN for example, the if statement is fairly similar: `IF cond THEN body END IF`. The end marker needs to change to have no dash (in COBOL, it is *END-IF*) and the "THEN" is mandatory and not optional. The comments are also different, we no longer need to ignore lines with * at the 7th column, but any character after an *!*, and so on.

Adapting the semi-parser to handle a language conceptually further from COBOL, like Java for example, will require to modify the code of the parser itself and not only its configuration file. This is because in its current form, the semi-parser can handle only one file at a time, and and would not be able to capture object-oriented concepts like inheritance, method overloading and method overriding. In general, the technique of fuzzy parsing matches

well the paradigm of imperative programming which is how we scoped our use case. We do note that control-flows have been extracted from various languages and paradigms, like the case of Prolog [Luo+92] or Java [Ami+12], although Prolog control-flows represent the flow but no longer the structure of the program and are therefore too far away from what we are doing here and from what the industrial case required.

7.5 Conclusion

In this chapter, we presented the part of our research that focuses on the extraction of significant constructs from COBOL code and their translation into CFGs, which are our chosen models. We explained why the constructs we extract are most appropriate for our specific use case, then detailed how we developed a semi-parser capable of extracting the desired information. We paid special attention to keeping our parser as flexible as possible, both in terms of picking exactly what will be extracted and the languages it can handle.

While fuzzy parsers are not new (dating as far as 1997 [KOP97]), they were not used previously in the context of creating control-flow graphs (full parsing being the most common method). As mentioned, many tools exist to create CFGs, but none were able to do so for our target language, COBOL. We did find one extension for Visual Studio Code, called *COBOL Control Flow* [Mic23], but it did not produce complete control flows for our industrial data, instead creating only a few connected nodes and many disconnected ones.

Validation of the fuzzy parser

8

In this chapter, we present how we evaluated our semi-parser and CFG generation both from a qualitative and a quantitative point of view. We first go over the methods we used to ascertain that the produced CFGs are correct, and then compare how the performance of our fuzzy parsing approach fares against using a full parser instead.

8.1 Qualitative Results – Correctness

The first step we took to ensure that our control-flow graphs were correct was to manually write small pieces of COBOL code to act as test files. We wrote some containing basic forms of the structures we wanted to analyse, along with some interesting edge cases, or cases that generated bugs during development. Using those as a base, we wrote a test suite to examine and verify the details of what was found during the parsing phase: which anchor was triggered when, what nodes did it create, and whether or not the content of those nodes was what we expected.

We also took care to test the produced CFGs, i.e. contained nodes and links between them. Listing 7.2 was part of this test suite, as well as its variation in Listing 7.3. One of our tests makes sure that both variants indeed produce the two different outputs shown in Figure 7.1. In total, we wrote 95 tests over 121 different handcrafted files, considering nested conditions and loops, some EXEC SQL to test our integration with ANTLR, the inherent case insensitivity of COBOL as well as difficult cases where strings or comments contain fragments that could be mistaken for parsable code.

For all of our handcrafted files, we test them in several phases: first, we test that our parser extracts the expected amount of anchors from the file. Secondly, we ensure that, once the CFGs are constructed, they have the correct number of nodes, and then that those nodes are of the correct type regarding to the constructs they originated from. Finally, we look at all of the links of the graphs and make sure that they are labelled correctly.

While this test suite was very useful during the incremental development phases of our fuzzy parser, it was still limited in a number of ways. Since our collaboration with Raincode Labs gave us access to their industrial-strength

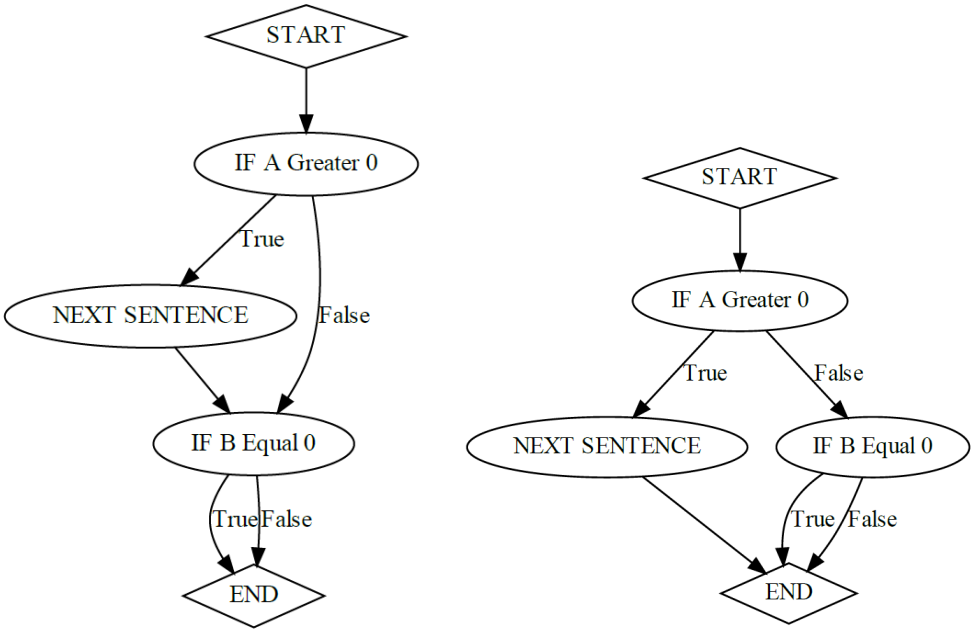


Figure 8.1: Outputted CFGs for code example Listing 7.2 and its variation using Raincode Labs' parser

full parser for COBOL, we took advantage of that.

We wrote a script to transform the parse trees created by their full parser to CFGs and compared these CFGs to our own. The comparison itself was done manually by probing diverse programs. We found that the CFGs were semantically equivalent in all cases, even if the syntax used was not always exactly the same. COBOL contains a lot of synonyms, which the Raincode Labs parser normalises (e.g., characters like < and = in the input appear as GREATER and EQUAL in the tree). The resulting CFGs produced from the Raincode Labs parser outputs for the code in Listing 7.2 and its variation can be seen on Figure 8.1. Apart from the small syntactic differences mentioned above, they are equivalent to the CFGs in Figure 7.1.

8.2 Quantitative Results – Efficiency

To compare their performance, we applied both our fuzzy parser and the full parser of Raincode Labs to four sets of files: our test files, the NIST COBOL test suite [Nat22] (a well-known open-source test suite for COBOL85), and all COBOL files both pre- and post-migration of one of the recent PACBASE migration projects. In total, that gave us 32 small handcrafted files, one open-source project of 410 files, and two sets of 3014 files each. Whereas the first

	Av. #LOC	Fuzzy tot. (s)	Full tot. (s)	Speedup fuzzy/full
Test files (32)	15	0.03	9.35	312
NIST (410)	781	37,45	127.87	3.4
Post-migration (3014)	1715	273.01	959.12	3.5
Pre-migration (3014)	2675	1272.29	947.53	0.7

Table 8.1: Execution times on all file sets

two sets are available to reproduce our experiment, the files of the PACBASE migration use case are confidential.

The setup of our experiment is as follows: for each COBOL file we run the first phase of our fuzzy parser (creating the nodes, but not linking them) and Raincode Labs' full parser. This is the fairest comparison we could come up with: at that point, our fuzzy parser has not fully generated the CFG yet, but from Raincode's side, we are left with a full parse tree that still needs to be transformed to a CFG as well. We chose not to measure the time it would take to obtain a CFG for both methods because in the case of the full parser, getting the CFG requires the parse tree to be written to disk in XML form and then be read from disk again.

The alternative would be to modify their parser so that it creates CFGs immediately, but this would have required substantial help from someone at Raincode Labs, for no other purpose than this performance comparison.

Also note that our fuzzy parser uses ANTLR to parse the embedded SQL code found inside the COBOL files, while the full parser leaves EXEC blocks untouched. We performed our comparisons both with and without parsing the embedded SQL, and found that running the ANTLR parser does not pose too much of an overhead, taking roughly 20 seconds for all 3014 files of our industrial use case (only those files actually contain EXEC SQL statements with embedded SQL code). The execution times reported in this section were computed with ANTLR parsing turned on, presumably giving our fuzzy parser a slight unfair disadvantage in the comparison with the full parser. There are limits in comparing tools in practice, and we hope our explanation helps the readers to interpret the results.

The Raincode parser reports its parsing times in a database while our fuzzy parser uses Python's `time.time()` to obtain the execution time. For each file, we run each parser 20 times and then calculate its average execution time. All of our tests were performed on a 12-core, 24-thread 3.70 GHz processor, with 32 GB of 3500 MHz RAM. We did a few experiments with a warm-up phase of 20 runs, but saw no significant changes in our numbers: neither for the fuzzy nor for the full parser. We therefore do not include warm-ups in our final reported numbers. The summary of our findings can be found in Table 8.1. The average file size for each codebase is included, as it turned out

to be relevant for the discussion to come. We then give the sum of the average execution time over the entire codebase for both the fuzzy and the full parser, and finally the speedup of using our fuzzy parser versus the Raincode full parser, for each codebase.

When analysing these results, we observe that the fuzzy parser offers a significant speedup when the files are small, but loses its advantage as the average file size grows larger. A likely explanation for this phenomenon is that the Raincode Labs full parser is I/O bound, meaning that reading a file of any size is a fairly costly operation for them, while the parsing task itself has become very optimised over their parsers' 20 years of existence. On the other hand, our fuzzy parser, implemented in Python, scans the input files faster, but sees its execution time grow as the files become longer and more complicated due to the number of regex matches to find, and the complexity of keeping track of embedded conditions and loops to generate the CFG. This explains why the full parser actually outperforms the fuzzy parser in the last line of Table 8.1. The files pre-migration are quite large and contain lots of embedded structures (their inappropriate nestedness levels is one of the reasons for refactoring), providing a worst-case scenario for our fuzzy parser, which then becomes slower than the full parser. Both parsers seem to have near-linear performance.

To confirm this phenomenon, we created a synthetic test set. We took a base file of 1000 lines and grew it in size by copying parts of its contents up to 30K lines. We then ran both parsers on these manually-generated files to observe their behaviour. The resulting graphs are shown in Figure 8.2. On the top graph, we can observe that the fuzzy parser is faster when the files remain under 10K lines, but that execution time keeps growing with the file size, while the full parser's execution time grows more slowly, beating the fuzzy parser's execution time for files larger than 10K lines. The bottom graph plots each parser's execution time normalised by the file size, i.e. the "cost" of running a parser. As expected, running the full parser on small files has a very high cost, but this cost decreases sharply as the file size grows to 5000 lines, and keeps decreasing slowly afterward. In a contrary fashion, the fuzzy parser is less costly to run on files smaller than 5000 lines, then roughly equivalent to the full parser's cost up to files of 10K, but then gets slightly costlier as file size increases.

To obtain these numbers, we did some initial optimisations on our fuzzy parsing code, and pre-compiled it using the Nuitka Python compiler [Hay+22], which eventually allowed us to get a performance roughly equivalent to (or better than) that of the full parser for most files in our experiments. We would like to emphasise here that we are comparing the current implementation of our fuzzy parser prototype to an established highly-optimised industrial-strength parser. Further promising optimisations of the fuzzy parser exist, in

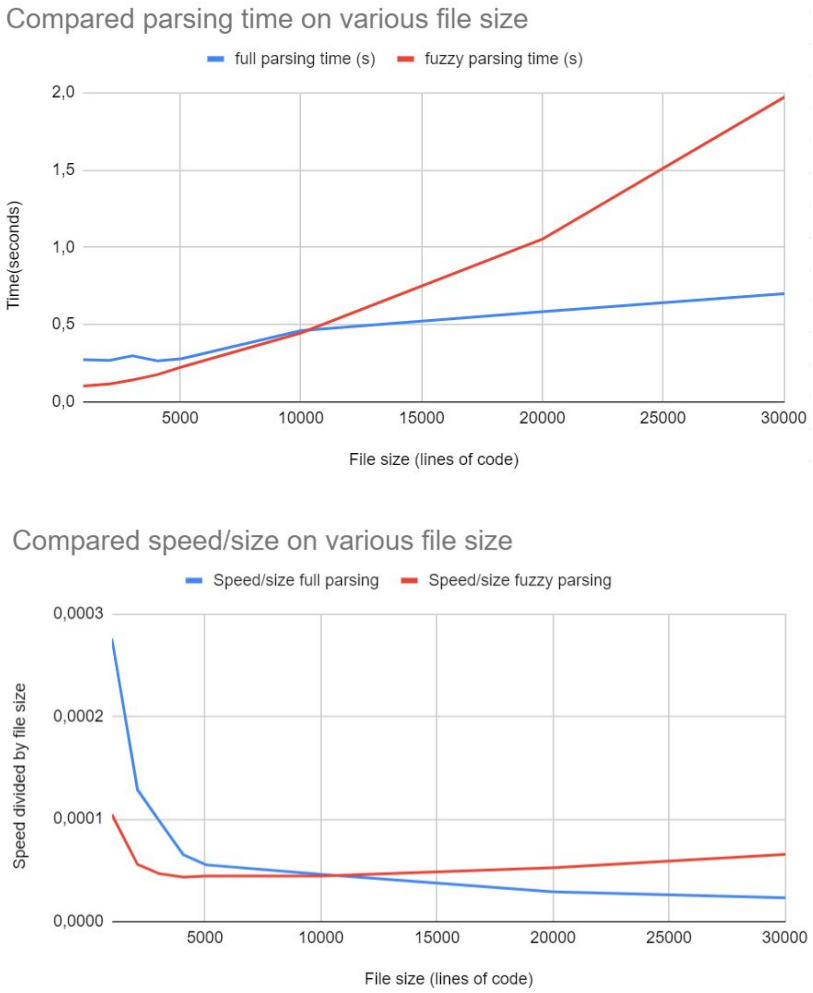


Figure 8.2: Graphs comparing both parsers' performance on various file sizes

particular to get rid of the slow regex module in favour of something faster.

While the execution times are significant (20 minutes for our longest run), this is not how our parser would typically be used in the industrial use case. In our experiment, we ran every file one by one in order to be able to compute the average execution time for all of them. The fuzzy parser would be used either while in a meeting with a customer, or run on servers overnight after a full migration task. In the first case, the files would still be computed one by one, but only two at a time during discussions, requiring 0.42s on average for a pre-migration file, and 0.09s for its post-migration equivalent, which is more than acceptable and lower than the time that would be needed by the full parser (respectively 0.31s and 0.32s on average for pre- and post-migration

files). In the second case, on a server, the process would be run in parallel to a very computationally heavy rewriting system for COBOL refactoring. The execution of our fuzzy parser is fairly lightweight and thus multi-thread-friendly, requiring only 5% of the processing power and 40 MB of RAM on the computer used for the experiment. This means that, putting the refactoring processes aside, up to 20 threads could be run in parallel on that specific machine, resulting in a total execution time of 77.2s to compute both sets (threads are fully independent), or just over a minute. For an overnight process, this overhead is again more than acceptable.

8.3 Discussion

In this section, we discuss the viability of using semi-parsing techniques like fuzzy parsing, versus using full parsers, to generate CFGs; criteria of correctness, efficiency and modularity; as well as the semi-parser's ability to be dialect-agnostic and its applicability to partial or non-compilable code.

8.3.1 Efficiency and Correctness

Both these criteria were discussed at length in the previous section. While our CFGs generated with a semi-parser were found to be correct when we analysed them in detail, using the fuzzy parser will not always result in faster execution times as compared to using the industrial-strength full parser we had at our disposal, especially not for larger programs. We already proposed a few solutions to this issue, the easiest one being to parallelize the execution of the CFG creation process when dealing with many files of larger size. However, execution time is not the only factor we should consider when talking about efficiency. The time resources that need to be invested before getting results are also of importance. Of course, if a full parser is already available for a given language, writing a script that extracts CFGs from its parse trees may indeed not be more complicated than writing a semi-parser. However, whereas we did have access to such a full parser for the case of COBOL (and welcomed it because it allowed us to compare the correctness and efficiency of our fuzzy parser with that full parser), especially when working with legacy languages we do not always have access to such a full parser. When no parsers are readily available, the time and expertise required to write a semi-parser will be far less than what is needed for writing a full parser. In our case, it took us a few months to obtain satisfying results for our fuzzy parser versus the estimated 2-3 years it could have taken for writing a full parser [Mas98].

8.3.2 Modularity

We can distinguish two important modularity concerns: the ability to cherry-pick what we want to be contained (or not) in the produced CFG, and the ability to parse other embedded languages. When creating CFGs from parse trees, picking what will be included in the final graphs is fairly easy, although less flexible than the configuration file that our semi-parsing solution proposes. In regard to parsing embedded languages, combining two grammars into a single one is often difficult to impossible as explained in subsection 7.2.1. As we experienced, at least for the case of parsing embedded SQL in COBOL, semi-parsing makes this possible relatively straightforwardly.

8.3.3 Dialect agnosticism

It is well known that many languages, and legacy languages in particular, have multiple dialects or versions, COBOL being a particularly impressive case with up to 300 language dialects [LV01]. Using a fuzzy parser, as long as the syntax of the structures we are interested in remains roughly the same, the dialect used would not interfere much with the CFG generation. In case of a diverging syntax for one of the control-flow structures considered, the only change needed would likely be to modify the regex expression describing it in the configuration file, making the adaptability of our fuzzy parser to language dialects very high.

8.3.4 Ability to handle partial or non-compilable code

When using a full parser, the code needs to be complete and syntactically correct in order to be processed. With fuzzy parsing, this is not required (although severe syntax errors within chosen structures will of course yield incorrect CFGs). In our industrial use case, this proves very useful in cases where the code is confidential, and the customer wishes to give away as little as possible of their codebase. Imagine a contract by which the customer agrees to give only the procedure divisions to Raincode Labs, but not the data divisions, which can be seen as more critical to the customer because they contain all variable definitions and values, while the former “only” describes the execution. In this case, a full parser would need to be partly rewritten to analyse this code, while with a fuzzy parser, we would be able to do so without any modification.

8.4 Conclusion

In this chapter, we demonstrated the accuracy of the CFGs created by our semi-parser by checking their equivalence to CFGs generated from full parse

trees. We then evaluated the performance of our semi-parser to gauge its efficiency. The results indicated that for smaller files (less than 10K LOC), the semi-parser outperforms a fully optimised industrial-strength parser. However, it was acknowledged that optimisation is needed to enhance its competitiveness with industrial-level parsers when dealing with significantly larger files. We then discussed the advantages of using semi-parsing: it is lightweight and lower-effort to create, and proves useful when the need arises to parse partial code or several different dialects.

Trace equivalence

9

This chapter serves as a direct continuation of the preceding one, wherein we focus on comparing the Control Flow Graphs (CFGs) generated through our semi-parser using the concept of trace equivalence.

9.1 Context

In this chapter, we describe how we compare the CFGs of a COBOL program before and after the migration process by way of a trace equivalence algorithm. Initially, we encountered a challenge, as the CFG format contains information at both the node and edge levels, which is not compatible with the base algorithm for trace equivalence which wants labelled edges only. Moreover, certain details present in the CFGs are irrelevant to the comparison, e.g. the names of the labels; their exact names do not contain any semantic meaning and we know from experience that many labels found in the code before the refactoring will be removed in the new version. Consequently, we address this issue by automatically translating the CFGs first into Labelled Transition Systems (LTS). This translation effectively condenses relevant information, such as conditional statements, onto the edges, leaving the nodes empty except for their id.

With the translated models in hand, we then proceed to compare these LTS models using trace equivalence. We start with the base algorithm, identify its limitations, and then present our proposed solutions to address these limitations effectively.

In addition to the binary outcomes of the trace equivalence algorithm (i.e., the two models are either equivalent or non-equivalent), we recognise the need for a more nuanced representation of the results. By marking the nodes explored during the comparison, we enable a more granular view of the points where the graphs diverge. We then create a visualisation tool that enables a detailed presentation of the locations where the non-equivalence is identified, again using Roassal as for the tool presented in subsection 5.3.4.

Finally, we conduct a comprehensive validation of both the trace equivalence algorithm and the visualisation tool, involving feedback from Raincode experts for the latter.

Once again, while this work is targeted towards our specific industrial

use case, the tools we developed were conceived with maximum flexibility and generability in mind. Furthermore, the process of comparing two versions of a program before and after refactoring is prevalent throughout both academia and industry. Our tool is capable of processing any input as long as it respects the LTS format, and while it might not give the best results out of the box due to the specifics of every refactoring process (see our discussion in section 10.4), it is easily extendable.

9.2 State of the art and background

In this section, we present the background knowledge necessary to comprehend this chapter as well as some related work. We start by defining the models that we will use, Labelled Transition Systems (subsection 9.2.1) and then detail the algorithm that we use to compare those models, trace equivalence (subsection 9.2.2).

9.2.1 Labelled Transition Systems (LTS)

Labelled Transition Systems (LTS) is a graph format used, among other things, in model-based testing [Tre08]. It presents the advantages of being both fairly close to our initial CFGs and having been used previously in the context of our goal of trace equivalence [TPB97]. It is often used in computer science to describe *the behaviour of systems*, which is our goal here. Conceptually, it is close to FSAs, although there are a few differences. First, FSAs accept a certain language, meaning that they have accepting and non-accepting states. This is not the case for LTSs, which do not have to have an accepting state. Secondly, LTSs do not have to be finite while FSAs do. Note that the LTS that we will use are always, in practice, finite since they represent a finite set of code instructions, the program we are analysing.

LTSs are often presented as simple models of communication systems. They are composed of *states*, describing information about the system at a certain moment of its behaviour. In our case, each state represents one of the code structures we extracted from the COBOL code and has a 1:1 match with a node in our initial CFGs. Their *labels* or *transitions* stand for observable actions of the program or process they represent. Those match with the links of the CFGs, but with more formality. Since the states are not annotated, an `IF` statement that would be represented in a CFG with a node labelled *IF A = 0* and two outgoing links *True* and *False* would become a state with no label in of itself and two outgoing links: *A = 0* and *NOT(A = 0)*, Figure 9.1 shows a simple visualisation for this example, CFG to the left and LTS to the right.

LTSs have two types of links:

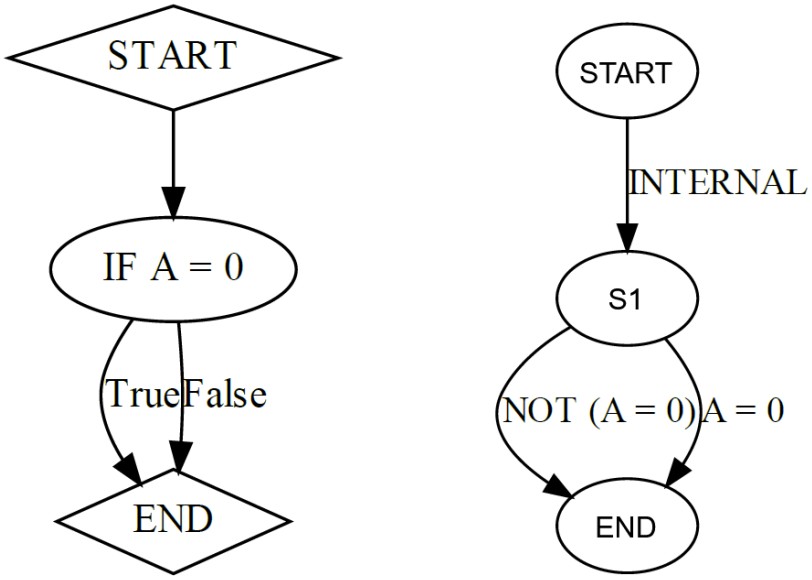


Figure 9.1: CFG (left) and LTS (right) representation of our small example.

