Understanding Large-scale Codebase Refactoring through Semantic-aware Differencing

Anonymous Author(s)

***Abstract*—Code refactoring is the act of restructuring the source code of a software system, to improve its design without changing the observable behaviour. When applying large automated refactorings, large either in number of refactorings applied or in codebase size, the refactored code may become significantly different from the original code. This makes it harder to provide evidence that the refactorings indeed preserve the original code behaviour. To demonstrate that the flow of control of a program remains equivalent before and after refactoring, we propose a tool that verifies trace equivalence on control flow graphs. Such a tool increases code owner's trust in the automated refactoring process. We present the tool's algorithm, implementation and visualisation support and illustrate and validate it by applying it on a large industrial case.**

***Index Terms*—trace equivalence, large-scale code refactoring, control flow graph, labelled-transition graph, visualisation, industrial use case, COBOL**

I. Introduction

When software ages, it tends to expand beyond the features initially planned, an effect known as scope creep [17]. Constant introduction of features and bug fixes makes code less healthy, harder to maintain, less habitable [14] and causes code smells. 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 routinely performed throughout the software life cycle. 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"* [25]. 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.

Murphy-Hill and Black [23] distinguish two main categories of refactoring. *Floss refactoring* refers to small tweaks applied opportunistically and frequently to a piece of code while making other changes. *Root-canal refactoring* refers to times when refactoring is the only change being made on the code and the creation of new features is temporarily halted. Large root-canal refactoring can even be regarded as a form of *reengineering* [9] when the subject system is so deeply transformed that it gets reconstituted in a new form [22]. A particular case of this, the use case of this paper, happens when a legacy system consisting of generated code has to be transformed into human-readable code so that maintenance activities on it can be continued.

When performed in-house, such large-scale root-canal refactoring tasks can take a considerable effort to a team, spanning from weeks to months [13] despite the use of automated tools. An alternative is to outsource, delegating such tasks to companies expert in the domain of migration and refactoring. However, when choosing this route and being presented with a new version of the code after it has been altered deeply, code owners tend to require guarantees to trust the result of the refactoring. Since changes are so vast, it is often impossible to manually verify them, creating the need for tools providing evidence that the behaviour of the code remains preserved.

We present a tool that compares both versions of a program (before and after a large-scale refactoring). The comparison is done on the level of control-flow graphs, abstracting away from statements to focus on the structure of both versions.[1] The tool relies on trace equivalence and software visualisation to provide evidence that the behaviour remains preserved after refactoring. When equivalence cannot be proved, it shows what parts of the graph could not be proved equivalent, so that the user can verify and test these parts manually.

Our use case is one of automated refactoring from machine generated code to human-readable code. The tool was tested on a large-scale refactoring process of legacy code performed by a partner company. We evaluate its efficiency, in terms of quality of output, both manually and automatically. Finally, we present best practices, lessons learned and limitations in designing and implementing such a tool. While our industrial data is confidential, we do include a reproduction package made of our handmade data and some open-source COBOL files[2]. We argue that the ideas and methodologies presented in this paper can be generalised beyond the specific context of our partner company. Demonstrating that a program retains its behavior after undergoing significant structural 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.

This project was made possible by the [redacted] funding of [redacted].

[1]The assumption to abstract away from statements makes sense in situations like our industrial case study where the refactoring rules applied focus mostly on changing program structure, and less on individual statements.

[2]Found on Zenodo at: https://zenodo.org/records/10037828

Special attention is given to explaining the rationale behind the design decisions we made. The clarity of these explanations is of key importance to facilitate understanding of our work by other researchers, should they wish to conduct similar investigations. We are transparent by explaining exactly what assumptions are applicable to our use case, so that others can determine if it would also be useful for their situation.

II. Problem statement

Many companies today are dealing with legacy software, making use of technologies that are either no longer known to freshly graduated developers, or third-party tools no longer supported by their vendors. This makes maintenance more challenging: not only does software looses quality as it ages, it becomes less inhabitable since its technology is less familiar. A solution is to refactor the entire code into something that becomes easier to maintain moving forward. When this task is delegated to external experts, code owners do not take part in the refactoring process, but only receive the refactored version of their code at the end of the process.

The partner company we are working with comprises experts in migrating legacy software. It offers several services including large-scale refactoring of legacy code. In this work, we studied their process of refactoring automatically generated COBOL code into more human-readable and maintainable COBOL.

The first step of such refactoring project is a consultation between the code owners (the client) and the migration experts. The refactoring process consists of an iterative and automatic application of code transformation rules that will restructure the code into something more readable as well as remove unwanted constructs and dead code. Some of those rules can be (de)activated to make the output best fit the needs of the client. Once the exact set of refactoring rules has been decided, the refactoring experts run their tool on the entire codebase and deliver the refactored code to their client.

Communication problems between experts and clients arise at several points in this process, mostly around understanding how the process works, why it does things in a certain way and whether or not the behaviour of the code remains equivalent after the refactoring. To address some of those issues, a variant of log-based behavioural differencing [11] was previously used. However, this does not provide sufficient guarantees that the code is correctly refactored. To increase the client's trust, until now, experts have performed only informal passes through selected files, making sure that the refactored code look "good enough".

One way for code owners to become more certain about the behaviour of the refactored code