1. Those with a specific label that is destined to be matched in the trace equivalence like we exemplified above and

2. Links that exist just to represent the flow of execution, denoted with an *INTERNAL* label.

During the verification of trace equivalence, those internal links are ignored, meaning that 0 to N *INTERNAL* links on one graph can be matched to 0 to N on the other.

In the next section, we will explain how we translated our CFGs into LTSs in order to use them to perform the trace equivalence.

9.2.2 Trace equivalence

Calculating equivalence between LTSs is not a novel idea [De 87] and the algorithm of trace equivalence has been used before to do so [Lev06]. Intuitively, two LTSs are considered trace equivalent if *they cannot be distinguished by interacting with them.* As described above, the LTSs are composed of *states* and *transitions*, and two states are considered trace equivalent if they can perform the same finite set of transitions.

The trace equivalence then takes the two LTSs corresponding to our pre- and post-refactoring versions and traverses them recursively. One full path

from the starting to the end state is called a *trace* and is considered valid if and only if it is composed of only trace equivalent states.

Consider the small example of a graph having a start state and then a state S1, as described above, with two outgoing links *A = 0* and *NOT(A = 0)*, both linking to the end state. This LTS would have two traces, as shown in Figure 9.2 in red and blue. Note that traces can include the same state several times due to looping structures.

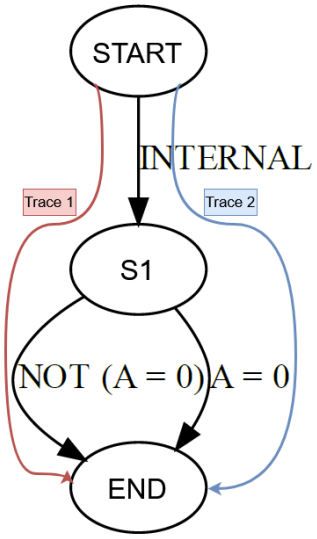


Figure 9.2: Traces in the LTS corresponding to our small example.

The algorithm will branch and generate new traces every time several paths can be explored, like in the case of a condition. All traces found in the first LTS must have a match in the second for the two graphs to be considered trace equivalent. The base algorithm for calculating trace equivalence between two graphs goes as follows, starting with the two start states of each graph:

1. Start at a corresponding state of each graph.
2. For each outgoing link of these states, look at each of their labels, generating new traces if there are more than one:
   - If it is *INTERNAL*, it can be consumed (skipped) and the algorithm goes back to step 1 with the target state as the starting one.
   - If it has a specific label, we must find an exact match for it in the other graph. In other words, both of the current states we are

analysing must have a copy of the exact same label. To come back to our small example, to match state S1, the other state must have both a *A = 0* and a *NOT(A = 0)* link.

3. When a link is skipped and/or matched, relaunch the algorithm recursively with its target state. If no match can be found for a link, the trace is invalid and therefore the graphs are not trace equivalent.

4. If a state with no outgoing link (end state) is reached or if a couple of states previously considered is explored again, the trace we are following is valid.

5. If all possible traces are valid, the entire graphs can be considered as trace equivalent.

To summarise, the trace equivalence algorithm takes two LTSs as input. As output, it gives a boolean answer: either the input graphs are equivalent or they are not. In addition to this boolean ouput, the version of trace equivalence that we implemented also marks every explored link from the graphs taken as input. Every path that exists in both input graphs is explored pairwise (one path from each graph). Along those paths, every link explored is marked "matching", if it is *INTERNAL* or if a link with the exact same label is available in the other path. The explored labels are marked "unmatching" if neither of the previous conditions are true. For the input graphs to be considered trace equivalent, no link must be labelled "unmatching".

In this base version of the algorithm, all labels that are matched between the graphs must have an exact correspondence, and states must be matched exactly one-to-one. This does not take into account any of the specificities that a refactoring can have and is fairly limiting. However, those limitations can be partially overcome, as we will demonstrate in the next section.

9.3 Calculating trace equivalence between two CFGs

In this section, we show how to compare COBOL programs before and after refactoring. We will present both the models used to represent the code, how we obtained them and the algorithm that compares those models. Finally, we will present the limitations of the algorithm and how we overcame them.

9.3.1 Transforming CFGs into LTSs

We start with the CFGs generated by our semi-parser, presented in chapter 7. As a quick reminder, they contain the following structures (as determined with the experts): the conditions IF and EVALUATE (switch/case style of multiple branching), the jumps GO TO, NEXT SENTENCE, PERFORM and

PERFORM THROUGH, the looping structure ("in-line PERFORM"). SQL calls are found between EXEC SQL and END-EXEC keywords. As PERFORM and GO TO make use of COBOL's paragraph names, they are also included.

However, as stated before, the generated CFGs are not yet in the right format for trace equivalence since they contain information both on their nodes and links. We did not find prior work on translating from this format to LTS, so we wrote the conversion script ourselves. While this translation might seem cumbersome, it is necessary for our purpose: we not only seek the guarantees given by the trace equivalence algorithm, but we also wish to visualise the results. For that, CFGs are preferred since they are more readable by humans.

Translating a CFG into an LTS requires thinking about how to replicate the flow that will be followed by the execution of the program and translating it to labelled links. For conditions, i.e. the IF statement, we label the branches with the base condition and its negation. For the jumps GO TO and NEXT SENTENCE as well as the labels, they are replaced with a single INTERNAL link: what the node was exactly does not matter in the execution, only where its link went and what link goes to it. Finally, for SQL nodes, the entire SQL statement becomes the label of the link. Listing 9.1 shows a small handcrafted COBOL code fragment, featuring most of the described constructs, and Figure 9.3 shows its equivalent CFG (on the left) and LTS (on the right). The numbering shown on the nodes is not included in either model and was simply added there to ease the reader's comprehension.

Listing 9.1: A small handcrafted COBOL code example.

```
1       IDENTIFICATION DIVISION.
2       PROGRAM-ID. Example1.
3       ENVIRONMENT DIVISION.
4       DATA DIVISION.
5       PROCEDURE DIVISION.
6       P1.
7           IF A > 0
8             NEXT SENTENCE
9           ELSE
10            IF B = 0
11              EXEC SQL OPEN DB END-EXEC
12            END-IF
13          END-IF.
```

The various PERFORM statements have to be carefully designed. As a reminder, PERFORM A and PERFORM A THROUGH B behave as follows: when encountered during the execution, they jump to the labelled paragraph A, execute everything contained in it, then the execution goes back to the PERFORM and continues. In the case of a THROUGH clause, one jump results in executing everything in paragraphs A and B and everything in between.

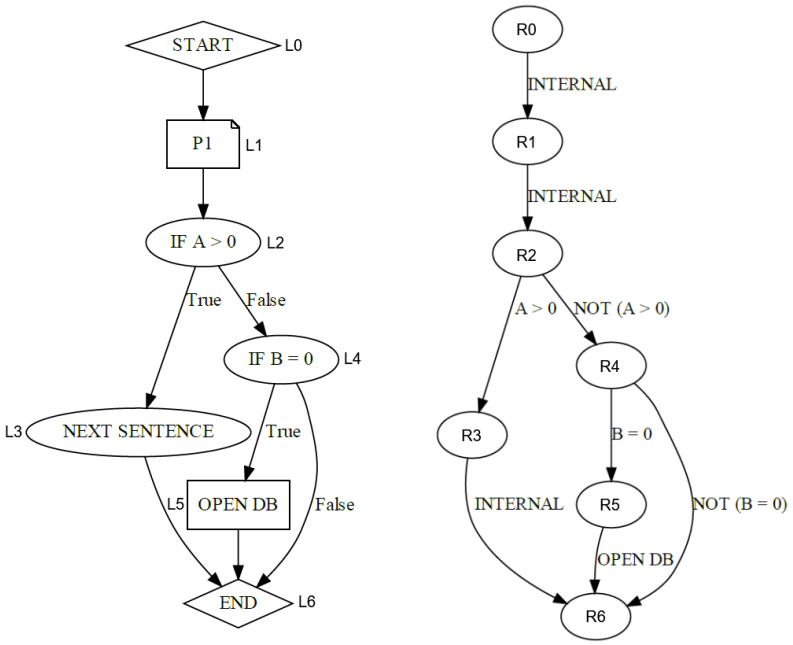


Figure 9.3: CFG and LTS representation of the code of Listing 9.1.

To handle this, we created two specific labels on the LTSs: the *PERFORM* and the *GO BACK*. A `PERFORM` node has both a *PERFORM* and an *INTERNAL* outgoing link. The *PERFORM* link has to be followed first, leading somewhere else in the LTS. The paths are then followed as normal until we reach the corresponding *GO BACK* that will lead back to the `PERFORM` node, allowing to proceed with its other child (*INTERNAL* link). The logic is the same for a `PERFORM THROUGH`, the placement of the *GO BACK* link being the only thing that changes. Note that, in the CFG, only the *PERFORM* link is shown, counting on the user's understanding of COBOL to infer when the execution returns (omitting the *GO BACK* links also reduces clutter). In a model meant to be automatically analysed such as the LTS, this link needs to be explicit. A CFG and corresponding LTS of this can be seen on Figure 9.4. In the example, the `PERFORM` is met first, then *P2* and its `IF`, then the execution goes back to the `PERFORM` node and keeps following the normal path, performing the test `IF A < 0`, then the `IF B = 0` a second time. For example, for non-zero value of `B`, the execution would be: L0-L1-L2-L4-L5-L7-L2-L3...

The in-line `PERFORM UNTIL` works similarly to the regular in-line `PERFORM` but with conditions instead of jumps to paragraphs. If its base condition is true, the body of the loop is entered and if it is the negation of the condition, the loop is exited and execution continues. An example of such a construct can be seen on the right-hand side of Figure 9.5 (CFG) and Figure 9.6 (LTS).

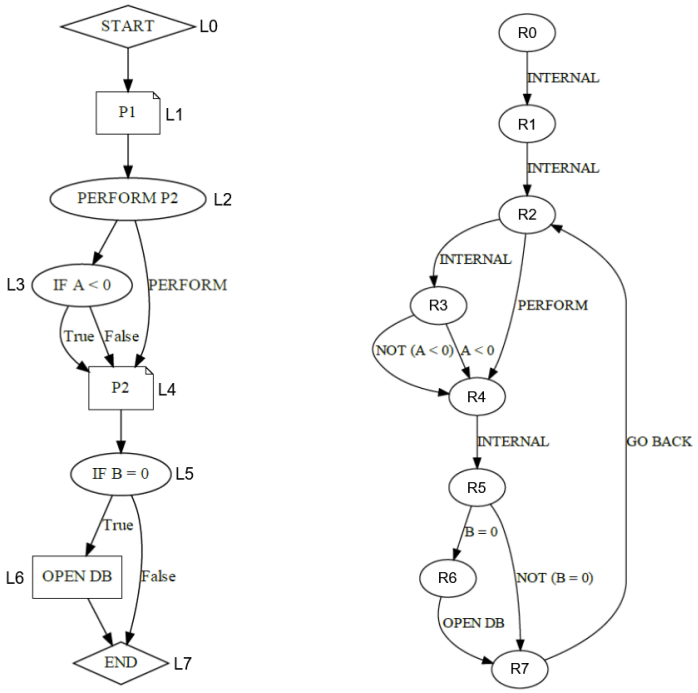


Figure 9.4: CFG and LTS representation of a `PERFORM` statement.

9.3.2 Calculate the trace equivalence between two LTS

Applied on the graphs shown on Figure 9.6 (which are the LTS version of the graphs of Figure 9.5), the base trace algorithm described in subsection 9.2.2 will perform as follows (some steps are combined for conciseness):

1. The INTERNAL outgoing links from nodes L0, L1 and L2 are skipped, meaning that node L3 is the next node. On the other side, the INTERNAL outgoing links from nodes R0 and R1 are skipped, making node R2 the next node.

2. Both sides contain two links, labelled `50 < J02IIR` and `NOT (50 < J02IIR)`. The execution continues and two separate paths are put on the stack: the result of following the links labelled `50 < J02IIR`: node L4 and node R3, and the results of following `NOT (50 < J02IIR)`: nodes L5 and R2.

3. Let's start with the `NOT (50 < J02IIR)` path: since node R2 has labelled links, the execution is paused. On the pre-refactoring side, after L5 we follow nodes L6, L1 and L2, each time skipping their *INTERNAL* links. We are then once again comparing node L3 with R2, and since

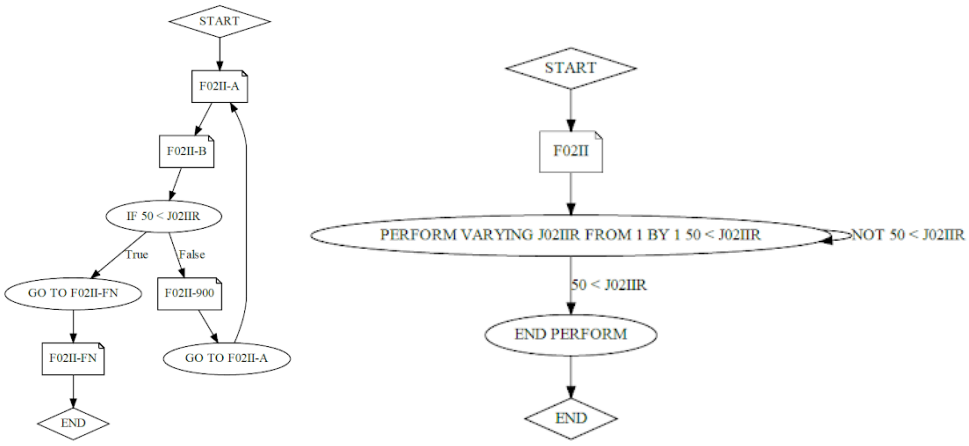


Figure 9.5: CFGs for a piece of code pre- and post-refactoring.

we have already considered them and matched their links, this path is valid.

4. Now looking at the `50 < J02IIR` path, we are at nodes L4 and R3. The *INTERNAL* links from nodes L4 and L7 are skipped, while the outgoing link from node R3 is skipped on the right-hand side. We are left with nodes L8 and R4 which both have no outgoing links, meaning that this path is valid.

5. Since both paths are valid, the graphs are considered trace-equivalent.

9.3.3 Handling conditions through semi-parsing

While this basic algorithm works for handcrafted examples as we have shown so far, it lacks a few things to be able to handle the reality of our use case of automated COBOL refactoring. The first thing missing is some awareness of how conditions work. E.g., when refactoring, it is common to flip the condition of an `IF` statement to swap the code contained in its else branch to the main branch. In this case, `A < 0` would become `A >= 0` and both versions would not be considered equivalent since their string representations are different (`NOT (A < 0)` versus `A >= 0`).

With COBOL in particular, this problem is very prominent since, as explained in section 2.2, it supports contracted expressions and some specific values are expressed with keywords. This flexibility of COBOL means that many semantically equivalent conditions would not be detected as such with the base algorithm.

This is an issue since generated COBOL tends to shorten conditions, choosing not to repeat variable names and operators as much as possible, whereas

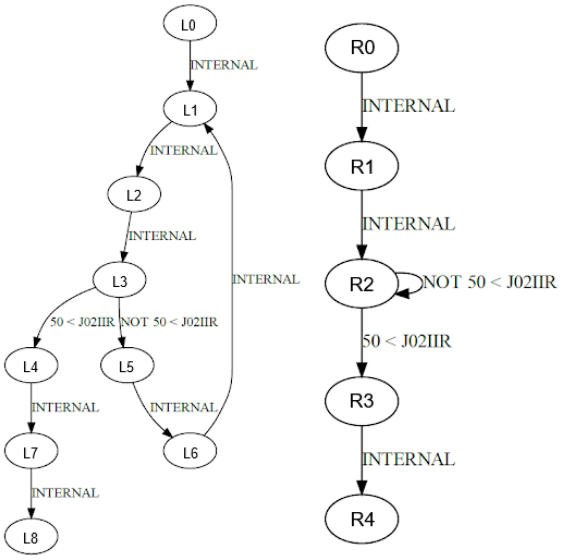


Figure 9.6: LTSs corresponding to the CFGs of Figure 9.5.

code owners would prefer the longer more human-readable versions, meaning that we encounter such mismatches fairly often.

To circumvent this, we wrote a linear semi-parser [Zay14] focused on COBOL conditions only, allowing us to handle all cases mentioned above and more. It is based on a COBOL-compatible tokenizer and a parsing automaton shown in Figure 9.7. The semi-parser works in linear time and expands all contracted expressions to their full canonical form, allowing us to compare them.

Our way of dealing with contracted expressions is another example of how we use the semantics of the chosen language (COBOL) inside our differencing algorithm. With some creative use of parentheses, a counterexample that our condition semi-parser cannot successfully identify can theoretically be constructed, but since such parenthesised expressions do not occur in generated COBOL code, the semi-parser is correct *enough* for its practical use.

Let us exemplify how the condition semi-parser works on a very condensed condition: `VAR NOT = ' ' AND 0`. Tokens are simply considered to be separated by a space so we have 6 of them here.

- We start on state 1, and we have three options: either we find an *operator*, a *word* or a *not*. Our first token is `VAR`, which is neither an operator nor a not, so we move to state 2.
- Our next token is `NOT`, which means we follow the link to state 6.
- In state 6, we need to either have another *not* or an *operator*, which

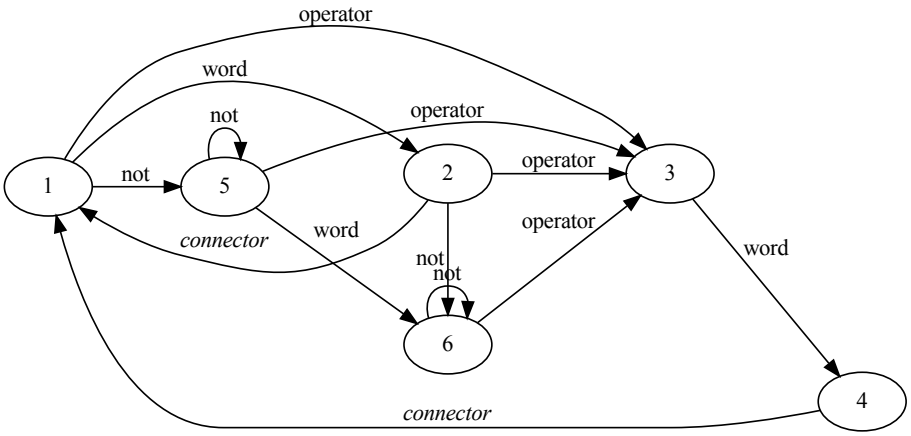


Figure 9.7: The parsing automaton for contracted expressions: connectors are AND and OR, and operators are =, >=, etc.

corresponds to our = token and takes us to state 3.

- State 3 expects only a *word*, which is the token ' ' and takes us to state 4.
- State 4 again has only one exit state requiring a *connector*, in this case AND and going to state 1.
- We are back to state one and parse the token 0 as a *word*, leading us to state 2.
- There are no more tokens to parse, however, state 2 is not a state we can stop at without doing some more work, although we will obtain a valid condition (which is not the case for states 3, 5 and 6).
- We detected that we have a left part of our condition that is fully formed, then an operator, and then a condensed condition. We will therefore copy the word, the not and the operator that we found on the left side to the right side, giving us our final condition of `VAR NOT = ' ' AND VAR NOT = 0`.

The use of this condition semi-parser does make our approach more specific to COBOL, but it also greatly improves the quality of our output. Anyone wishing to reproduce our work should reflect on how much effort would be necessary to implement a version suitable for their purpose and reflect on the tradeoffs. It is also important to note that this is independent and our proposed solution can work without it, although with inferior results.

9.3.4 Error recovery

The last piece necessary for the trace equivalence algorithm is error recovery. Sometimes, the COBOL generator produces code that does nothing, such as PERFORM statements jumping to paragraphs containing only an EXIT statement, which is an equivalent of `pass` in Python. When refactored, these paragraphs are deleted altogether. In such cases, the trace equivalence would try to match the *PERFORM* and *GO BACK* links, but since they have no equivalent anymore in the refactored version, the path is considered as non-equivalent and is not explored further. In other cases, the generator created redundant `IF` conditions, e.g. `IF A < 0 AND A < 10`, that get refactored to only the stronger condition, `IF A < 0`. In those cases as well, the traces would be marked as invalid even if this is the only difference there is.

To handle this better, when no match is found for a link, the trace equivalence algorithm enters the error recovery mode: it is allowed to skip up to N links (where N is defined in a configuration file), considering them as if they were labelled *INTERNAL*, until it can find a match again. If no match can be found after exploring N links, the path is declared invalid. If there is a match further on, all skipped links are marked as "unsure" (to be checked by a human afterward) and the execution continues as before.

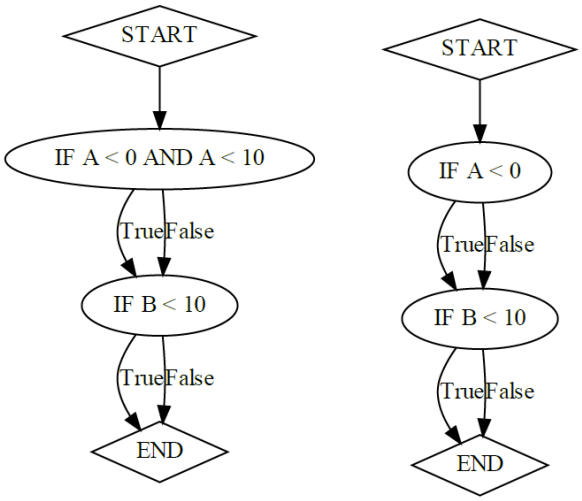


Figure 9.8: CFGs for our small example, before (left) and after (right) refactoring

Figure 9.8 shows the graphs for our example of `IF` with the condition refactored (we have chosen to show the CFGs for ease of reading). In this case, the error recovery will provide better results. On the left side of Figure 9.9

(showing the LTS corresponding to the left-hand side of Figure 9.8), we show the way links are matched by the trace equivalence without error recovery: the first *INTERNAL* link is matched, but both links coming from the first IF are not matched, and the graphs are not explored further. On the right side, the execution with error recovery is shown: both links of the first IF are now marked in orange as *unsure* since the algorithm could find a match further down, for the second IF.

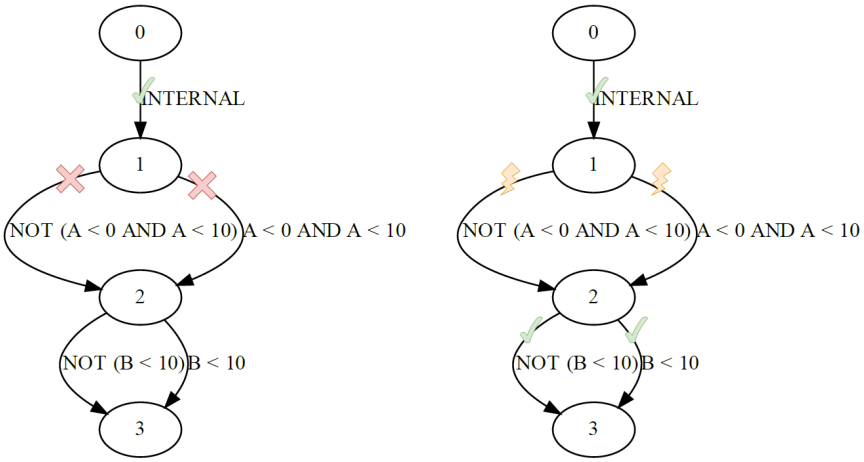


Figure 9.9: Trace equivalence execution on graphs from Figure 9.8, without (left) and with (right) error-recovery.

9.3.5 Special cases

Even equipped with the contracted expression parser and the error recovery mode, the trace equivalence algorithm still occasionally produced unsatisfactory results. This is due to the fact that it needs to match links one to one, making the consideration of some more elaborate refactorings impossible. In particular, there are two specific cases that the COBOL generator tends to create which are then refactored.

The first case concerns several nested IF statements that get refactored into a single IF having all conditions linked by AND keywords, as illustrated in Listing 9.2 (pre-refactoring) and Listing 9.3 (post-refactoring).

Since both conditions A > 0 and B = 0 are on different links in the pre-refactoring version of the graph while being on a single link in the refactored version, the base trace equivalence algorithm will not find a match between both graphs for the presented snippets.

Listing 9.2: Nested IFs before refactoring.

```
1        IF A > 0
2          IF B = 0
3            (...statements...)
4          END-IF
5        END-IF
```

Listing 9.3: Refactored nested IFs.

```
1        IF A > 0 AND B = 0
2         (...statements...)
3        END-IF
```

To handle this situation, which appeared frequently in our use case, we implemented a strategy that allows the algorithm to match several links to a single one, constructing the more complex condition by linking the smaller ones with the AND keyword. We move through the `IF` links as long as we find them nested into each other, testing at each step if we succeed in constructing a condition equivalent to the longer one. If we do, all explored links are matched to the first one and the trace equivalence continues from there.

The second special case that we implemented, is the refactoring of several `IF` statements testing different values for a same variable into an EVALUATE. The technique is similar to the first described, except that every branch of the EVALUATE must have its equivalent in an `IF` while all the False branches of the `IF`s are matched to the WHEN OTHER (equivalent of `default`) from the EVALUATE. A concrete example of this refactoring pattern can be seen on Listing 9.4 (pre-refactoring) and Listing 9.5 (post-refactoring).

Listing 9.4: Successive IFs before refactoring.

```
1        IF A = 0
2            (...statement 1...)
3        END-IF.
4        IF A = 1
5            (...statement 2...)
6        END-IF.
7        (...statement 3...)
```

Listing 9.5: Refactored successive IFs.

```
1        EVALUATE A
2          WHEN 0
3            (...statement 1...)
4          WHEN 1
5            (...statement 2...)
6          WHEN OTHER
7            (...statement 3...)
8        END-EVALUATE
```

These structures are currently the only special cases that we have implemented. While other such cases could have been identified and implemented, we decided not to do so because the risk of making our tool less general outweighs the benefits of having a few more matches.

9.4 Generalisability

To maintain a broad applicability for the trace equivalence comparison, our approach focuses on representing COBOL behaviour primarily through the LTSs. This approach enhances the generalisability of our technique, since it allows to focus on only the models in order for the comparison to work on a new target language. Outside of the LTSs, the elements of the trace equivalence algorithm that are heavily dependent on COBOL are the condition semi-parser, a component that can be deactivated or exchanged for a condition parser of the target language. Furthermore, since the condition semi-parser is based on an automaton describing the way conditions are handled in COBOL, it would be light work to create a new automaton describing the conditions in the new target language and adapt the semi-parser we created.

The extensions related to if factorization and if-to-evaluate translation can be readily customised for a different programming language by specifying the appropriate keywords, corresponding to the equivalents of AND and EVALUATE in the target language. These extensions can also be easily deactivated as needed.

The sole COBOL-specific aspect of the trace equivalence algorithm involves the use of a stack to track the PERFORM links and determine when it is appropriate to follow the corresponding GO BACK links. This aspect will require to be adapted when transitioning to a new target language. Note that any new language will most probably also have its own specificities that will need special handling, for example, dynamic binding in object-oriented languages.

9.5 Conclusion

In this chapter, we presented how we translated the CFGs generated by our semi-parser to LTSs, in order to be able to use them in the trace equivalence algorithm. This translation algorithm is original work, as we did not find prior work that could the conversion. We have given an intuition on how the trace equivalence algorithm works in its most basic version, and demonstrated how it limits the amount of successful matches that we can obtain when comparing code before and after an automated refactoring. As shown in a survey [AlO+21], showing that refactoring does not alter the behaviour of an artefact is common and often done using techniques of model checking

like we did in this work. But code is not the only kind of artefact that can be refactored and analysed; models like UML have been the target of techniques similar to ours [HA23; HA08].

We proposed custom improvements to the base algorithm in order to improve the amount of states matched in the LTSs and implemented these improvements. We paid attention to keeping the solution from reimplementing the refactoring rules created by Raincode Labs. Indeed, we want to avoid this for two reasons: first, we want to keep the tool as generic as possible and second, implementing every refactoring rule risks introducing bugs in our tool.

Validation of the trace equivalence algorithm

10

In this chapter, we first provide some metrics about our use case itself, discuss how we designed our validation process, then explain the results from both our manual and automatic validations as well as threats to validity.

10.1 Partitioning the dataset

From the refactoring experts, we received a corpus of 3014 COBOL files varying greatly in length, with files as small as 128 LOC and as big as 22'246 LOC. Whereas the mean file size pre-refactoring was 2680 LOC, after refactoring it became 1719; and in total respectively 8 084 770 and 5 184 942 LOC. On this amount of data, it was impossible to perform a full manual qualitative analysis. We therefore started the validation process by partitioning our files into representative categories.

We chose to create partitions according to how significantly the automated refactoring affected the files. For this, we analysed the decrease of the file sizes in lines of code due to the refactoring process, simply calculated by: $(LOC_{init} - LOC_{new})/LOC_{init}$. This means that if a file went from 200 to 150 LOC, its size was reduced by 25%. Applying this formula to our entire corpus, we obtain the distribution shown in blue on Figure 10.1.

We observed that the most common diminution is 21% to 30%, but it can go up to 86% for some files. On the other extreme of the scale, we can see files with a negative percentage, meaning that their size in LOC was *increased* by the refactoring process. This can be explained by the fact that, while it aims mostly at simplifying the generated code structure, the refactoring process also targets code readability. For this, generated comments are sometimes added to the program. We manually checked those files with a negative percentage, and confirmed that their size increase was indeed caused by the addition of comments.

Comments are ignored by our tool as they cannot contain code — being originally a punchcard-based language, COBOL only supports full-line com-

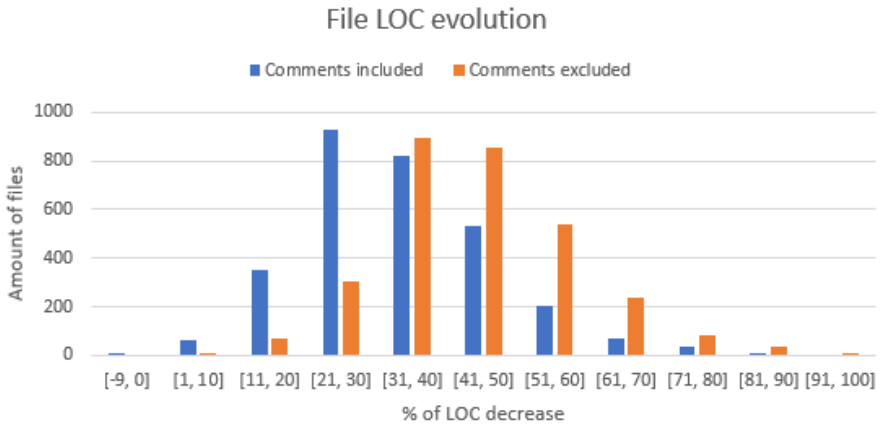


Figure 10.1: Percentage of decrease in LOC in files after the refactoring, with and without comments included

ments. Thus, we normalised the previous data, excluding comments from our LOC count to get the distribution shown in orange on Figure 10.1. The lowest amount of LOC decrease is now 5%, the highest 93%, and the most common range between 31% and 40%.

The second metric that was analysed was comparing the evolution of the size of our LTSs to the normalised version of the evolution of the size of the LOC presented above. Ideally, we want the difference between the two to be as low as possible so that the LTSs reflect the changes as accurately as possible. This was calculated by applying the same formula described above to both the file and LTS size, then taking the absolute value of the subtraction. This is shown on Figure 10.2. We can see that more than half the files have a low difference of less than 12%, but there are some files that have variations up to 50%.

We manually analysed the two files having the most variation and noticed that they both had the same kind of refactoring patterns. A shortened version of our findings is shown in Listing 10.1 and Listing 10.2. This is a particular variation of the `IF` to `EVALUATE` that we presented above. While the two pieces of code do not differ much in amount of LOC, they vary in the amount of nodes they will create in the LTSs. The pre-refactored version generates 3 nodes: the `IF` having two links, the `NEXT SENTENCE` and the `GO TO`, totaling 4 links. The refactored version is only a single `EVALUATE` node and one link, which explains why we have such a discrepancy between variations of LOC and links.

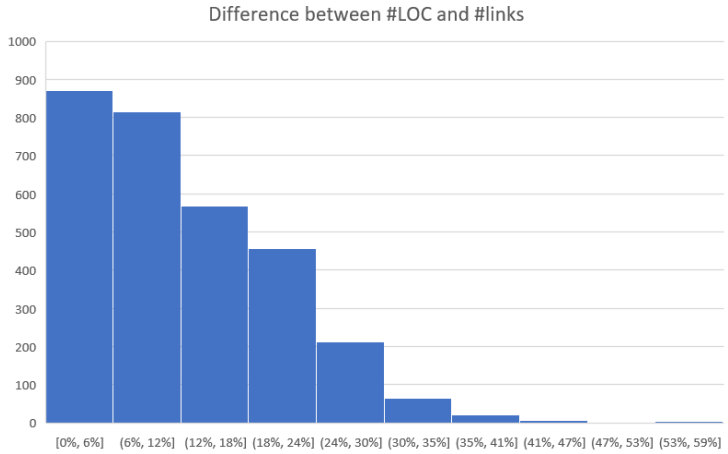


Figure 10.2: Difference in evolution between LOC and links

Listing 10.1: Generated If statement.

```
1       IF A = 0
2           NEXT SENTENCE ELSE GO TO PX.
3       "Statement
```

Listing 10.2: Refactored evaluate.

```
1       EVALUATE A
2         WHEN 0
3           "Statement
4       END-EVALUATE
```

This analysis allowed us to separate the files into five partitions regarding their percentage of decrease, each partition of more or less equal size, containing between 500 and 600 files each (see Figure 10.3). For our manual analysis, we then picked 5 files at random from each of these 5 partitions.

10.2 Manual analysis

The manual analysis that we performed, can be split into two phases: first, we ran the equivalence algorithm with no error recovery, analysed the results, then re-ran it with error recovery set to skip up to 20 nodes. We chose this number experimentally by running the algorithm for different values of this threshold and found that a value of 20 provides the best trade-off between time of execution and increase in amount of matches. With a low amount of skipped nodes (5 to 10), the amount of matched nodes grows very little, meaning that the algorithm does not allow enough additional exploration for the error recovery to become effective. With a much higher amount of

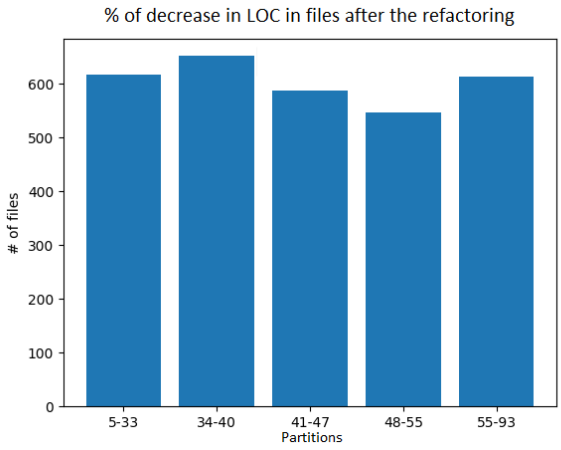


Figure 10.3: 5 partitions for the files

skipped nodes (40), the execution time explodes while only gaining a few more matches, meaning that time is lost backtracking with no benefits. The final number of 20 nodes corresponds to an allowance of 5% of states to be skipped at maximum, but this analysis should be repeated if this technique is applied in other contexts. We analysed the output of the algorithm via logs from the trace equivalence.

In the first phase, our first step was to check if the files could be shown to be trace equivalent. If so, we moved on. If not, we looked for a reasonable explanation as to why the two versions (pre- and post-refactoring) could not be matched to find further limitations in the implementation of our tool. In cases where the files could be declared trace equivalent without error recovery, we did not perform the second phase of the analysis. In the other cases, we performed a similar analysis on the output given from running the algorithm with error recovery.

Out of the 25 manually analysed files, only 4 (16%) were detected as trace equivalent without error recovery. Adding the error recovery allowed us to show equivalence for 6 more files, increasing our total number of files shown trace equivalent to 10 out of 25, or 40%. For 8 more files, the error recovery greatly increased the amount of links that we were able to match, indicating that it is indeed very helpful to the process.

Going into more detail, our analysis showed that the easiest files to show equivalent are the smallest in size, although the partition they belong to does not seem to be of much influence. Indeed, when looking at our manual analysis, we got files that can be shown equivalent in all 5 partitions we created. The size of the file (both in terms of LOC and of LTS transitions) seems to be

a more important factor than how much of the file size was affected by the refactoring.

This validation also allowed us to both find limitations in our tool and report insights to Raincode Labs: when refactoring a complex condition, its various members mostly stay in the same order as they were in the original file, but we encountered one occurrence (in our 25 files) in which the order of the variables was changed (from `A = 0 OR B = 0` to `B = 0 OR A = 0`). While these two conditions were indeed semantically equivalent, the current version of our condition parser did not detect them as such, although error recovery did allow us to consider the file as trace equivalent. Since this happens only a very small fraction of the time, we reported the behaviour to the company who explained that they indeed do not guarantee that variables will remain in the same order as they were before. We recommended them to restructure the refactoring rules to respect the original order in the future.

Finally, we noticed a limitation in our custom if-factorization implementation: we designed it to look for `IF`s that are strictly contiguous. However, we found cases where jumping structures (i.e. `NEXT SENTENCE`) were found in between the `IF`s. In that case, our implementation could not match the conditions. The error recovery algorithm allowed us to skip those nodes and show equivalence in some cases, but not in all of them.

10.3 Automated analysis

For our automated analysis, we ran our trace equivalence algorithm on all of our 3014 files, first without and then with the error recovery set to skip up to 20 nodes. For each file, we reported how many LTS links each version of the file had, how many of those were explored, how many were matched and if relevant how many were skipped by the error recovery.

Without the error recovery, only 139 out of the 3014 files were shown to be equivalent (before and after refactoring). The number of files found to be equivalent grows to 902 with error recovery. Unfortunately, this still means that we can show equivalence only in respectively 4.61% and 29,92% of the files. Keeping in line with our expectation that smaller files are easier to prove equivalent, the biggest file for which we matched equivalence without error recovery was 8782 LOC, and 19907 LOC with error recovery activated (both values are the size pre-refactoring).

Note that not all links from the pre-refactored version of the code need to be matched for the graphs to be trace-equivalent. Indeed, this first version of the code often contains dead code that is present in the model but is not reachable from anywhere and will therefore never be explored. Since the refactored version should not contain any dead code, we based our automated analysis on the percentage of links explored and matched in this post-refactored ver-

sion.

Figure 10.4 shows the amount of links explored for the files with (orange) and without (blue) error recovery activated. Activating error recovery increases the amount of links explored overall, as well as significantly increases the amount of links matched. The amount of links explored increased in 2843 cases (94,28%), and the number of matched nodes augmented in 2128 cases (70,64%). We also noted that the number of nodes that were skipped by the error recovery algorithm remained fairly low, meaning that when nodes are ignored, the algorithm is able to find its bearing fairly fast.

Finally, we can see that with the error recovery on, 912 or 30.21% of the files have more than 40% of their links shown as equivalent, giving an indication that the refactoring was behaviour-preserving.

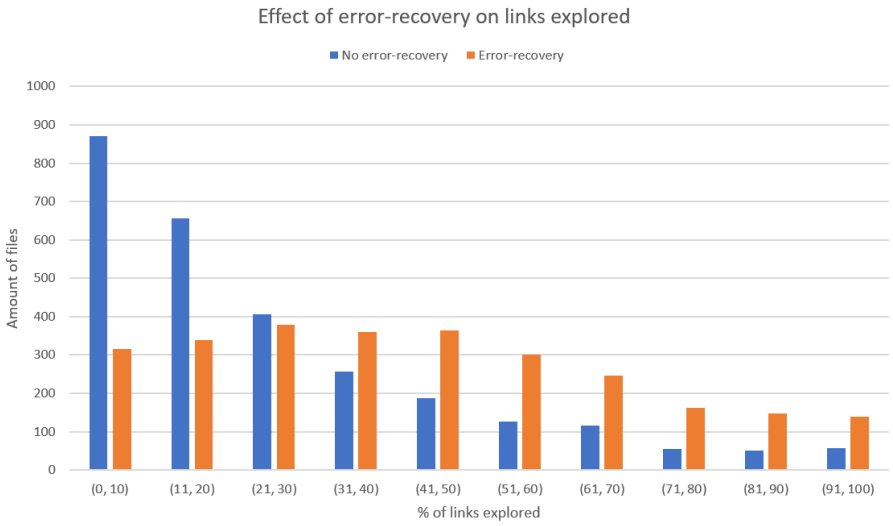


Figure 10.4: Percentage of links explored by the trace equivalence algorithm with and without error recovery

10.4 Threats to validity

Before turning our attention to the lessons learned from this study, let us first identify some of its threats to validity:

- Construct Validity: The use of LOC as a metric to measure the impact of refactoring might not be the best approach as it does not account for the complexity of the code nor of the quality of the code.
- Internal Validity: Our manual analysis of only 5 files in each partition

may not be representative of the entire dataset, leading to potential bias in the results.

- Internal Validity: The validation was performed by a researcher, playing the role of code owner. While the identity of the code owners is confidential information, the refactoring experts could have been a better choice to act as a proxy for their code owners. However, the researcher used their software engineering knowledge to assess equivalence of the refactored and original version of the code. The fact that this was possible using this technique on code completely unknown by the researcher points towards the fact that code owners should be able to benefit from it as well.
- External Validity: The results of this study may only be applicable to refactoring of COBOL programs and may not be generalisable to other programming languages or systems. However, we did pay special attention to making all parts of our tool as generic as possible.
- External Validity: The dataset used in this study is not representative of all COBOL programs, as it was obtained from a single client company, which may limit the generalisability of the results.

Finally, we would like to highlight some points of attention to keep in mind when designing and implementing such tools:

- *Carefully pick the language construct to extract*
  It is important to consider carefully what language elements need to be extracted from the source code in order to produce a useful output. Taking into account too few elements will be too abstract to be useful in an analysis, while extracting too many will make the trace equivalence harder to perform; both computationally and timewise. In our case, we used the expertise provided by Raincode Labs to pick the most relevant language constructs.
- *Understand the language semantics*
  When translating from CFG to LTS, one has to be careful to deeply understand the semantics of certain language constructs in order to craft graphs that correctly describe a program's behaviour. Consider for example COBOL's PERFORM statement. When it is executed, it first jumps to the specified paragraph(s), executes all of it (them), and then jumps back to the regular execution. Furthermore, when such a statement is found while executing code from another PERFORM, it creates an execution stack, meaning that all statements from the inner perform will be executed before going back to the initial one. However, GO TO

statements can take the execution out of range of the current execution context, eliminating the capacity of the PERFORM to go back to where it started.

- *Focus effort on what matters*
  A lot of the initial limitations of our trace equivalence algorithm were linked to our inability to parse the conditions. While the implementation of our condition parser was an extra effort specific to COBOL, it did improve our results greatly while not impeding our efficiency because of its linear time execution. Furthermore, the same principle could be applied to another language, even though it would require reimplementing the condition parser for that language.

10.5 Conclusion

In this chapter, we have learned that, while it is possible to show trace equivalence for some files, it will not be the case for all of them and should not be expected, hence the technique we presented is more suited to point out areas of concern in refactored code which should be tested more thoroughly before releasing it, rather than asking someone to blindly trust that the new code is behaviorally equivalent to the original one.

However, thanks to the error recovery, our algorithm does allow for having some gaps in the nodes that can be matched between two graphs and to find more overall matches. This mechanism allows for graphs to be matched partially, and those gaps are good points of entry to be looked into by the code owner to verify whether they are indeed equivalent as well. Moreso, by implementing additional special cases in the algorithm like we did in subsection 9.3.5, we could augment the amount of successful matches further.

There may always be cases that remain uncovered because of intricacies of the particular programming language or dialect considered, or because of particularities of the refactoring rules that were applied. We aimed to have a *generic* tool, as opposed to one that reimplements the semantics of each of the individual refactorings. This would defeat the purpose of the tool which is to provide guarantees that two versions of the code remain equivalent, regardless of how these versions were obtained. Furthermore, the implementation of many intricate special cases could introduce possible bugs in the implementation of the verification algorithm, render it more complex, and lowering our confidence in its capabilities.

We found that the trace equivalence algorithm excels when comparing structural changes, such as those shown on Figure 9.6. Even when the code or CFG of a program may look very different to a human before and after refactoring, and would be difficult to compare side by side or with a regular

differencing algorithm, they are shown to be equivalent upon the very first iteration of our trace equivalence algorithm. Cases containing more structural than logical changes are best for our tool, which is the case for the large-scale refactoring process we analysed. This can be generalised further to any use case where proving equivalence between two programs that have been heavily restructured could be useful, which should be applicable to other (automated) refactoring tools in other programming languages.

Visualisation of the trace equivalence

11

In this chapter, we present the goals that we set for the visualisation that we created and detail how we adapted the trace equivalence algorithm in order to create a basis for the visualisation tool. We then show how we implemented this visualisation tool, the limitations we found regarding the scale of our industrial data and the techniques we applied to overcome those limitations.

11.1 Requirements for the visualisation

A first pragmatic requirement we have for the visualisation tool is to make it using the same framework as the tool we presented in subsection 5.3.4. This presents the advantage of keeping both user interfaces similar and therefore easier to use for Raincode Labs users. Furthermore, it allowed us to get input from and consult Alexandre Bergel, an expert in using this visualisation framework, on the usability of our tool.

In the context of visualising trace equivalence, we have established two main goals of the visualisation tool. The first goal is to provide a comprehensive project overview, allowing users to see at a glance the status of all migrated files within a project. Given the size of the PACBASE migration projects handled by Raincode Labs, this overview is intended to be scalable and usable with thousands of files. To achieve this, we would like to show an itemised list of all migrated files, offering a summary of their status. This status summary serves to inform tool users about the extent to which the nodes of each file are matching, unmatching, skipped or remained ignored by the trace equivalence. The challenge is to organise this list in a way that facilitates the analysis for tool users. We imagined two approaches: either sorting the list by alphabetical order (allowing to easily find a specific file) or sorting by the amount of nodes matched (allowing to easily find the files that have the most or least amount of nodes matched). Both sorting methods will be implemented in the final tool.

In addition to the broader project overview, the second goal of the visuali-

sation tool is to provide users with a detailed visualisation for every analysed file. This detailed view will include the CFGs of both versions of the code, before and after the migration. The emphasis here is on making these visualisations informative and easy to interpret. To this end, we used colour coding to distinguish elements in the graphs, clearly indicating whether each node and link has been matched, unmatched, skipped, or ignored in the trace equivalence process (same colour scheme as the status summary). This colour-coding mechanism will enhance the comprehensibility of the visualisation, making it easier for tool users to identify where and why the code was considered unmatching.

In addition to the use of colours to convey information, this second goal requires to pay close attention to the graphical layout of the graphs, striving to emphasise key information effectively and hiding uninteresting nodes and links whenever possible. For this, we used the fusing algorithm that we presented in subsection 5.3.4. Finally, we wanted to make our visualisation interactive to further facilitate the exploration and analysis process. From our analysis of the graph, we found ourselves often looking at a node in one of the two CFGs and wondering where its equivalent was in the other. This is where we created the interactive features: it highlights corresponding nodes in the two graphs helping to compare what changed on a smaller scale. With all of the features we described, we aim to provide users with a powerful tool that offers both a high-level project overview and a granular file-level analysis, enhancing their ability to understand and act upon the results of the trace equivalence process.

11.2 Marking the models

As explained in subsection 9.2.2, the trace equivalence algorithm compares LTS models through their execution paths. In its basic version, it simply gives a binary answer: either the models are trace equivalent or they are not.

Since we want to create a visualisation allowing for more granularity, the algorithm needs to be modified. We decided to record every step of the trace equivalence algorithm by marking the transitions and states of the LTSs.

We start by marking the transitions between the states. Before the start of the trace equivalence, every transition is marked as "unexplored". Then, during the execution of the algorithm, we mark the transitions as follows:

- If both transitions currently considered have the same label or when a *INTERNAL* link is ignored, we mark the transitions as *matched*;
- If the two transitions currently considered have a different label but further matches can be found by skipping them with the backtracking algorithm, we mark them as *unsure*;

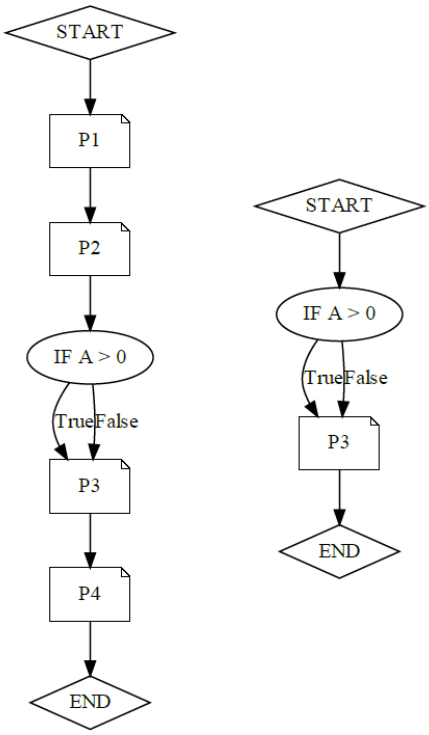


Figure 11.1: CFGs for a simple piece of code to compare.

- If both transitions currently considered have different labels and we can no longer backtrack, we mark them as *unmatched*.

Let us exemplify with a very small example. On Figure 11.1, we show the CFGs corresponding to two simple pieces of code with very few differences between them. All the links of Figure 11.2, which shows the LTSs corresponding to the CFGs of Figure 11.1, start out as unexplored. The marking of links follows the execution of the trace algorithm and goes as follows:

1. On the left side, the INTERNAL outgoing links from nodes L0, L1 and L2 are skipped and all marked as matched. On the right side, the INTERNAL outgoing link from node R0 is skipped and marked as matched.

2. Since node L3 and node R1 both have a link labelled `A > 0` and another labelled `NOT (A > 0)`, all four links are marked as matched.

3. Following the path of the labels `NOT (A > 0)`, on the left side, the *INTERNAL* links from nodes L4 and L5 are marked as matched. On the

right side, the *INTERNAL* link from node R2 is also marked as matched. We arrive at nodes L6 and R3 which are both end states, so the path is valid.

4. When the algorithm considers the path `A > 0`, it detects that the nodes L4 and R2 have already been explored together and the second path is also valid.

5. Since both paths are valid the graph is trace equivalent and all the transitions were marked as matched.

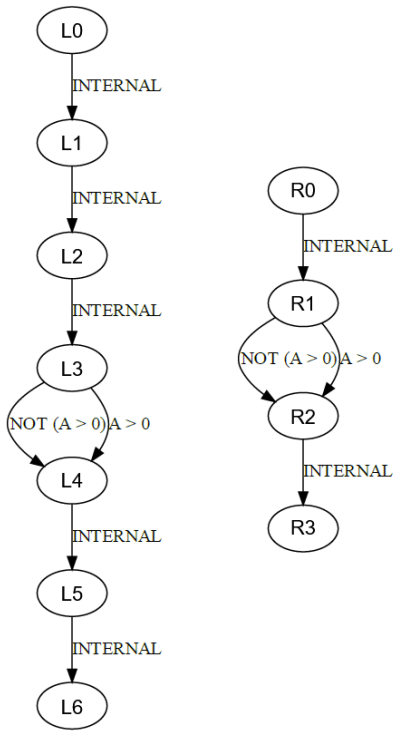


Figure 11.2: LTSs corresponding to the CFGs of Figure 11.1.

Note that this example is exempt from cycles for ease of comprehension. This is not the case with our industrial data, since its nature is far more complex. A detailed explanation of the inner workings of the algorithm on cycles from the real data would require several dozen of steps, rendering it overwhelming. We therefore only provide illustrations of examples containing cycles (see Figure 11.5). We decided to handle cycles as follows: when we meet a transition that has already been visited and marked, we simply add

more marks if necessary (i.e. if they are different). This means that any transition can be marked with all of the four "base values": unexplored, matched, unsure or matched, or a combinaison of the last three (the "unexplored" mark gets removed when the algorithm explores a node). The following list details all possible matches:

- unexplored
- matched
- unsure
- unmatched
- matched + unsure
- matched + unmatched
- unmatched + unsure
- matched + unsure + unmatched

This allows us the maximum precision to describe the execution of the trace equivalence in our visualisation.

To allow us to make our visualisation tool interactive, we also marked the states of the LTS models. When the trace equivalence explores execution paths, it does so in both models at the same time. In essence, it creates groups of nodes that are matched together on either model, i.e. nodes from the pre-refactoring LTS are matched with nodes on the post-refactoring LTS and vice versa. We identified both *perfect matches*, states that are exactly the same, like the `IF` from our example above and *group matches*. Group matches are nodes that have been explored together but cannot be linked specifically to a corresponding node in the other graph since their transitions are *INTERNAL*. Going back to the example of Figure 11.2, we get the following matching of states:

1. Nodes L0, L1 and L2 are all explored at once and therefore form a group. On the other side, the only node explored is R0. These are group matches.

2. Since node L3 and node R1 both have a link labeled `A > 0` and another labeled `NOT (A > 0)`, we stop to mark the state matches as being perfect matches. Node R1 is marked as the equivalent of node L3, and vice versa. Node R0 is marked as the equivalent of nodes L0, L1 and L2 and vice versa.

3. Following the path of the labels `NOT (A > 0)`, nodes L4, L5 and L6 form a group on the left side. On the right side, nodes R2 and R3 form a group. Since we reach the end of the graph, the nodes are marked as explained before, in a second group match.

4. Once again, the algorithm detects that the nodes L4 and R2 have already been explored together and stops there.

5. We now know that the graphs are trace equivalent, have marked all the transitions and grouped all the states.

Figure 11.3 shows the evolutions of the match groups and the transition marking for the example of Figure 11.2. All match groups start empty and all transitions unexplored (no icon), a green tick marks a link as matched.

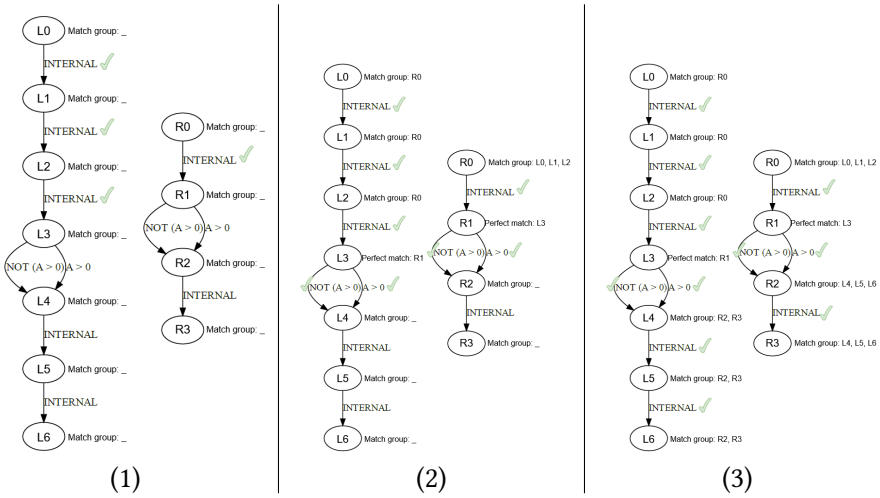


Figure 11.3: Matching of transitions and states on the example of Figure 11.2. Numbers (1), (2) and (3) correspond to the steps described in both numbered lists above

11.3 Visualisation tool

To present the results of the trace equivalence algorithm, we created a visualisation tool using Roassal [Ber16]. We chose to keep the technology and therefore style of visualisation consistent with the tool that we created for the log-based behavioural differencing presented in subsection 5.3.4.

The goal of this visualisation tool is to present the result from the trace equivalence algorithm and make it easy to interpret. We achieve this by presenting a clear colour code, as well as using our CFGs models, a format that

is familiar to developers. Our tool is composed of two views: the menu and the detailed view. The menu shows an overview of the entire refactoring project and is shown in Figure 11.4. On the left-hand side are two fields used to specify the files that will be shown (one points to a directory containing the COBOL files, the other to the output directory from our trace equivalence algorithm). Clicking on the *visualise* button will populate the right-hand side of the view. This panel also allows the user to order the files displayed on the right: either alphabetically or by most or least transitions marked as matched.

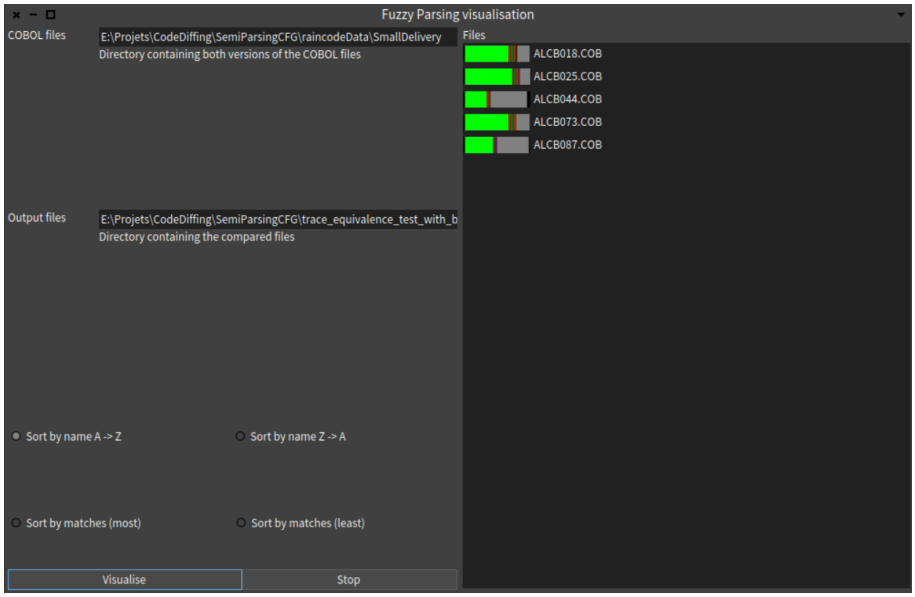


Figure 11.4: Menu of the trace equivalence visualisation tool.

Since transitions can have multiple markings, we introduced colours beyond the usual red/orange/green of common differencing tools. Our colour scheme works as follows:

- *Unexplored* nodes and links are in grey
- *Matched* nodes and links are in green
- *Unsure* nodes and links are in orange
- *Unmatched* nodes and links are in red
- *Matched + unmatched* nodes and links are in dark green
- *Matched + unsure* nodes and links are in brown
- *Unmatched + unsure* nodes and links are in purple
- *Matched + unsure + unmatched* nodes are in black

Note that a node is always the colour of its outgoing links. On the right-hand side of the menu window are the files from the refactoring process. We display both the file name and a summary of the result of the trace equivalence. We calculate the percentages of every match type over both models and display a bar divided into areas of colours corresponding to their amount in both models, using the colour code described above. This summary, along with the available sorting buttons, helps users quickly find what is important to them.

To obtain the second view of our tool, the detailed view, the user simply selects the desired file from the list on the right. This opens a new window showing the two CFGs corresponding to the file pre- and post-refactoring side-by-side (pre-refactoring on the left, post-refactoring on the right), with colour-coding as explained above. Figure 11.5 shows the result of applying the trace equivalence algorithm to a portion of a file being refactored, in a simple case where the models are trace equivalent. We added a *start* node, shaped differently than the others (triangle) to help our users find where to begin their analysis. To avoid clutter, we also denoted nodes that can result in the end of the execution with a thick black border (instead of a link towards an *end* node).

When presented with a detailed view, a user can interact with it by hovering the mouse on the different nodes. In the case of a node, it shows its label, e.g. *PERFORM F94FT*. In the case of a link, it reminds the user of the meaning of its colour (in our previously shown example, all link tooltip would show *matched*). It also shows the match group for the node, as explained in section 11.2. If nodes are highlighted in light blue, it means they do not have an

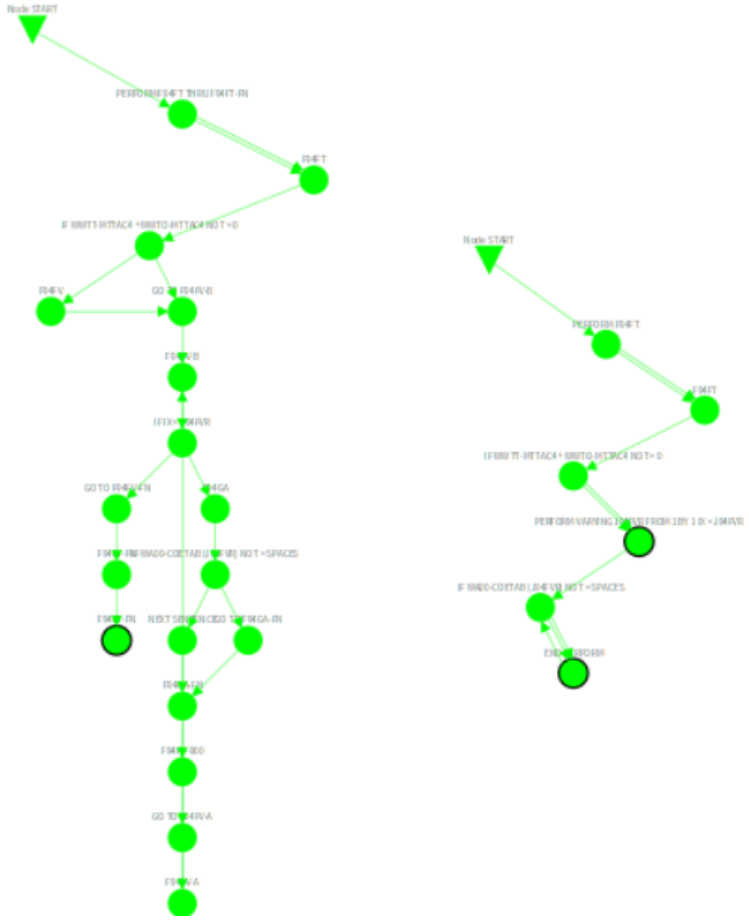


Figure 11.5: Detailed view on the refactoring of a portion of COBOL code.

exact equivalent in the other graph (group match). If they are highlighted in dark blue, they are a perfect match to each other. Examples of group match (left) and perfect match (right) are shown on Figure 11.6. Note that perfect matches are not always one-to-one since the trace equivalence algorithm detects both if factorization and if-to-evaluate transformations.

The example that we have shown so far is fairly small in terms of size: 40 LOC and no more than 20 nodes for the larger pre-refactoring CFG. While it helps in understanding the visualisation tool, it also shows that we will encounter scaling issues when looking at detailed views of entire files. This is indeed the case, as shown in Figure 11.7 (we apologise for the size of the picture). Here, our goal is to give an intuition of the scale of the full-size graphs, not to allow the reader to analyse them).

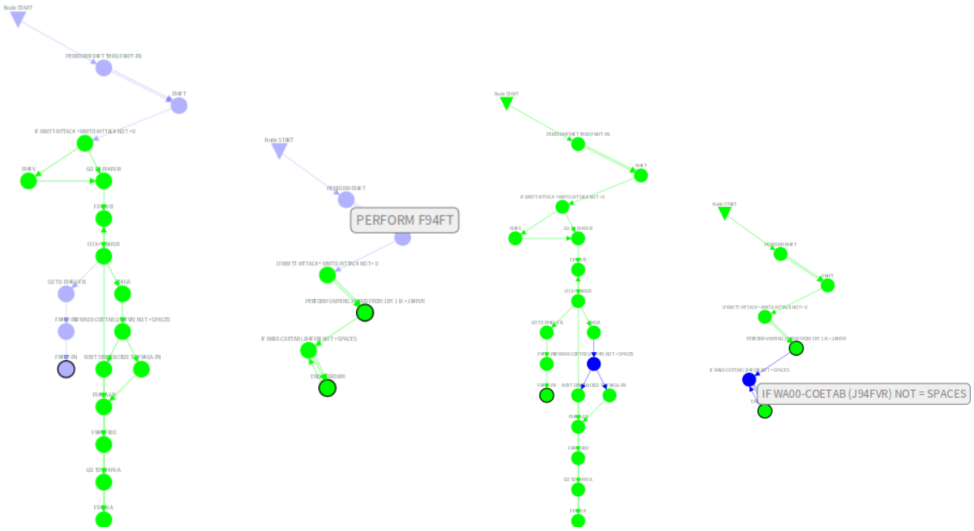


Figure 11.6: Highlighting of a group match (left) and perfect match (right) on the example from Figure 11.5

When analysing our results, we noticed that we had graphs of similar shapes as the ones generated by the log-based behavioural differencing in section 5.3. Due to the nature of COBOL and its construction based on paragraphs, we once again see long lines of nodes that are all matched and clutter the view, distracting from the important information. An example of this (zoomed in to be readable) is shown on Figure 11.8.

To solve that issue, we applied the same fusing algorithm that we described in subsection 5.3.4 and obtained similarly satisfying results, as shown in Figure 11.9 (pre-refactoring) and Figure 11.10 (post-refactoring) which show the fused version of both sides Figure 11.7.

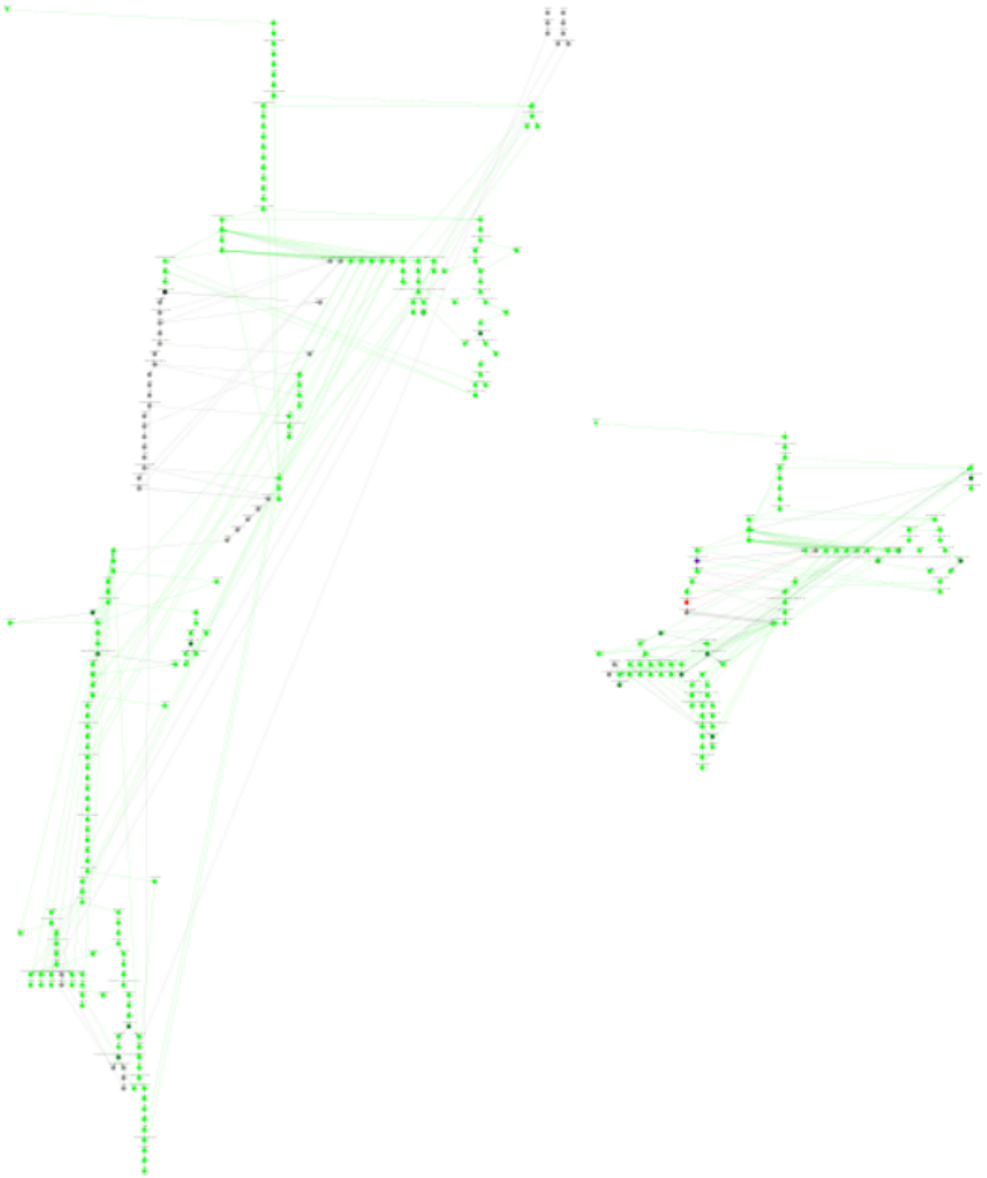


Figure 11.7: Detailed view on file ALCB025, no fusing

Figure 11.8: A succession of matched labels from file ALCB025

We also implemented buttons allowing users to show or hide the dead code present in the pre-refactoring version of the code, which added further clutter to the graph. The dead code-free version of Figure 11.9 can be seen on Figure 11.11.

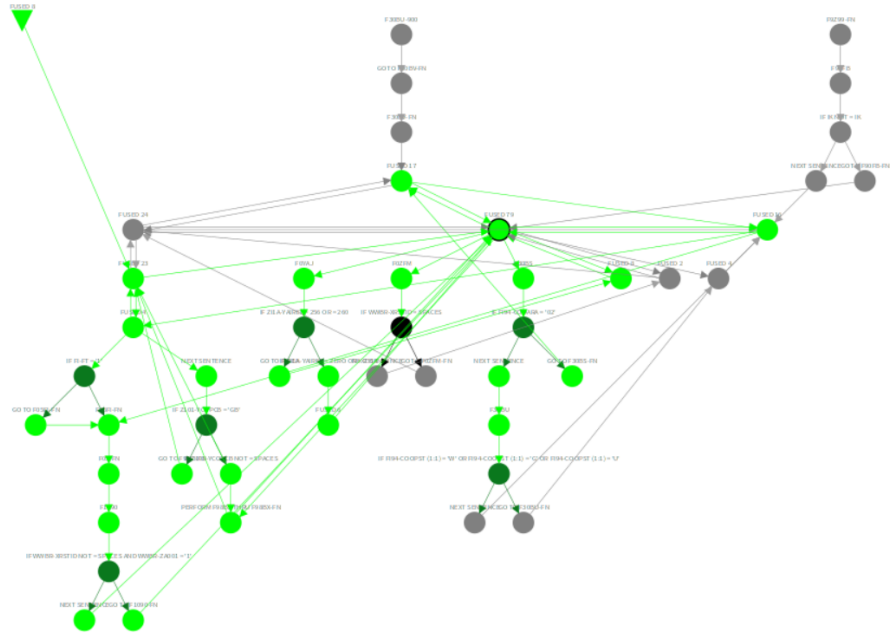


Figure 11.9: Detailed view on the pre-refactoring version of file ALCB025, with fusing

The goal of this tool is to be easy to use with a simple interface and well-known colour-coding, as well as helping a refactoring expert or the code owner to find points of interest in the code. Since we include the labels in the graphs, and since labels (paragraph names) in COBOL code are unique, it is very easy to find the exact place in the code where we detected a mismatch, and start a deeper analysis from there — e.g., by inspecting two non-matching conditions. Moreover, for the visualisation we chose to go back to the form of CFGs and not the LTSs that we use to perform the trace equivalence since CFGs are designed to be read by developers and are more natural for them to interpret.

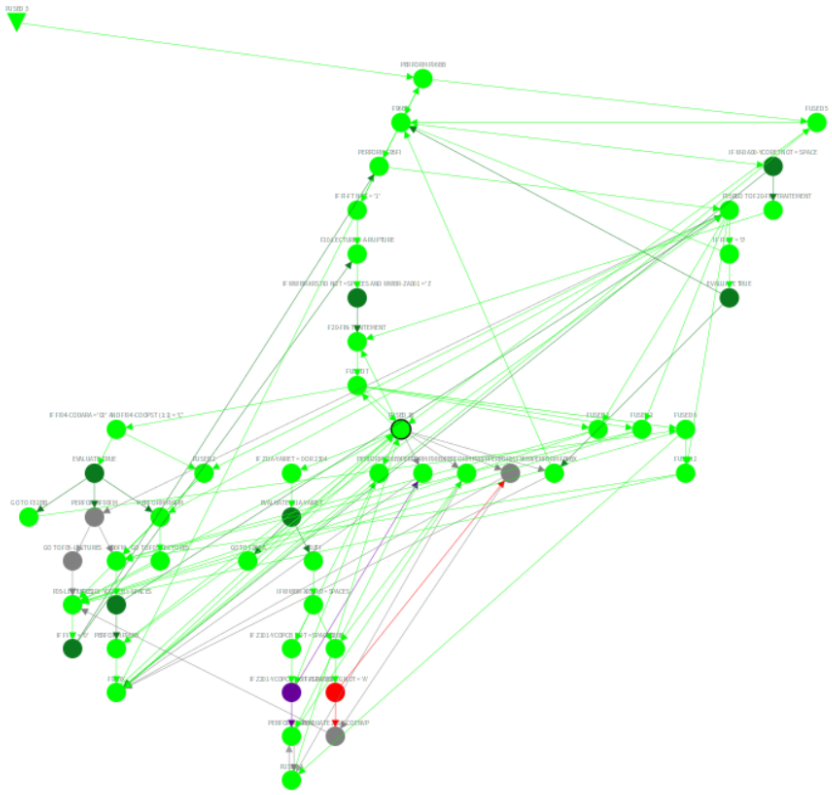


Figure 11.10: Detailed view on the post-refactoring version of file ALCB025, with fusing

11.4 Conclusion

In this chapter, we explained how we modified the trace equivalence algorithm in order to mark the LTS models and visualise the parts of those models that can be shown to be equivalent and the ones for which it is not possible with our algorithm. We have shown how we implemented a visualisation tool using Roassal, making use of colours and interactions to help convey the information to users. In the next chapter, we will describe how we validated the tool presented in this chapter.

Figure 11.11: Detailed view on the pre-refactoring version of file ALCB025, with fusing and dead code hidden

Validation of the trace equivalence visualisation tool

12

In this chapter, we first describe the methodology that we used to validate the trace equivalence visualisation tool that we created, enlisting the help of Raincode Labs experts. We paid attention to validate that the tool is understandable and usable, that the interactions it offers are satisfactory and that the fusing algorithm we applied helps with scaling, while not hindering the comprehension of the graph. Finally, we discuss the satisfaction of our users as well as explore other use cases for this tool within Raincode Labs.

12.1 Interview set up

In this section, we detail how we conducted the validation interviews. We start by explaining the preamble and three phases of the interview and we will describe the results in the next sections. As was mentioned before in section 6.2, only two Raincode Labs employees are involved in PACBASE-migration projects and participated to this validation. In this section, we will once again refer to them as participants P_C and P_T, respectively for customer- and technical-oriented.

Note that we performed the two rounds of validation during two different weeks. This allowed us to implement feedback from the first interview with P_C and present it to our second participant, P_T.

12.1.1 Preamble

Before showing the visualisation tool to our participants, we took a few minutes to contextualise how it came to be. We started by listing the code structures that we extracted from the COBOL code, reminding them that they were decided upon in a previous meeting. We then explained that we extracted them from the code using a semi-parser and arranged the structures into Control-Flow Graphs (a model familiar to both participants). Finally, we

told them that we transformed the CFGs into LTSs and compared them using trace equivalence, giving an intuition on how the algorithm works with execution paths.

We explained that they were going to interact with a visualisation tool showing the result of the trace equivalence and prompted them to look at a user manual for the tool (found in Appendix C). This manual focuses on the meaning of shapes and colours shown in the graph, detailing the *base colours* (grey, green, orange and red), the *compounded colours* (dark green, brown, purple and black) as well as the highlighting (light and dark blue for group and perfect matches).

Finally, we asked them to "think aloud" during their use of the visualisation, in order for us to get the most possible feedback from the interview. We recorded both the screen of the laptop used along with the sound of our conversation.

12.1.2 First phase – Tutorial

In the first phase of the interview, the participants were presented with five small handcrafted example files and the fusing algorithm was turned off in the visualisation tool. They were instructed to open them one after another, analyse them in order to familiarise themselves with the tool and ask questions whenever something was unclear to them.

The examples presented all of the *base colours*, showing an example of a file fully matched, one where a node was marked unsure and one where there were unmatched and unexplored nodes. They were then shown an example similar to Figure 11.1, familiarising them with group and perfect matches.

Finally, they were shown Figure 11.5 and asked to manually analyse it using the interactions, explaining why everything was matching. After this, we concluded the first phase by explaining that if we were to keep to tool configured as it was, we would not be able to analyse full files, and explained that we were going to turn on the fusing algorithm for the rest of the interview.

12.1.3 Second phase – Unsupervised exploration

After activating the fusing algorithm, the participants were presented with the five first files from a migration project and asked to explore them and narrate their experience, again asking questions whenever they felt it was necessary.

In this phase, we let them analyse without any directions from us until they expressed that they had seen enough and had nothing more to say.

12.1.4 Third phase – Closing questions

We concluded the interviews by going over a set of pre-defined questions that we will detail in the next section. Finally, we asked them if they had any further remarks or feedback to give.

12.2 Methodology

To structure the presentation of the results of our user validation, we once again take inspiration from the work of Sillito et al. [SMD08]. We used the vocal description of the actions performed by our participants to analyse not only the answers they gave to our specific questions but also the questions that arose from them. We can split our observations into four categories:

- Questions about the **domain** (input, output, inner representation of data in the tool);
- Questions about **technical execution** (linked to the way we extracted things from the COBOL file or how we computed the execution paths);
- Questions about the **meaning** of a visual component (they either asked us directly or used the user manual to look for the answer);
- Questions leading to ideas for **future work** (when they asked if some feature would be possible to implement).

This methodology allows us to get the most feedback, both in terms of quality and quantity and compensates for our low number of participants in the validation.

12.3 Results of the validation

In this section, we present the questions that arose during the interview as well as the ones that we asked our participants. We present the answers to those questions in the next section.

Domain

We had a very low amount of domain questions, P_C being the only one to ask during phase 2: *"Are you sure the graphs are not flipped? Since the right graph is bigger than the left, I would expect that one to be the pre-refactoring"*

Technical execution

While P_C seemed satisfied with the technical information that we gave in the preamble of the interview, P_T had two technical questions. One refers to the constructs that we extracted from the COBOL: *"Do you handle the periods?"* and the other was about our handling of the way COBOL works: *"Can you*

deal with the fact that a perform comes back, but only if the execution does not jump outside of its boundaries?".

Meaning

P_C had a question regarding the meaning of the large border denoting and ending node : *"Does this mean that the execution always stops there, or that it can stop there?"*. P_T felt overwhelmed when reading the user manual and when first interacting with the full graphs. He had to refer to the manual a few times to remember the meaning of colours, although he did not ask us a direct question.

Future work

P_C had many ideas and questions on how he could use the tool in different contexts. We retain the two most significant: *"Could we use this tool at a smaller scale so it can be less overwhelming to analyse its output?"* and *"What kinds of results would we get if we were to apply it to the migrated code before and after a redelivery?"*. P_T was more focused on how we could tailor the tool to improve the amount of matches, asking: *"Why not implement a few more of the specifics of our process in order to not have so many things flagged as unmatched?"*

Open questions

We also had a set of questions allowing us to validate specific aspects of our tool:

1. We asked if the colours we chose were easy to differentiate visually as well as understandable.
2. We wanted to make sure that our choice of highlighting group and perfect match was accessible.
3. We asked if anything was missing from the main menu and if the summary of each file was a useful feature.
4. We asked if the layout of the graphs was satisfactory and if the amount of information shown on screen was not overwhelming.
5. We asked both participants if they felt it would be appropriate to show the visualisation tool to their clients.
6. Finally, we asked if the participants were surprised by the results they were shown.

12.4 Analysis of the results

Domain

We explain our very low amount of domain questions by two factors: first, our

validation participants are experts in their field and therefore are very familiar with the domain, they had no trouble comprehending any of the extracted structures, nor with our chosen format of control-flow graphs. Second, the fact that we took the time to contextualise how we obtained the result that was presented, probably helped answer most questions regarding the input or output of our tool before they happened.

P_C's confusion about which graph corresponds to which version of the program is easily answered: while it is true that the amount of nodes found in the CFG before refactoring is always larger than the amount of nodes in the CFG after refactoring, this is no longer guaranteed in the visualisation, when the fusing algorithm is applied. To prove our point, we showed P_C the same graphs without fusing, which clarified things for him. We asked if this was a downside of the visualisation for him, and he answered *"No, now that I understand that it is because some of the nodes are hidden, I am no longer confused"*.

Technical execution

The respective client and technical orientations of our participants help explain why only P_T had technical questions. When P_C looked at our visualisation, he saw exactly what he expected and assumed that we had done our due diligence. For P_T, since he has been working on the automated refactoring process, he is deeply aware of the challenges of COBOL and wanted to make sure that we did as well so he could trust our tool.

We answered his question of *"Do you handle the periods?"* by explaining that yes, we do extract every period from the COBOL files, but they "disappear" from our visualisation as their only use is to serve as an anchor for the links in our models. This answer satisfied him, as he confirmed that *"Yes, it makes sense to not show them as individual nodes then"*.

When he asked *"Can you deal with the fact that a perform comes back, but only if the execution does not jump outside of its boundaries?"*, we explained the strategy that we implemented to handle the `PERFORM` statements, telling him that we based it directly on the COBOL documentation. Again, he was satisfied with this answer and from this point onwards, trusted that our tool did correctly reflect the behaviour of COBOL.

Meaning

The only question we received regarding the meaning of the things on screen concerned the end nodes: since several nodes can lead to an ending state, we chose to remove the dedicated end node and instead change the border of all nodes which could result in an ending state to reduce clutter in the graph. This was slightly confusing for P_C who asked if a black border meant that the execution *always* ended or if it meant that it *could* end. We explained that, if the node had no outgoing link, it meant that the execution always ended while if the node had (at least) one outgoing link, it meant that the execution

could end there. This answer satisfied P_C and could be included in a new iteration of the user manual to avoid all confusion.

While P_T said he felt overwhelmed at first by the amount of colours and nodes in the full graphs, he did not need our help to get over the issue. Furthermore, he felt more comfortable after a few minutes and did not express further confusion.

Future work

P_C's questions about future work lead to several ideas on how the tool could be applied in different contexts. First, he told us that he expected the output to be more accessible if we focused not on the entire automated refactoring process, but on the effects of a single migration rule. It is possible for Raincode Labs to run the process with a very limited amount of rules activated, and P_C thinks this visualisation could help convince a client that a specific rule will not alter the behaviour of the code. Furthermore, it could be used in the context of the extra analysis performed when a redelivery triggers a new rule (see subsubsection 4.3.2.4). Since it is already part of the process to generate the version just before and just after the application of the new rule, those two versions could be used as inputs. The ability to say that an automated process showed the two versions to be equivalent would strongly benefit P_C's interactions with clients.

Second, he asked us if it would be possible to analyse the code not at the level of an entire file, but at a smaller scale, for example focusing on a few paragraphs. He felt that this would make the analysis easier since it would greatly reduce the amount of information shown at once. We informed him that this would require some tweaking of our fuzzy parser but would be easy to achieve and he said that it would be a useful feature to have.

Finally, P_C wondered what kinds of results we would get if we compared not the generated code to its refactored version, but two refactored versions of the code together, in the context of a redelivery. P_C's expectations were that this would lead to a higher percentage of nodes matched thanks to our backtracking algorithm. Since we had the data available to us, we ran the experiment and found it to have mixed results: some files are shown to be equivalent with a few nodes marked as unsure, corresponding to features added in between the delivery and redelivery, while others have changed too much and render the comparison more confusing than it is helpful.

Nevertheless, we showed the results of this experiment P_T who was interested. He said that he was not surprised that comparing the two refactored versions as we did had mixed results. However, he had an idea to go further in that direction. He would perform a meta-comparison, by comparing the pre-refactoring versions and the post-refactoring version separately, then looking at both comparisons to see if we find similar amounts and kinds of changes in them. This approach was also tested: we see the expected results,

but it requires more manual analysis. We do not have a meta-comparison tool allowing us to compare the result of the visualisation for both versions, requiring us to manually switch between the two views.

During his interview, P_T conducted a detailed analysis of a node flagged as unmatched. We believe this to be a perfect example on how our tool could be useful, and therefore detail the process (we took screenshots of the recorded interview to do so). His analysis started in our tool. He opened the first file, scrolled to a red (unmatched) node on the pre-refactoring side and decided that he intended to understand why it was marked as not matching. The node in question, `IF SQLCODE = -803` can be seen in Figure 12.1.

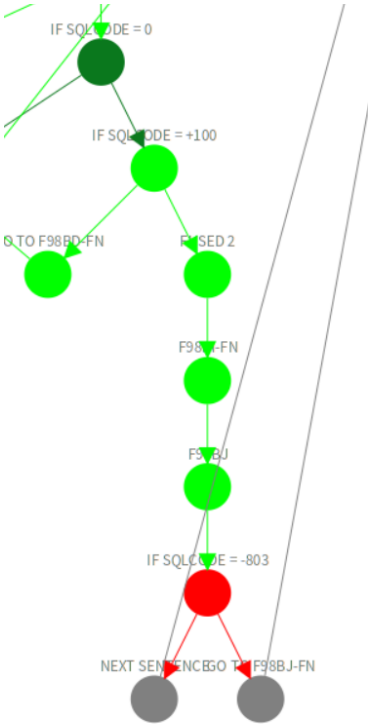


Figure 12.1: The chosen node and its surrounding

To continue, he hovered his mouse on every node linked to the red node, looking for the first perfect match. There is two labels and a fused node (containing more labels), then another if, `IF SQLCODE = +100`. It is a perfect match (with the `IF SQLCODE = 0` above) to an `EVALUATE SQLCODE` on the post-refactoring side, as can be seen on Figure 12.2.

This information was enough for P_T to move to looking at the code. By using the find feature of the text editor, he started by confirming that the `IF SQLCODE = -803` was indeed not in the post-refactored version. Once

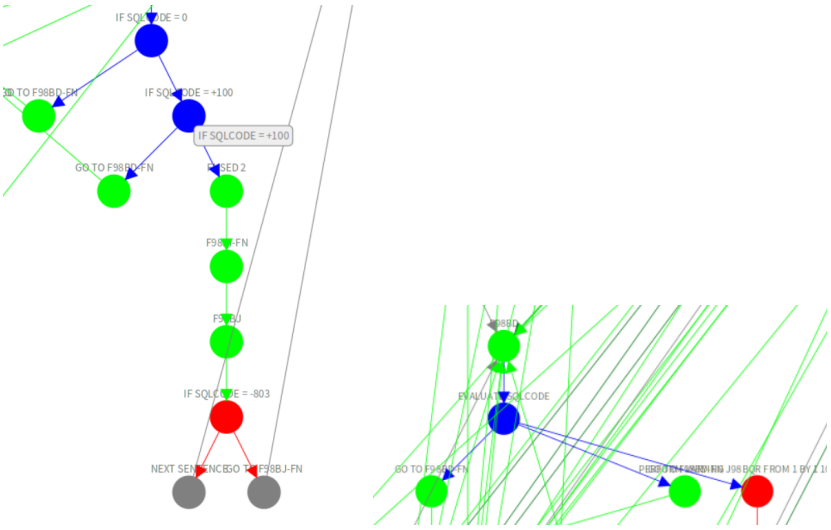


Figure 12.2: Perfect match in the pre- (left) and post-refactoring (right) versions of the graphs

he did, he looked for the `IF SQLCODE = 0` and `IF SQLCODE = +100` in the pre-refactored version, as well as the `EVALUATE SQLCODE` in the post-refactored code. We show an extract of the pre-refactored version of the code on Listing 12.1 and its post-refactored version equivalent on Listing 12.2.

Listing 12.1: Pre-refactoring version of the code analysed (PROGRAM-IDs were removed to ease the comprehension)

```
...
014260  IF SQLCODE = 0
014270  MOVE 0 TO IK
014280  GO TO F98BD-FN.
014290  IF SQLCODE = +100
014300  MOVE 1 TO IK
014310  GO TO F98BD-FN.
014320  MOVE 2 TO IK.
014330  F98BF-FN. EXIT.
014340*N98BI. NOTE *INTERCEPTION SQLCODE NEGATIF *.
014350  F98BI. EXIT.
014360  F98BI-FN. EXIT.
014370*N98BJ. NOTE *ERREUR SUR INSERTION *.
014380  F98BJ. IF SQLCODE = -803
014390  NEXT SENTENCE ELSE GO TO F98BJ-FN.
014400  F98BJ-FN. EXIT.
...
```

Listing 12.2: Post-refactoring version of the code analysed

```
1  ...
2       F98BD.
3          EVALUATE SQLCODE
4       *-----------------------------------------
5       *N98BF. FIN DE LECTURE
6       *-----------------------------------------
7            WHEN 0
8              MOVE 0 TO IK
9              GO TO F98BD-FN
10           WHEN 100
11             MOVE 1 TO IK
12             GO TO F98BD-FN
13           WHEN OTHER
14             CONTINUE
15         END-EVALUATE
16         MOVE 2 TO IK
17         DISPLAY '--------------------------'
18 ...
```

When looking at the code, P_T started by confirming he was at the correct place, which he was able to do by referring to the label names shown in our tool. In the graph, he identified labels F98BJ (pre-refactoring) and F98BD (post-refactoring), then confirmed their presence in the code. Once he was certain he was looking at the correct code snippet, P_T started his analysis. He came to the following conclusion: the refactoring process completely deleted the if statement because it was *doing nothing* (his wording). This entire process, from picking the node to analyse to opening the files and coming up with the answer, took him around five minutes.

Indeed, we can see that the statement IF SQLCODE = -803 (line 14 and 15 of Listing 12.1) only contains jumps: a NEXT SENTENCE and a GO TO F98BJ-FN. Both jumps lead the execution to the same point (the very end of line 15 for the first one and line 16 for the second), meaning that the computation of the statement did not, in fact, perform anything useful. This is why there is no IF SQLCODE = -803 in the refactored version of the code. Instead, we can see the result of refactoring both IF SQLCODE = 0 and IF SQLCODE = +100 into the EVALUATE SQLCODE of line 3. The WHEN 0 of line 7 corresponding to the body of the first if, and the WHEN 100 of line 10 to the second if. After this (line 16), we see the same MOVE statement as on line 8 of Listing 12.1, then the code continues further with the DISPLAY (not shown in the code of Listing 12.1, but present several lines after).

After his analysis, P_T asked why our tool was flagging this as unmatching. We explained that, due to the partial nature of our semi-parser, we have no way to know that the if statement did not contain any other statement than the jumps, which is why we flag it. He then asked us *"Why not implement*

a few more or the specifics of our process in order to not have so many things flagged as unmached?" and we explained our reasoning for not wanting to implement all the migration rules in our tool to avoid bias. He was not fully satisfied with this answer, arguing that he would be interested in seeing at least this case implemented and the effect it would have on the output since he feels this specific situation will be very frequent.

Open questions

1: Are the colours clear and visible? Are there too many?

P_C found that the colours were clear and easy to distinguish, he intuitively understood that we derived the compounded colours from the base colours and assured us that the amount of colour was acceptable. P_T was more hesitant when reading the user manual but said that after a few minutes with the tool, he was able to navigate and interpret easily.

2: Is the highlighting clear? Is it helpful?

Both participants actively took advantage of the highlighting and stated that it was easy to understand and helped their analysis.

3: Is the menu clear? Is it missing any feature? Is the summary for every file useful?

Both participants found the menu clear and easy to use. P_C suggested that we add the option of sorting the files by amount of matches rather than only alphabetically, which we implemented before the interview with P_T. We asked him directly about it, and he validated that it was indeed useful. Both participants appreciated the summary of every file, relying on it to pick the files they wanted to analyse.

4: Is the layout satisfactory? Is there too much information displayed at once?

Both participants appreciated the tree-like layout and said it was familiar and therefore useful. Both agreed that there was a lot of information displayed on the screen, but that this was unavoidable due to the nature of our data and that the fusing algorithm helped make the graphs usable. P_C also stated that these limitations always happen to him when he wants to visualise graphs from industrial legacy code.

5: Would you show (part of) this to a client?

Both participants said that the amount of nodes unexplored makes them wary of showing the graphs to clients. They fear that the fact that there are anywhere from 10 to 40% of the nodes that are not explored will stress the client even more. However, P_C would consider using this at a smaller scale as discussed in the future work section.

6: Are you surprised of the result?

In this case, our participants disagree. P_C is not surprised by the results, he said *"I know how complex our process is and I did not expect that it would be possible to show trace equivalence for the entirety of the programs"*. P_T expected us to include more of the specificities of the automated refactoring in our

tool since they feel so natural to him. He therefore expected our number of matches to be higher and stated that he thinks implementing more special cases would not introduce too much bias. As long as we do not copy the code he wrote for the refactoring, he feels that if two different people arrive at the same result, it is a form of proof.

12.5 Discussion

We observed that the visualisation tool was easy to learn and use for our participants. Neither felt the need to zoom or scroll much in the graphs nor did they reposition any nodes. Instead, they made use of the interactivity and expressed very little confusion. This suggests that the tool is well-designed and that the first phase of the interview where we allowed them to familiarise themselves with the interface did improve their experience.

Interestingly, P_T needed proof that our knowledge of the inner workings of COBOL was sufficient before he could trust our tool. This highlights that it is difficult for someone to trust a tool when they do not know how it works. In this case, much like with clients of the automated refactoring process, showing how we did things and how we documented ourselves helped alleviate the scepticism of P_T.

We have seen that the tool does not show trace equivalence for every node in the graphs and is probably not suited for clients in its current form. However, it did allow P_T to quickly find lines of code flagged as unmatched in both pre- and post-refactoring versions of a file and to demonstrate why the flagged execution would not cause any bug for the client. This proves that the tool can be used efficiently to pinpoint points of interest in the code and helps support the more in-depth analysis that is then necessary.

The tool could also be an asset when the engineers perform a redelivery and a new migration rule gets triggered. This situation was highlighted in our discussion as causing the most stress to clients, as they did not have the opportunity to test the new rule. Having a way to show them that the version before and after the rule is applied would greatly help alleviate any doubt towards the redelivered code.

Furthermore, both participants see ways to improve and make use of the visualisation tool. Using it with fewer migration rules or on specific parts of programs could help guide clients in their testing or further help them comprehend the effect of a specific migration rule.

While implementing more special cases must be done with caution, we believe that it is possible to do so without making the tool biased, at least for the case described above. For example, we could tweak our semi-parser to denote when it ignores part of the code, indicating the presence of some statements that it did not parse. With this, we could detect that the removal

of the if statement analysed by P_T is indeed behaviour-preserving. An implementation of this is a good avenue for future work with this tool, as would be a deeper analysis (helped by P_T) of the migration rules to identify other candidates for special cases.

12.6 Conclusion

In this chapter, we presented our methodology to validate the tool that we described in chapter 11. We conducted semi-formal interviews with two experts from Raincode Labs, asking their opinions on the usability of the visualisation tool both in terms of ease of use and versatility. Finally, we presented our findings and discussed their implications.

This validation lets us conclude that the tool could be used as-is when presenting rules to clients to enforce their trust at the beginning of the project or in the context of a redelivery. The tool could also be extended to be able to analyse parts of a file that are critical or less understood by clients. Another opportunity would be to explore the implementation of further special cases that would improve the amount of matches, although we should remain cautious about not implementing the logic from the automated refactoring process into the tool to keep our approach unbiased.

13 Conclusion

We started this thesis by introducing our partner company, Raincode Labs, and outlined its use case of automated large-scale refactoring of legacy code. We set out to help them explain two essential aspects of their project: *how* does it work and *why* can it be trusted. This lead us to our main research question: *how can we help code owners and refactoring engineers understand the process and results of automated refactoring on large legacy code bases?* We further refined this question into five sub-questions, all of which are answered by the different contributions of this thesis.

The first part of this thesis tackles the first fundamental need of Raincode Labs experts: explaining *how* the automated refactoring process functions. This is translated into the first sub-question that we asked ourselves, *how can we provide insights as to how a large-scale process reacts to changes?* We present our log-based behavioural differencing tool as an answer to this first question. We found that extracting models from the logs of a process is a powerful tool to understand it. Using our tool, the experts were able to quickly perform an analysis of the changes depicted. Moreover, they easily imagined other scenarios in which the tool could be useful to them.

Nonetheless, we were immediately confronted with another of the sub-questions: *how do we manage to scale to the size of a real industrial use case?* The scale of the industrial data at hand rendered the visualisation proposed in the original approach ineffective. While visualisation tools are indeed powerful, we learned that it is essential to identify the information that needs to be shown and that can be hidden to alleviate clutter. It might also be necessary to analyse only part of the process, or to divide it into manageable portions.

The third part of this thesis answers the second fundamental question: *why* can the automated refactoring process be trusted. The first step in answering this question is to better understand what the process is doing. To do so, we asked ourselves *how can we extract from the code the structural information impacted by the automated refactoring, and do so in a language-agnostic way?* Raincode Labs experts helped us in identifying the structures most impacted by their process. To extract the structures, we proposed a semi-parsing tool. This tool is very flexible and easy to configure, which makes it COBOL dialect agnostic and allows to handle other languages that follow the

paradigm of imperative programming with minimal effort. It is also able to parse partial or uncompilable code, a situation that is common for Raincode Labs. However, it would require more optimisation to be as time-efficient as industrial-grade full parsers when used on large files.

Due to the scale of the refactoring projects of Raincode Labs, it is impossible to perform a manual analysis of all the files, creating the need for a tool that performs an automated comparison. We therefore asked ourselves *what models do we need in order to compare them and show that they are equivalent?* We proposed the use of both CFGs and LTSs, that allows us to create both an automated comparison and a clear visualisation tool. For the comparison, we propose to use trace equivalence. We found the base version of the algorithm to excel at showing that changes in the structure of the programs are behaviour-preserving, but to be limited when handling changes in conditions. We proposed several solutions to the limitations that we found, but ultimately conclude that trace equivalence does not suffice to show that the refactoring is behaviour-preserving when applied to an entire PACBASE migration. However, we do show that the visualisation tool we created allows a Raincode Labs expert to efficiently identify why a piece of code flagged as non-equivalent is not at risk of creating bugs. Moreso, the validation participants imagined other use cases for our tool.

For each of the three techniques and two visualisation tools presented in this work, we ensure that they are adequate through qualitative and quantitative validations. This lead us to our last sub-question: *how can the tools produced in this thesis be used, in which contexts within and outside of the company?* In this text, we paid close attention to explaining our design choices as well as contextualising our work. This will allow other researchers to replicate our findings and apply them outside of the use cases provided by Raincode Labs. We have shown that Raincode Labs experts have used our tools during the validations and were able to imagine different uses for them. Our choice of framework for the visualisation tools was partially influenced by the fact that knowledge of its programming language exists within the company. Our tools are open-source, and therefore available to anyone, inside or outside of Raincode Labs. As we mentioned, our log-based differencing tool helps to answer questions that are asked by people analysing logs in various contexts, from Raincode Labs to embedded systems, showing how generic the problem of log analysis is. Our semi-parser is already able to handle any of the 400 COBOL dialects, and would be easily extendable to other legacy languages like FORTRAN by modifying its configuration file. Finally, the trace equivalence algorithm we presented was designed to be as language-agnostic as possible, relying mostly on the model it takes as input to reflect the behaviour of the language it analyses.

Both approaches presented in this thesis are complementary and will help

Raincode Labs answer the questions frequently asked by their clients. When the need arises to argue why they should pick a configuration over another, the log-based visualisation will help show them how the refactoring process evolves with the two different configurations. When the time comes to persuade clients that the result of the migration is behaviour-preserving, the trace equivalence will be able to sift through all of the files and flag the ones that are most critical to analyse. Finally, the trace-equivalence visualisation will help guide the client to the lines of code that cannot be shown to be trace equivalent and therefore should be tested in priority.

We also found that helping the understanding of such a large-scale refactoring process will have unforeseen benefits for Raincode Labs. The log-based visualisation can help the task of maintaining the refactoring process, allowing engineers to ensure that the modification they applied to a specific rule did not affect the rest of the process. The log-based differencing algorithm itself could also be key in improving the efficiency of the refactoring process, helping to discover the best candidate rule to apply next.

For future work, two avenues are open: either within the context of Raincode Labs or outside of it. Within, we could develop the optimisation tool imagined from the log-based behavioural differencing and analyse how different strategies of migration rule ordering affect the execution time of the refactoring process, but also if it affects its output at all. A second idea of future work is to further refine the trace equivalence algorithm and the fuzzy parser, allowing them to target specific parts of a file and to have more special cases. We could study how much effort is required for every one of the different implementations, its effect on the output in terms of matches and execution time, and therefore discuss its advantages and tradeoffs.

Outside of Raincode Labs, we could prove that our techniques are applicable to other contexts. For the log-based behavioural differencing and its visualisation, we could explore the idea of comparing logs from the boot sequence of different Linux distributions. The goal would be to show if both compared distributions are compatible with a specific set of software, i.e. if they have a specific set of features that are necessary. The input of the tool would be the boot log from a known-compatible distribution as the source and the boot log from another distribution as the target. The expected output of the tool would be slightly different than what we did with Raincode Labs: the tool should highlight if all necessary features are present in the target distribution. This means that we would only need to show the presence or absence of the target features, which should be simpler than showing their evolution.

Finally, we could analyse a different automated refactoring process (e.g. one that has been presented in chapter 3), in another language than COBOL. For this, we would need to perform another analysis on the structure most

affected by this new refactoring tool, and then adapt our semi-parser and CFG generator. The CFG to LTS translation script should also be adapted to take into account the specificities of the new language, but the trace equivalence algorithm should be applicable as-is. We could then analyse its output to look for other special cases that would be specific to the tool, as well as evaluate if the ones that we created for the COBOL refactoring tool apply to other tools. An analysis of the amount of matches possible in this new context would also be of interest to further refine our comprehension of how adequate the algorithm of trace equivalence is in the context of showing that refactorings are behaviour-preserving.

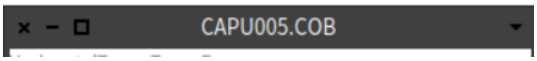
To conclude, this thesis allowed us to explore the boundaries of the discipline of differencing. First, thanks to our partner Raincode Labs, we had both real industrial-scale data and concrete issues to solve, which gave us a clear direction and objectives as well as provided unique challenges like the use of a legacy language, COBOL. On the other hand, the feedback from the scientific community allowed us to push the techniques we worked on even further by reflecting on ways to make them applicable outside of the initial context they were imagined for.

User manual of log-based behavioural differencing

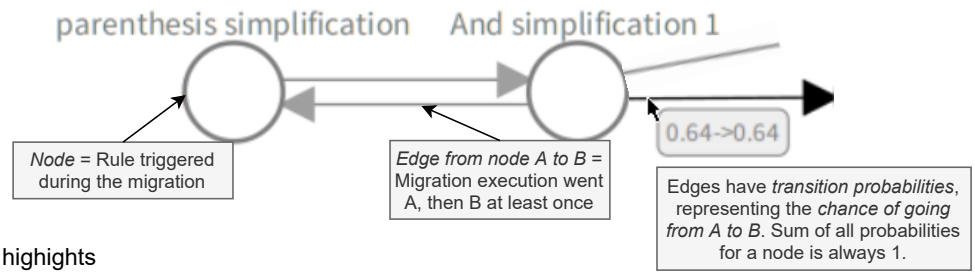
A

In this appendix, we include the user manual presented to the two participants of the validation of the log-based behavioural differencing tool. This is referenced in chapter 6.

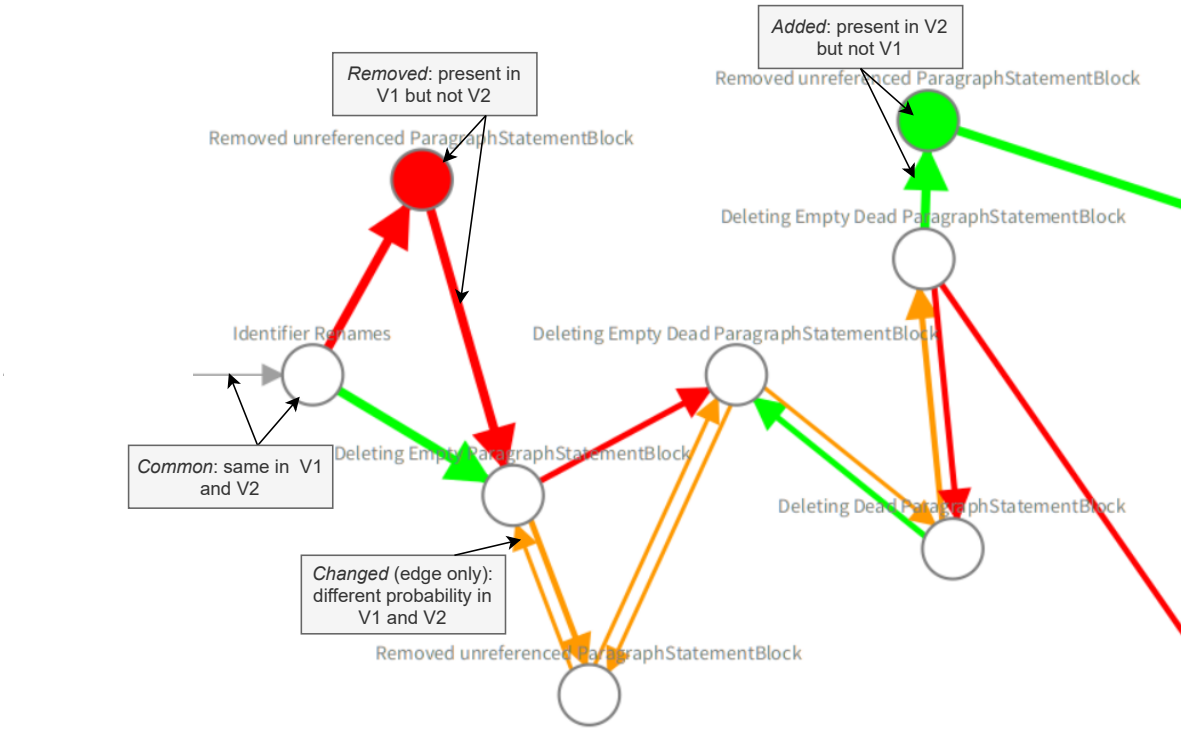
Each graph represent the diff for the migration of **one** program (name as title of window). The first version is the delivery, and the second is the redelivery



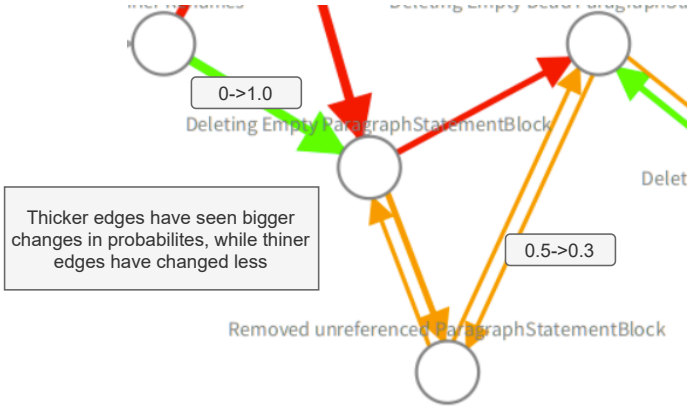
Base graph elements: node and edge



Colors in graphs: diff highights



Line thickness: Variation in probabilities



Particularities: fused and first-seen nodes

Questions of log-based behavioural differencing validation

B

In this appendix, we include all questions that the two participants of the validation of the log-based behavioural differencing tool asked themselves during the interview. They are timestamped and color-coded according to their category, as described in chapter 6.

	Questions about the meaning of a specific visual element (conceptua	Questions about the domain	
	Questions about the use of the tool (practical questions)	Questions/ideas about future work	
SubjectA			
Video time	Question	Context	Action taken
5'30"	What is going on here?	We are at the very beginning of the interview, SubjectA is getting situated in the tool	- Spread out the different graph files that are open to see them all - Open one at random in full screen - Zoom at the beginning of the graph
6'40"	What are the elements present here?	"	- Scroll in the graphs, identifying all the different elements (colors, shapes)
7'45"	"Trying to understand how this graph was generated" (direct quote)	"	-Starting back at the start of the graph, trying to read all the nodes - Realise he has not enough information as is
8'20"	What was in the cheat sheet again?	Prompted by me to look at it	- Opens his email, the sheet and starts looking
8'30	"What does this tool take as input?"	After looking at the sheet for a few seconds	- Looks for the answer for ~10 seconds then asks me directly. I inform him on the "level" of the input (the patches from the database
9'30"	Did I understand this information correctly?	After looking at the sheet for som more time	- Explain by example what he understood so I can confirm it is correct
10'20"	Can I now better understand the graphs?	"	- Goes back to the first graph, starts an explanation to confirm good understanding. - Keep exploring once confirmed
11'30	What happened in that long line of nodes?	Giving me his interpretation of the graphs	- Drag and drop in the graph without changing the zoom level, reading the names of the nodes
12'14"	"Can you handle rules that are repeated in the migration process?"	"	- Scrolling around for a few seconds, then giving up and asking me directly, probably because that's not an easy info to get as is
13'35"	How can I see better this specific part of the graph?	"	- Slides nodes around to rearrange them (unsuccesfully this time, because we were in a GOTO cluster)
14'00"	How do other graphs look?	"	- Minimize current window by double-clicking, then select another graph to maximize it
14'10"	What could be contained in that gray node?	"	- Mouse over said node and sees all the rules
14'30"	How could I analyze this new graph?	"	- Spread out the different nodes a bit, mousing over a few of their names (we are on the very simple graph, not much to see here
14'50"	How could I analyze this new graph?	"	- Stop to look at the graph as a whole first, then zoom to look at the different parts more in detail
15'05"	What was the meaning of orange again?	The orange color is on the links only	- Open sheet sheet, find the orange bits and read the doc
15'20"	Could you clarify the explanation on the cheat sheet?	The explanation talks about probabilities, which is not the most intuitive thing	- I explain the orange in the simpler terms of a transition "happening more or lest often", and SubjectA seems to understand
15'40"	I need to read that node title better	"	- Drags node just a bit above (happens a few times in a row as SubjectA is analyzing this last graph
19'50"	"Did you compare those graphs with the diff output of both migrated version corresponding to the logs?"		SubjectA thinks it might be interesting and insightfull to do so. I haven't done much of that beyond making sure that there are indeed differences when I detect them (and none when I detect none)
[Big gap for discussing ideas about other features etc]			
	"Could you go more abstract?"	SubjectA would like to see those graphs at a higher level (rules, not patches)	
	"Could you indicate where in the migration we are, when exiting a fused node?"		
15'20"	"How do I open a new graph"	I asked SubjectA to open the unfused graphs	- Feeds the correct code, maximize the window
25'30"	I don't see anything ! What is going on here?		- Zoom, zoom and zoom. Find the color. Zoom some more ! Then scroll around
26'15"	Where is the information we had before?	Prompted by one of my questions	- Drag some more until he sees color, then focus on that
SubjectA is a bit unsure about the possibility of having rules appear multiple times. He has trouble visualizing it, and therefore fears it might be confusing. I don't have an example to show him, so it's hard for me to reassure him			
[The end of the interview was only made of SubjectA answering my different questions. While he did fiddle with the graphs, he didn't seem to need any more "new" information and therefore probably didn't ask himself any more questions regarding the tool]			
What does SubjectA do when "explaining his domain" (happens multiple time during the process:			- Points at one node, explain what the rule does in general, then follow one link to another node. Explain how they might interact, then go to the bigger picture of the entire structure we are looking at and give an interpretation for the changes depicted (client removed code, one rule gets triggered before another, causing chain reaction, etc)
SubjectB			
0'00"	"So this is from the database, at the patch level?"	I consulted SubjectB previously, so he expects the input to be this way	- I confirm that he is right
0'10"	What is going on here?	The files just opened, and SubjectB is getting situated	- Spread out the different graph files, look at the help menu
0'20"	How can I see better?	"	- Zoom on the graphs, first without maximizing the video
0'25"	How do I maximize this?	"	- Stuggles to find the maximize option (probably because it is backwards, Unix-style)
0'30"	Okay, now what is written here?	"	- Zoom in the middle part of the graph (there is lots of color there), then drag nodes around right away
0'45"	How can I arrange this in a way that suits me better?	"	- Drags nodes to try and arrange them in a more curved pattern
1'10"	How do I navigate this?	"	- Identifie the bar at the bottom of the screen, grab it and drag until the beginning
1'20"	What does this do?	SubjectB just noticed the layout button (horizontal, vertical, force)	- Clicks and tries them out. End up prefering the force-based layout and sticking with it
1'30"	How can I arrange this to read it better?	Confronted with a "GOTO cluster", SubjectB tries to make it more readable	- Drags nodes, but quickly gives up because of the complexity
2'30"	"This is a state machine, right?"	SubjectB wants to confirm with me that he understands what he sees	- I tell him, that yes, it is
2'40"	What does the mouse over do?	SubjectB is currently explaining his interpretation of the graph, pointing as he explai	- Put mouse over node and reads out the rule name from the popup rather than the label
3'00"	"An arrow means this patch was executed, then that one?"	SubjectB is pointing at nodes and confirming what he sees	- I tell him that it is correct
4'50" + 6'20	What is in this fused node?	"	- Put mouse over node and quickly look at all the names
5'30"	SubjectB explains that he expects to see the node "Remove labels" s	SubjectB is explaining his domain to me	- I confirm that I saw it often, even if I don't know "by heart" where it is on the graph we are looking at
5'45"	Can a node be repeated in the graph?	SubjectB paused because he didn't think so	- I explain that yes, it can be, and go a bit over how an FSA works. SubjectB seems to dislike the idea of repeating nodes
7'00"	"Can you clarify the meaning of the orange and the gray nodes?"		- I explain the bit about rules happening more or less, and that the fused nodes means no changes.
7'40"	How can I open a new graph?	Prompted by me, SubjectB is looking at the other example graphs	- Minimize easily the first window, then clicks on the maximize for the other graph (less struggling here, still not very natural)
7'45"	How can I make this go away?	We had a small bug where, even if the window did maximize, another one stayed "c	- Shakes the window around in disbelief, then clicks the minimize button, making it really go away
7'55"	What is going on in this new graph?	Going back to analyzing	- Drags a few nodes and mouse over a few other while reflecting
9'00"	"Could you show new rules?" (i.e. the new application of rules)	SubjectB would like new rules highlighted, either new to the file or new to the entire	- I explain that we have this for the entire process, but that I could do it for a single file as well
9'30"	How is a new rule (in the entire process) shown?	After asking me this, I directed him to the graph showing one	- Looks at the graph a few seconds, searching for the yellow node then mouse over it
11'40"	But then, what are the green nodes?	A bit confused after all this discussion about new nodes to a file or to the entire proc	- I explain the green nodes and what it means
[Again, quite a big gap for discussing ideas and aswering questions. As SubjectA, SubjectB "fiddles" with the grah while we talk, but not to answer any questions I think]			
19'30"	"How do I open a new graph"	I asked SubjectA to open the unfused graphs	- Feeds the correct code, maximize the window
19'40"	What is this? I can't see	SubjectB just open the unfused graph	- Maximize window (navigating now looks and feels more natural, click his favorite "fused" layout
20'00"	How can I arrange this better?	"	- After zooming in a bit, moves node around to rearrange them
21'00"	Where are the informations I saw earlier?	Prompted by one question	- SubjectB looks for the color in the graph and circle it with his mouse
22'00"	How can I see what is in a fused node?	Prompted by my explanation that this information is available	- Clicks the node, then maximize the new window
[Here as well, the end of the interview is SubjectB fiddeling with the graphs, more to occupy his hand I believe]			
When explaining his interpretation of a group of nodes, SubjectB has a similar approch as SubjectA: pointing at nodes and cicling groupes with his mouse, while he explains rules and their interaction. However, he goes directly to explaining why the **migration** changes (this rule got skipped, this rule triggered earlier), focusing much more on the process itself rather than on what the client did			

User manual of trace equivalence visualisation

C

In this appendix, we include the user manual presented to the two participants of the validation of the trace equivalence visualisation tool. This us referenced in chapter 12.

General meaning of shapes

Triangular nodes start the execution (they do not correspond to any code)

Simple round nodes are pieces of code

Node with a black outline are nodes that can result in the end of the execution

Meaning of simple colors

Green nodes and links have a match in the other graph, in all cases

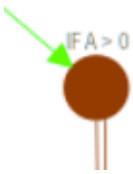
Red nodes and links have no match in the other graph, in any cases

Orange nodes and links have been skipped by the algorithm to find matches further down the graph, in all cases

Gray nodes and links have not been explored by the algorithm, we therefore have no information on them

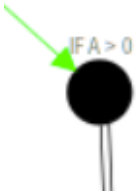
Meaning of combined colors

Dark green nodes and links have a match in the other graph in some cases and none in other cases



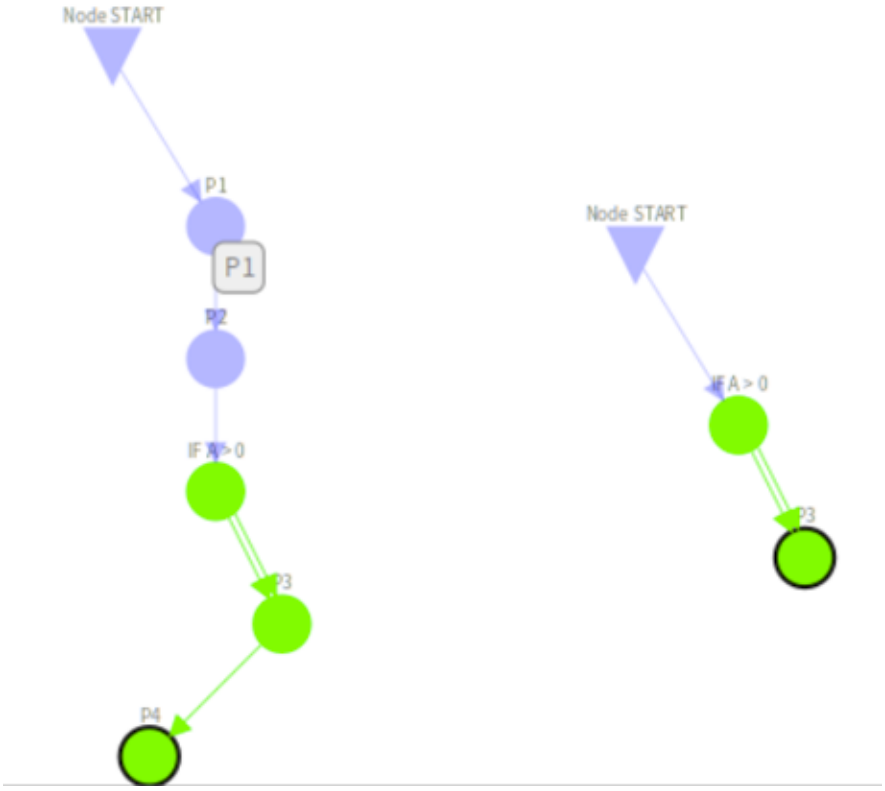
Brown nodes and links have a match in the other graph in some cases and have been skipped in other cases

Purple nodes and links have no match in the other graph in some cases and have been skipped in other cases

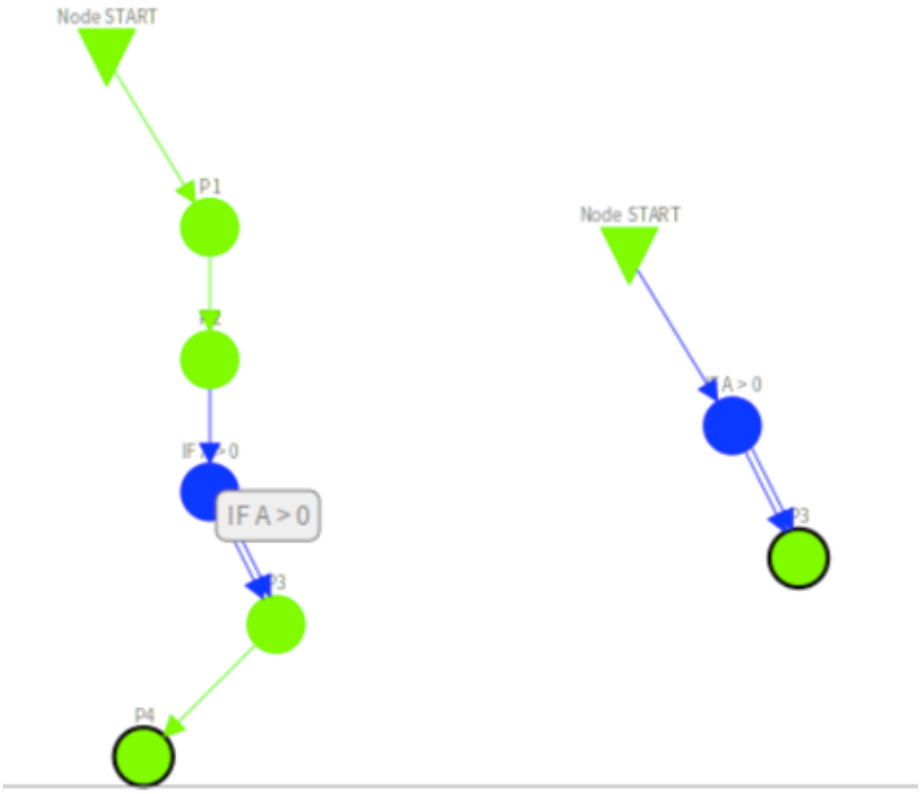


Black nodes and links have no match in the other graph in some cases, some matches in other and have been skipped in yet other cases

Interacting with the tool



Mousing over the graph nodes will highligth them if possible. Nodes that are matched toegether in groups will light up in light blue



Whenever possible, nodes that are an exact match to each other will light up in dark blue

Note that nodes that have no matches (red, orange, gray and purple) will never be have a group to highlight

Bibliography

[Ach+12] M. Acher, P. Heymans, P. Collet, C. Quinton, P. Lahire, and P. Merle. "Feature Model Differences". In: *Proceedings of the 24th International Conference on Advanced Information Systems Engineering (CAiSE)*. Vol. 7328. LNCS. Springer, 2012, pp. 629–645. ISBN: 978-3-642-31094-2. DOI: 10.1007/978-3-642-31095-9_41.

[Ada15] W. C. Adams. "Conducting Semi-Structured Interviews". In: *Handbook of Practical Program Evaluation*. John Wiley & Sons, Ltd, 2015. Chap. 19, pp. 492–505. ISBN: 9781119171386. DOI: https://doi.org/10.1002/9781119171386.ch19. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781119171386.ch19.

[Ali09] A. Alimucaj. *Eclipse Control Flow Graph Generator*. SourceForge, http://eclipsefcg.sf.net. 2009.

[All70] F. E. Allen. "Control Flow Analysis". In: *Proceedings of a Symposium on Compiler Optimization*. ACM, 1970, pp. 1–19. ISBN: 9781450373869. DOI: 10.1145/800028.808479.

[AlO+21] E. A. AlOmar, M. W. Mkaouer, C. Newman, and A. Ouni. "On preserving the behavior in software refactoring: A systematic mapping study". In: *Information and Software Technology* 140 (2021), p. 106675. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2021.106675.

[Alp87] A. Alper. "Users say Pacbase worth effort". In: *Computerworld* (Aug. 1987).

[Ami+12] A. Amighi, P. Gomes, D. Gurov, and M. Huisman. "Sound Control-Flow Graph Extraction for Java Programs with Exceptions". In: Oct. 2012, pp. 33–47. ISBN: 978-3-642-33825-0. DOI: 10.1007/978-3-642-33826-7_3.

[Anq+22] N. Anquetil, M. Campero, S. Ducasse, J.-P. Sandoval, and P. Tesone. "Transformation-based Refactorings: a First Analysis". In: *IWST 22 - International Workshop of Smalltalk Technologies*. Novisad, Serbia, 2022.

[AOH04] T. Apiwattanapong, A. Orso, and M. Harrold. "A differencing algorithm for object-oriented programs". In: *Proceedings. 19th International Conference on Automated Software Engineering, 2004.* 2004, pp. 2–13. DOI: 10.1109/ASE.2004.1342719.

[AST13] M. Andrews, C. Squire, and M. Tamboukou. *Doing narrative research.* Sage, 2013.

[Atw+21] H. Atwi, B. Lin, N. Tsantalis, Y. Kashiwa, Y. Kamei, N. Ubayashi, G. Bavota, and M. Lanza. "PyRef: refactoring detection in Python projects". In: *Proceedings of the 21st IEEE International Working Conference on Source Code Analysis and Manipulation.* IEEE. 2021, pp. 136–141. DOI: 10.1109/SCAM52516.2021.00025.

[Aub04] D. Auber. "Tulip — A Huge Graph Visualization Framework". In: *Graph Drawing Software* (Jan. 2004). DOI: 10.1007/978-3-642-18638-7_5.

[Awa+22] P. Awadhutkar, A. Tamrawi, R. Goluch, and S. Kothari. "Control flow equivalence method for establishing sanctity of compiling". In: *Computers & Security* 115 (2022), p. 102608. ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2022.102608.

[Bay+99] J. Bayer, J.-F. Girard, M. Wurthner, J.-M. DeBaud, and M. Apel. "Transitioning Legacy Assets to a Product Line Architecture". In: *Proceedings of the Second Joint Meeting of the Seventh European Software Engineering Conference and the Seventh Symposium on the Foundations of Software Engineering.* Vol. 1687. LNCS. Springer-Verlag, 1999, pp. 446–463. ISBN: 3-540-66538-2. DOI: 10.1007/3-540-48166-4_27.

[Bdv18] A. Bolt, M. de Leoni, and W. M. van der Aalst. "Process Variant Comparison: Using Event Logs to Detect Differences in Behavior and Business Rules". In: *Information Systems (selected papers from CAiSE 2016)* 74 (2018). DOI: 10.1016/j.is.2017.12.006, pp. 53–66. ISSN: 0306-4379.

[Ber+13] A. Bergel, D. Cassou, S. Ducasse, and J. Laval. *Deep Into Pharo.* Square Bracket Associates, 2013, p. 420. ISBN: 978-3-9523341-6-4.

[Ber+21] A. Bergel et al. Roassal homepage. Online: http://agilevisualization.com/. 2021.

[Ber02] J.-P. Bernardy. "Reviving Pacbase COBOL-generated code". In: *Proceedings of the 26th Annual International Computer Software and Applications.* IEEE, 2002. DOI: 10.1109/CMPSAC.2002.1045091.

[Ber16] A. Bergel. *Agile Visualization.* LULU Press, 2016. ISBN: 9781365314094.

[Bes+21a] I. Beschastnikh et al. Perfume frontend GitHub. Online: `https://github.com/ModelInference/perfume-frontend`. 2021.

[Bes+21b] I. Beschastnikh et al. Synoptic GitHub page. Online: `https://github.com/ModelInference/synoptic`. 2021.

[BF72] A. W. Biermann and J. A. Feldman. "On the Synthesis of Finite-State Machines from Samples of Their Behavior". In: *IEEE Transactions on Computers* C-21.6 (1972). DOI: 10.1109/TC.1972.5009015, pp. 592–597. DOI: 10.1109/TC.1972.5009015.

[BF99] K. Beck and M. Fowler. *Refactoring: Improving the Design of Existing Code*. USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0201485672.

[Bla16] D. Blasband. *Rise and Fall of Software Recipes*. Reality Bites Publishing, 2016.

[Bla98] D. Blasband. "Automatic parser generation for ancient languages". PhD thesis. PhD thesis, Universit éeLibre de Bruxelles, 1998.

[BMM06] D. Bruschi, L. Martignoni, and M. Monga. "Detecting self-mutating malware using control-flow graph matching". In: *Proceedings of the Third International Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. Springer. 2006, pp. 129–143.

[BSL19] C. K. Behera, G. Sanjog, and D. Lalitha Bhaskari. "Control Flow Graph Matching for Detecting Obfuscated Programs". In: *Software Engineering*. Ed. by M. N. Hoda, N. Chauhan, S. M. K. Quadri, and P. R. Srivastava. Singapore: Springer Singapore, 2019, pp. 267–275. ISBN: 978-981-10-8848-3.

[BWK05] S. Berner, R. Weber, and R. K. Keller. "Observations and Lessons Learned from Automated Testing". In: *Proceedings of the 27th International Conference on Software Engineering*. ICSE '05. St. Louis, MO, USA: Association for Computing Machinery, 2005, pp. 571–579. ISBN: 1581139632. DOI: 10.1145/1062455.1062556.

[Cam+10] D. O. Camacho, K. Mens, M. van den Brand, and J. Vinju. "Automated generation of program translation and verification tools using annotated grammars". In: *Science of Computer Programming* 75.1-2 (2010), pp. 3–20. DOI: 10.1016/j.scico.2009.10.003.

[CC90] E. Chikofsky and J. Cross. "Reverse engineering and design recovery: a taxonomy". In: *IEEE Software* 7.1 (1990), pp. 13–17. DOI: 10.1109/52.43044.

[CG14] J. L. Cánovas Izquierdo and J. García Molina. "Extracting models from source code in software modernization". In: *Software and Systems Modeling* 13.2 (2014), pp. 713–734. ISSN: 16191374. DOI: 10.1007/S10270-012-0270-Z/FIGURES/25.

[Cio+14] Ş. Ciobâcă, D. Lucanu, V. Rusu, and G. Roşu. "A Language-Independent Proof System for Mutual Program Equivalence". In: *Formal Aspects of Computing* 28 (2014). DOI: 10.1007/s00165-016-0361-7.

[CL04] C.-C. Chiang and R. Y. Lee. "Developing tools for reverse engineering in a software product-line architecture". In: *Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration (IRI)*. IEEE. 2004, pp. 42–47. DOI: 10.1109/IRI.2004.1431434.

[CM08] D. Campbell and M. Miller. "Designing Refactoring Tools for Developers". In: *Proceedings of the 2nd Workshop on Refactoring Tools*. WRT '08. Nashville, Tennessee: Association for Computing Machinery, 2008. ISBN: 9781605583396. DOI: 10.1145/1636642.1636651.

[Coe+18] A. Coet, L. Truong, V. Kumar, A. Naku, and R. Golosynsky. *StatiCFG*. https://github.com/coetaur0/staticfg. 2018.

[Cor+00] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. "Bandera: Extracting Finite-State Models from Java Source Code". In: *Proceedings of the 22nd International Conference on Software Engineering*. ICSE '00. Limerick, Ireland: Association for Computing Machinery, 2000, pp. 439–448. ISBN: 1581132069. DOI: 10.1145/337180.337234.

[Cra02] S. Craciunescu. "Proving the Equivalence of CLP Programs". In: *Proceedings of the 18th International Conference on Logic Programming*. ICLP '02. Springer-Verlag, 2002, pp. 287–301. ISBN: 3540439307. DOI: 10.5555/645522.656324.

[DB09] D. K. Deeptimahanti and M. A. Babar. "An Automated Tool for Generating UML Models from Natural Language Requirements". In: *2009 IEEE/ACM International Conference on Automated Software Engineering*. 2009, pp. 680–682. DOI: 10.1109/ASE.2009.48.

[DDN00] S. Demeyer, S. Ducasse, and O. Nierstrasz. "Finding refactorings via change metrics". In: vol. 35. Oct. 2000, pp. 166–177. DOI: 10.1145/353171.353183.

[De 87] R. De Nicola. "Extensional equivalences for transition systems". In: *Acta Informatica* 24.2 (1987), pp. 211–237.

[Dea+03] T. R. Dean, J. R. Cordy, A. J. Malton, and K. A. Schneider. "Agile Parsing in TXL". In: *Automated Software Engineering* 10.4 (2003), pp. 311–336. DOI: 10.1023/A:1025801405075.

[Dec+19] M. Decker, M. Collard, L. Volkert, and J. Maletic. "srcDiff: A Syntactic Differencing Approach to Improve the Understandability of Deltas". In: *Journal of Software: Evolution and Process* 32.4 (Oct. 2019). DOI: 10.1002/smr.2226, e2226.

[Dek+20a] C. Deknop, J. Fabry, K. Mens, and V. Zaytsev. "Advanced Differencing of Legacy Code and Migration Logs". In: *Pre-proceedings of the 13th Seminar on Advanced Techniques and Tools for Software Evolution (SATToSE)*. Ed. by A. Oprescu and E. Constantinou. 2020.

[Dek+20b] 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)*. Ed. by M. Morisio, M. Torchiano, and A. Jedlitschka. Springer, 2020, pp. 270–286. ISBN: 978-3-030-64147-4. DOI: 10.1007/978-3-030-64148-1_17.

[Dek+21] 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)*. Ed. by A. Telea, L. Merino, and J. P. S. Alcocer. IEEE, 2021, pp. 1–11. DOI: 10.1109/VISSOFT52517.2021.00010.

[Dek+22a] C. Deknop, J. Fabry, K. Mens, and V. Zaytsev. "Generating Customised Control-Flow Graphs for Legacy Languages with Semi-Parsing". In: *Proceedings of the 38th IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Ed. by R. Koschke, G. Papadopoulos, A. Capiluppi, and S. Han. Best Artefact Award. 2022. DOI: 10.1109/ICSME55016.2022.00072.

[Dek+22b] C. Deknop, J. Fabry, K. Mens, and V. Zaytsev. "Visualising CFG Differences Through Trace Equivalence". In: *21st Belgium-Netherlands Software Evolution Workshop (BENEVOL)*. Ed. by T. Mens, E. Constantinou, and M. Wessel. 2022.

[Dek21] C. Deknop. *Validation data on GitHub*. Online: https://github.com/CelineDknp/PACBASEValidationData. 2021.

[Dek22] C. Deknop. *FuzzyParser GitHub repository*. https://github.com/CelineDknp/SemiParsingCFG. 2022.

[Dil+22] M. Dilhara, A. Ketkar, N. Sannidhi, and D. Dig. "Discovering repetitive code changes in Python ML systems". In: *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering*. IEEE/ACM. 2022, pp. 736–748. DOI: 10.1145/3510003.3510225.

[DKU17] L. Duarte, J. Kramer, and S. Uchitel. "Using contexts to extract models from code". In: *Software & Systems Modeling* 16 (2017), pp. 523–557. DOI: 10.1007/s10270-015-0466-0.

[Ell+22] J. Ellson, E. Gansner, Y. Hu, S. North, et al. *Graphviz*. https://graphviz.org/. 2022.

[EnS22] EnSoft Corp. *Atlas for Java and C — Understand Code Someone Else Wrote!* https://www.ensoftcorp.com/atlas/sdk/. 2022.

[ESP21] S. Erdweg, T. Szabó, and A. Pacak. "Concise, Type-Safe, and Efficient Structural Diffing". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021, pp. 406–419. ISBN: 9781450383912. DOI: 10.1145/3453483.3454052.

[Fal+14] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. "Fine-Grained and Accurate Source Code Differencing". In: *Proceedings of the 29th International Conference on Automated Software Engineering (ASE)*. ACM, 2014, pp. 313–324. ISBN: 9781450330138. DOI: 10.1145/2642937.2642982.

[GDM12] X. Ge, Q. L. DuBose, and E. Murphy-Hill. "Reconciling manual and automatic refactoring". In: *2012 34th International Conference on Software Engineering (ICSE)*. 2012, pp. 211–221. DOI: 10.1109/ICSE.2012.6227192.

[GJ08] D. Grune and C. J. H. Jacobs. *Parsing Techniques — A Practical Guide*. Second. Addison-Wesley, 2008.

[Gop17] R. Gopinath. *PyCFG for Python MCI*. https://github.com/vrthra/pycfg. 2017.

[Gra20] GrammaTech. *CodeSonar C/C++: SAST when Safety and Security Matter*. https://www.grammatech.com/codesonar-cc. 2020.

[Gre+09] B. Gretarsson, S. Bostandjiev, J. O'Donovan, and T. Höllerer. "WiGis: A Framework for Scalable Web-Based Interactive Graph Visualizations". In: vol. 5849. Sept. 2009, pp. 119–134. ISBN: 978-3-642-11804-3. DOI: 10.1007/978-3-642-11805-0_13.

[GRS17] 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)*. DOI: 10.1109/ISSRE.2017.14. 2017, pp. 282–293.

[GS10] B. Godlin and O. Strichman. "Inference Rules for Proving the Equivalence of Recursive Procedures". In: vol. 45. 2010, pp. 167–184. ISBN: 978-3-642-13753-2. DOI: 10.1007/978-3-642-13754-9_8.

[Gut+08] J. Gutierrez, C. Nebut, M. Escalona, M. Mejías, and I. Ramos. "Visualization of Use Cases Through Automatically Generated Activity Diagrams". In: vol. 5301. Sept. 2008. ISBN: 978-3-540-87874-2. DOI: 10.1007/978-3-540-87875-9_6.

[GW20] S. Gholamian and P. A. S. Ward. "Logging Statements' Prediction Based on Source Code Clones". In: *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. SAC. ACM, 2020, pp. 82–91. ISBN: 9781450368667. DOI: 10.1145/3341105.3373845.

[HA08] S. Hosseini and M. A. Azgomi. "UML Model Refactoring with Emphasis on Behavior Preservation". In: *2008 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering*. 2008, pp. 125–128. DOI: 10.1109/TASE.2008.43.

[HA23] S. Heidari and S. Ajoudanian. "Automatic pattern-based consistency checking in model refactoring: introducing a formal behavioral preserving method". In: *Innovations in Systems and Software Engineering* (Feb. 2023), pp. 1–20. DOI: 10.1007/s11334-022-00525-8.

[Hay+22] K. Hayen et al. *Nuitka the Python Compiler*. https://nuitka.net/index.html. 2022.

[Hem+21] I. Hemati Moghadam, M. Ó. Cinnéide, F. Zarepour, and M. A. Jahanmir. "RefDetect: A Multi-Language Refactoring Detection Tool based on String Alignment". In: *IEEE Access* 9 (2021), pp. 86698–86727. DOI: 10.1109/ACCESS.2021.3086689.

[Hew12] 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.

[HH01] G. J. Holzmann and M. H. Smith. "Software model checking: extracting verification models from source code†". In: *Software Testing, Verification and Reliability* 11.2 (2001), pp. 65–79. DOI: https://doi.org/10.1002/stvr.228.

[Hil14] M. Hills. "Streamlining Control Flow Graph Construction with DCFlow". In: *Proceedings of the Seventh International Conference on Software Language Engineering*. Ed. by B. Combemale, D. J. Pearce, O. Barais, and J. J. Vinju. Vol. 8706. LNCS. Springer, 2014, pp. 322–341. ISBN: 978-3-319-11244-2. DOI: 10.1007/978-3-319-11245-9_18.

[HM76] J. W. Hunt and M. D. MacIlroy. *An algorithm for differential file comparison*. Bell Laboratories Murray Hill, 1976.

[HMM00] I. Herman, G. Melancon, and M. Marshall. "Graph visualization and navigation in information visualization: A survey". In: *IEEE Transactions on Visualization and Computer Graphics* 6.1 (2000), pp. 24–43. DOI: 10.1109/2945.841119.

[How96] D. J. Howe. "Proving Congruence of Bisimulation in Functional Programming Languages". In: *Information and Computation* 124.2 (1996), pp. 103–112. ISSN: 0890-5401. DOI: https://doi.org/10.1006/inco.1996.0008.

[HTS13] S. Hayashi, S. Thangthumachit, and M. Saeki. "REdiffs: Refactoring-aware difference viewer for Java". In: *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE)*. 2013, pp. 487–488. DOI: 10.1109/WCRE.2013.6671331.

[Hua+18] K. Huang, B. Chen, X. Peng, D. Zhou, Y. Wang, Y. Liu, and W. Zhao. "ClDiff: Generating Concise Linked Code Differences". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE '18. Montpellier, France: Association for Computing Machinery, 2018, pp. 679–690. ISBN: 9781450359375. DOI: 10.1145/3238147.3238219.

[HVT96] J. J. Hunt, K.-P. Vo, and W. F. Tichy. "An Empirical Study of Delta Algorithms". In: *Proceedings of the Sixth Workshop on System Configuration Management (SCM)*. Springer, 1996, pp. 49–66. ISBN: 354061964X. DOI: 10.1007/BFb0023080.

[IBM20] IBM. PACBASE documentation page. Online: https://www.ibm.com/support/pages/documentation-visualage-pacbase. 2020.

[IBM23a] IBM. CA-DATACOM documentation page. Online: https://www.ibm.com/docs/fr/omegamon-for-cics/5.5.0?topic=support-installing-ca-datacom-ca-ideal. 2023.

[IBM23b] IBM. *COBOL User's Guide*. https://www.ibm.com/docs/en/i/7.4?topic=languages-cobol. 2023.

[IBM23c] IBM. *VisualAge Pacbase documentation*. https://public.dhe.ibm.com/software/vapacbase/pdf30_f/spe353f.pdf. 2023.

[IDE] E. IDE. *JSparrow main page*. https://jsparrow.io/. Accessed: 2023-02-24.

[Ive+22] J. Ivers, R. L. Nord, I. Ozkaya, C. Seifried, C. S. Timperley, and M. Kessentini. "Industry Experiences with Large-Scale Refactoring". In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2022. Singapore, Singapore: Association for Computing Machinery, 2022, pp. 1544–1554. ISBN: 9781450394130. DOI: 10.1145/3540250.3558954.

[JF09] P. Julius and J. Fredrick. "Creating habitable code: Lessons in longevity from cruisecontrol". In: *Proceedings - 2009 Agile Conference, AGILE 2009* (2009), pp. 259–264. DOI: 10.1109/AGILE.2009.21.

[Jon98] C. Jones. *The Year 2000 Software Problem — Quantifying the Costs and Assessing the Consequences*. Addison-Wesley-Longman, 1998. ISBN: 978-0-201-30964-5.

[Kha+14] R. Khadka, B. V. Batlajery, A. M. Saeidi, S. Jansen, and J. Hage. "How Do Professionals Perceive Legacy Systems and Software Modernization?" In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 36–47. ISBN: 9781450327565. DOI: 10.1145/2568225.2568318.

[Kim+10] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit. "Ref-Finder: A Refactoring Reconstruction Tool Based on Logic Query Templates". In: *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 2010. ISBN: 9781605587912. DOI: 10.1145/1882291.1882353.

[KK15] I. Kochurkin and I. Khudyashev. *MySQL (Positive Technologies) grammar*. https://github.com/antlr/grammars-v4/tree/master/sql/mysql/Positive-Technologies. Licensed under the MIT License. 2015.

[KLV05] P. Klint, R. Lämmel, and C. Verhoef. "Toward an Engineering Discipline for Grammarware". In: *ACM Trans. Softw. Eng. Methodol.* 14.3 (2005), pp. 331–380. ISSN: 1049-331X. DOI: 10.1145/1072997.1073000.

[KM17] R. Khatchadourian and H. Masuhara. "Automated Refactoring of Legacy Java Software to Default Methods". In: *Proceedings of the 39th International Conference on Software Engineering*. ICSE. Buenos Aires, Argentina: IEEE Press, 2017, pp. 82–93. ISBN: 9781538638682. DOI: 10.1109/ICSE.2017.16.

[KN09] M. Kim and D. Notkin. "Discovering and Representing Systematic Code Changes". In: *Proceedings of the 31st International Conference on Software Engineering (ICSE)*. IEEE, 2009, pp. 309–319. ISBN: 9781424434534. DOI: 10.1109/ICSE.2009.5070531.

[KNG07] M. Kim, D. Notkin, and D. Grossman. "Automatic Inference of Structural Changes for Matching across Program Versions". In: *Proceedings of the 29th International Conference on Software Engineering (ICSE)*. IEEE, 2007, pp. 333–343. DOI: 10.1109/ICSE.2007.20.

[KO04] H. Koike and K. Ohno. "SnortView: Visualization System of Snort Logs". In: New York, NY, USA: Association for Computing Machinery, 2004. ISBN: 1581139748. DOI: 10.1145/1029208.1029232.

[Kol+06] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi. "Refactoring a Legacy Component for Reuse in a Software Product Line: A Case Study: Practice Articles". In: *Journal of Software Maintenance and Evolution: Research and Practice* 18.2 (2006), pp. 109–132. ISSN: 1532-060X. DOI: 10.1002/smr.329.

[Kom+20] B. Komal, U. I. Janjua, F. Anwar, T. M. Madni, M. F. Cheema, M. N. Malik, and A. R. Shahid. "The Impact of Scope Creep on Project Success: An Empirical Investigation". In: *IEEE Access* 8 (2020), pp. 125755–125775. DOI: 10.1109/ACCESS.2020.3007098.

[KOP97] R. KOPPLER. "A Systematic Approach to Fuzzy Parsing". In: *Software: Practice and Experience* 27.6 (1997), pp. 637–649. DOI: https://doi.org/10.1002/(SICI)1097-024X(199706)27:6<637::AID-SPE99>3.0.CO;2-3.

[KSI22] T. Kanda, K. Shimari, and K. Inoue. "Didiffff: A Viewer for Comparing Changes in Both Code and Execution Traces". In: *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. ICPC '22. Virtual Event: Association for Computing Machinery, 2022, pp. 528–532. ISBN: 9781450392983. DOI: 10.1145/3524610.3527877.

[KSV09] P. Klint, T. van der Storm, and J. J. Vinju. "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation". In: *Proceedings of the Ninth International Working Conference on Source Code Analysis and Manipulation*. IEEE Computer Society, 2009, pp. 168–177. ISBN: 978-0-7695-3793-1. DOI: 10.1109/SCAM.2009.28.

[Kur+21] Z. Kurbatova, V. Kovalenko, I. Savu, B. Brockbernd, D. Andreescu, M. Anton, R. Venediktov, E. Tikhomirova, and T. Bryksin. "RefactorInsight: Enhancing ide representation of changes in git with refactorings information". In: *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*. IEEE. 2021, pp. 1276–1280. DOI: 10.1109/ASE51524.2021.9678646.

[KV10] L. C. L. Kats and E. Visser. "The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs". In: *Proceedings of the 25th Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. Ed. by W. R. Cook, S. Clarke, and M. C. Rinard. ACM, 2010, pp. 444–463. ISBN: 978-1-4503-0203-6. DOI: 10.1145/1869459.1869497.

[LBL06] J. Liu, D. Batory, and C. Lengauer. "Feature Oriented Refactoring of Legacy Applications". In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE. Shanghai, China: Association for Computing Machinery, 2006, pp. 112–121. ISBN: 1595933751. DOI: 10.1145/1134285.1134303.

[Lev06] P. B. Levy. "Infinite Trace Equivalence". In: *Electron. Notes Theor. Comput. Sci.* 155 (2006), pp. 467–496. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2005.11.069.

[LHM03] Y. Lin, R. Holt, and A. Malton. "Completeness of a fact extractor". In: Dec. 2003, pp. 196–205. ISBN: 0-7695-2027-8. DOI: 10.1109/WCRE.2003.1287250.

[Li+19] Z. Li, T. Chen, J. Yang, and W. Shang. "DLFinder: Characterizing and Detecting Duplicate Logging Code Smells". In: *Proceedings of the 41st International Conference on Software Engineering (ICSE)*. 2019, pp. 152–163. DOI: 10.1109/ICSE.2019.00032.

[Li19] Z. Li. "Characterizing and Detecting Duplicate Logging Code Smells". In: *Companion Proceedings of the 41st International Conference on Software Engineering (ICSE), Student Research Competition*. DOI: 10.1109/ICSE-Companion.2019.00062. 2019, pp. 147–149.

[Lim14] H.-I. Lim. "Comparing Control Flow Graphs of Binary Programs through Match Propagation". In: *2014 IEEE 38th Annual Computer Software and Applications Conference*. 2014, pp. 598–599. DOI: 10.1109/COMPSAC.2014.84.

[LP14] W. Le and S. D. Pattison. "Patch Verification via Multiversion Interprocedural Control Flow Graphs". In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 1047–1058. ISBN: 9781450327565. DOI: 10.1145/2568225.2568304.

[Luo+92] G. Luo, G. Bochmann, B. Sarikaya, and M. Boyer. "Control-flow based testing of Prolog programs". In: *[1992] Proceedings Third International Symposium on Software Reliability Engineering*. 1992, pp. 104–113. DOI: 10.1109/ISSRE.1992.285853.

[LV01] R. Lämmel and C. Verhoef. "Cracking the 500-Language Problem". In: *IEEE Software* 18.6 (2001), pp. 78–88. DOI: 10.1109/52.965809.

[Mac12] L. A. Macaulay. *Requirements engineering*. Springer Science & Business Media, 2012.

[Mas98] V. Maslov. *Re: An Odd Grammar Question*. http://compilers.iecc.com/comparch/article/98-05-108. May 1998.

[MB08] E. Murphy-Hill and A. P. Black. "Refactoring Tools: Fitness for Purpose". In: *IEEE Software* 25.5 (2008), pp. 38–44. DOI: 10.1109/MS.2008.123.

[MC04] J. I. Maletic and M. L. Collard. "Supporting Source Code Difference Analysis". In: *Proceedings of the 20th International Conference on Software Maintenance (ICSM)*. DOI: 10.1109/ICSM.2004.1357805. IEEE, 2004, pp. 210–219. ISBN: 0-7695-2213-0.

[MDJ02] T. Mens, S. Demeyer, and D. Janssens. "Formalising Behaviour Preserving Program Transformations". In: Oct. 2002, pp. 286–301. ISBN: 978-3-540-44310-0. DOI: 10.1007/3-540-45832-8_22.

[Men+04] T. Mens, N. Eetvelde, D. Janssens, and S. Demeyer. "Formalising refactorings with graph transformations". In: *Journal of Software Maintenance and Evolution* 11 (Jan. 2004), pp. 1001–1025.

[Men02] T. Mens. "A state-of-the-art survey on software merging". In: *IEEE Transactions on Software Engineering* 28.5 (2002), pp. 449–462. DOI: 10.1109/TSE.2002.1000449.

[Mic23] Microsoft. COBOL Control Flow download page. Online: https://marketplace.visualstudio.com/items?itemName=broadcomMFD.ccf. 2023.

[ML19] H. Min and Z. Li Ping. "Survey on Software Clone Detection Research". In: *Proceedings of the Third International Conference on Management Engineering, Software Engineering and Service Sciences (ICMSS)*. ACM, 2019, pp. 9–16. ISBN: 9781450361897. DOI: 10.1145/3312662.3312707.

[MR15] S. Maoz and J. O. Ringert. "A framework for relating syntactic and semantic model differences". In: *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 2015, pp. 24–33. DOI: 10.1109/MODELS.2015.7338232.

[MRR11] 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)*. Ed. by T. Gyimóthy and A. Zeller. ACM, 2011, pp. 179–189. ISBN: 978-1-4503-0443-6. DOI: 10.1145/2025113.2025140.

[MS06] E. Mealy and P. Strooper. "Evaluating software refactoring tool support". In: *Australian Software Engineering Conference (ASWEC'06)*. 2006, 10 pp.–340. DOI: 10.1109/ASWEC.2006.26.

[Mui09] D. de Muinck Keizer. "A Control Flow Graph Generator for Java Code". PhD thesis. Leiden University, The Netherlands, 2009.

[Nag+07] V. Nagarajan, R. Gupta, M. Madou, X. Zhang, and B. D. Sutter. "Matching Control Flow of Program Versions". In: *Proceedings of the 23rd International Conference on Software Maintenance*. IEEE, 2007, pp. 84–93. DOI: 10.1109/ICSM.2007.4362621.

[Nat22] National Institute of Standards and Technology. *NIST Cobol Available Test Suites*. https://www.itl.nist.gov/div897/ctg/cobol_form.htm. 2022.

[Nea19] T. Neale. *GTIRB Ghidra Plugin*. GitHub, https://github.com/GrammaTech/gtirb-ghidra-plugin. 2019.

[Ngu+18] H. Nguyen, M. Dumas, M. La Rosa, and A. H. M. ter Hofstede. "Multi-perspective Comparison of Business Process Variants Based on Event Logs". In: *Conceptual Modeling*. Ed. by J. C. Trujillo, K. C. Davis, X. Du, Z. Li, T. W. Ling, G. Li, and M. L. Lee. DOI: 10.1007/978-3-030-00847-5_32. Cham: Springer, 2018, pp. 449–459. ISBN: 978-3-030-00847-5.

[Opd92] W. F. Opdyke. "Refactoring Object-Oriented Frameworks". PhD thesis. University of Illinois at Urbana-Champaign, 1992.

[Ora23] Oracle. *Fortran User's Guide*. https://docs.oracle.com/cd/E19059-01/fortec6u2/806-7988/806-7988.pdf. 2023.

[Pha21] Pharo consortium. Pharo homepage. Online: https://pharo.org. 2021.

[Pre+10] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. "Template-based reconstruction of complex refactorings". In: *Proceedings of 2010 IEEE International Conference on Software Maintenance*. IEEE. 2010, pp. 1–10. DOI: 10.1109/ICSM.2010.5609577.

[Pur00] H. Purchase. "Effective information visualisation: a study of graph drawing aesthetics and algorithms". In: *Interacting with Computers* 13.2 (2000), pp. 147–162. DOI: 10.1016/S0953-5438(00)00032-1.

[Pyt22] Python Software Foundation. *re — Regular expression operations*. https://docs.python.org/3/library/re.html. 2022.

[Rag+04] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine. "Dex: a semantic-graph differencing tool for studying changes in large code bases". In: *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* 2004, pp. 188–197. DOI: 10.1109/ICSM.2004.1357803.

[Rag02] G. Raghavan. "Improving software quality in product families through systematic reengineering". In: *Proceedings of the Seventh European Conference on Software Quality (ECSQ)*. Springer. 2002, pp. 90–99.

[Rai18] Raincode Labs. *PACBASE Migration: More than 200 Million Lines Migrated*. https://www.raincodelabs.com/pacbase/. 2018.

[Rai23] Raincode Labs. *Raincode Labs hope page*. https://www.raincodelabs.com/. 2023.

[RB08] M. Ribeiro and P. Borba. "Recommending Refactorings When Restructuring Variabilities in Software Product Lines". In: *Proceedings of the 2nd Workshop on Refactoring Tools*. WRT. Nashville, Tennessee: Association for Computing Machinery, 2008. ISBN: 9781605583396. DOI: 10.1145/1636642.1636650.

[Rém00] C. Rémy. *Un nouveau PacBase, entierement Java*. 01net, https://www.01net.com/actualites/un-nouveau-pacbase-entierement-java-114108.html. July 2000.

[Ren10] L. Renggli. *Dynamic Language Embedding*. Lulu. com, 2010.

[SD10] Q. D. Soetens and S. Demeyer. "Studying the Effect of Refactorings: A Complexity Metrics Perspective". In: *2010 Seventh International Conference on the Quality of Information and Communications Technology*. 2010, pp. 313–318. DOI: 10.1109/QUATIC.2010.58.

[SHV16] M. Schuts, J. Hooman, and F. Vaandrager. "Refactoring of Legacy Software Using Model Learning and Equivalence Checking: An Industrial Experience Report". In: June 2016, pp. 311–325. ISBN: 978-3-319-33692-3. DOI: 10.1007/978-3-319-33693-0_20.

[Sil+20] D. Silva, J. Silva, G. Santos, R. Terra, and M. T. Valente. "RefDiff 2.0: A Multi-language Refactoring Detection Tool". In: *IEEE Transactions on Software Engineering* (2020). DOI: 10.1109/TSE.2020.2968072.

[SMD08] J. Sillito, G. C. Murphy, and K. De Volder. "Asking and Answering Questions during a Programming Change Task". In: *IEEE Transactions on Software Engineering* 34.4 (2008). DOI: 10.1109/TSE.2008.26, pp. 434–451. DOI: 10.1109/TSE.2008.26.

[SN02] S. D. S. Demeyer and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann and DPunkt, 2002. ISBN: 9783952334126.

[Sof23] Softonic. *Meld download page*. https://meld.en.softonic.com/. May 2023.

[SPD13] Q. D. Soetens, J. Perez, and S. Demeyer. "An Initial Investigation into Change-Based Reconstruction of Floss-Refactorings". In: *2013 IEEE International Conference on Software Maintenance*. 2013, pp. 384–387. DOI: 10.1109/ICSM.2013.53.

[SV17] J. Smits and E. Visser. "FlowSpec: Declarative Dataflow Analysis Specification". In: *Proceedings of the 10th International Conference on Software Language Engineering (SLE)*. ACM, 2017, pp. 221–231. ISBN: 978-1-4503-5525-4. DOI: 10.1145/3136014.3136029.

[TC19] B. A. Tama and M. Comuzzi. "An Empirical Comparison of Classification Techniques for Next Event Prediction using Business Process Event Logs". In: *Expert Systems with Applications* 129 (2019). DOI: 10.1016/j.eswa.2019.04.016, pp. 233–245. ISSN: 0957-4174.

[Tic+00] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz. "A meta-model for language-independent refactoring". In: *Proceedings International Symposium on Principles of Software Evolution.* 2000, pp. 154–164. DOI: 10.1109/ISPSE.2000.913233.

[TKD20] N. Tsantalis, A. Ketkar, and D. Dig. "RefactoringMiner 2.0". In: *IEEE Transactions on Software Engineering* (2020). DOI: 10.1109/TSE.2020.3007722.

[TPB97] Q. M. Tan, A. Petrenko, and G. v. Bochmann. "Checking Experiments with Labeled Transition Systems for Trace Equivalence". In: *Testing of Communicating Systems: IFIP TC6 10th International Workshop on Testing of Communicating Systems, 8–10 September 1997, Cheju Island, Korea.* Ed. by M. Kim, S. Kang, and K. Hong. Boston, MA: Springer US, 1997, pp. 167–182. DOI: 10.1007/978-0-387-35198-8_11.

[TRB18] G. Torre, R. Robbes, and A. Bergel. "Imprecisions Diagnostic in Source Code Deltas". In: *Proceedings of the 15th International Conference on Mining Software Repositories.* MSR. DOI: 10.1145/3196398.3196404. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 492–502. ISBN: 9781450357166. DOI: 10.1145/3196398.3196404.

[Tre08] J. Tretmans. "Model Based Testing with Labelled Transition Systems". In: *Formal Methods and Testing: An Outcome of the FORTEST Network, Revised Selected Papers.* Ed. by R. M. Hierons, J. P. Bowen, and M. Harman. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–38. ISBN: 978-3-540-78917-8. DOI: 10.1007/978-3-540-78917-8_1.

[UCL03] UCLA Compilers Group. *Avrora: The AVR Simulation and Analysis Framework.* `http://compilers.cs.ucla.edu/avrora/cfg.html`. 2003.

[VD04] F. Van Rysselberghe and S. Demeyer. "Studying software evolution information by visualizing the change history". In: *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* 2004, pp. 328–337. DOI: 10.1109/ICSM.2004.1357818.

[WD14] T. Wood and S. Drossopoulou. "Program Equivalence through Trace Equivalence". In: *Foundations of Object Oriented Languages, FOOL* (2014).

[Wei84] M. Weiser. "Program Slicing". In: *IEEE Transactions on Software Engineering* SE-10.4 (1984), pp. 352–357. DOI: 10.1109/TSE.1984.5010248.

[Wyn+17] M. Wynn, E. Poppe, J. Xu, A. ter Hofstede, R. Brown, A. Pini, and W. van der Aalst. "ProcessProfiler3D: A visualisation framework for log-based process performance comparison". In: *Decision Support Systems* 100 (2017). Smart Business Process Management, pp. 93–108. ISSN: 0167-9236. DOI: https://doi.org/10.1016/j.dss.2017.04.004.

[Xin10] Z. Xing. "Model Comparison with GenericDiff". In: *Proceedings of the 25th International Conference on Automated Software Engineering (ASE)*. ACM, 2010, pp. 135–138. DOI: 10.1145/1858996.1859020.

[XS05] 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. DOI: 10.1145/1101908.1101919.

[XS07] Z. Xing and E. Stroulia. "API-Evolution Support with Diff-CatchUp". In: *IEEE Transactions on Software Engineering* 33.12 (2007), pp. 818–836. DOI: 10.1109/TSE.2007.70747.

[XXJ11] Y. Xue, Z. Xing, and S. Jarzabek. "CloneDiff: semantic differencing of clones". In: *Proceeding of the Fifth ICSE International Workshop on Software Clones (IWSC)*. Ed. by J. R. Cordy, K. Inoue, S. Jarzabek, and R. Koschke. ACM, 2011, pp. 83–84. DOI: 10.1145/1985404.1985428.

[Yan+23] N. Yang, P. Cuijpers, D. Hendriks, R. Schiffelers, J. Lukkien, and A. Serebrenik. "An interview study about the use of logs in embedded software engineering". In: *Empirical Software Engineering* 28.2 (2023), pp. 1–56. ISSN: 15737616. DOI: 10.1007/S10664-022-10258-8/TABLES/10.

[Yan91] W. Yang. "Identifying Syntactic Differences between Two Programs". In: *Software Practice & Experience* 21.7 (1991), pp. 739–755. ISSN: 0038-0644. DOI: 10.1002/spe.4380210706.

[YBL10] T. Yue, L. Briand, and Y. Labiche. "An Automated Approach to Transform Use Cases into Activity Diagrams". In: 2010, pp. 337–353. ISBN: 978-3-642-13594-1. DOI: 10.1007/978-3-642-13595-8_26.

[YM13] M. Ying and J. Miller. "Refactoring legacy AJAX applications to improve the efficiency of the data exchange component". In: *Journal of Systems and Software* 86.1 (2013), pp. 72–88. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2012.07.019.

[Zay11] V. Zaytsev. "Language Convergence Infrastructure". In: *Post-proceedings of the Third International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE)*. Ed. by J. M. Fernandes, R. Lämmel, J. Visser, and J. Saraiva. Vol. 6491. LNCS. Springer, Jan. 2011, pp. 481–497. DOI: 10.1007/978-3-642-18023-1_16.

[Zay14] V. Zaytsev. "Formal Foundations for Semi-parsing". In: *Proceedings of the IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE 2014 ERA)*. Ed. by S. Demeyer, D. Binkley, and F. Ricca. IEEE, Feb. 2014, pp. 313–317. DOI: 10.1109/CSMR-WCRE.2014.6747184.

[Zay15] V. Zaytsev. "Grammar Zoo: A Corpus of Experimental Grammarware". In: *Fifth Special issue on Experimental Software and Toolkits of Science of Computer Programming (SCP EST5)* 98 (Feb. 2015), pp. 28–51. DOI: 10.1016/j.scico.2014.07.010.

[Zay17] V. Zaytsev. "Parser Generation by Example for Legacy Pattern Languages". In: *Proceedings of the 16th International Conference on Generative Programming: Concepts and Experience (GPCE)*. Ed. by M. Flatt and S. Erdweg. ACM, 2017, pp. 212–218. ISBN: 978-1-4503-5524-7. DOI: 10.1145/3136040.3136058.

[Zay19] V. Zaytsev. "Event-Based Parsing". In: *Proceedings of the Sixth Workshop on Reactive and Event-based Languages and Systems (REBLS)*. Ed. by T. Kamina and H. Masuhara. 2019. DOI: 10.1145/3358503.3361275.

[ZB14] V. Zaytsev and A. H. Bagge. "Parsing in a Broad Sense". In: *Proceedings of the 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2014)*. Vol. 8767. LNCS. Springer, Oct. 2014, pp. 50–67. DOI: 10.1007/978-3-319-11653-2_4.

[Zim+04] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. "Mining Version Histories to Guide Software Changes". In: *Proceedings of the 26th International Conference on Software Engineering (ICSE)*. USA: IEEE, 2004, pp. 563–572. ISBN: 0769521630. DOI: 10.1109/ICSE.2004.1317478.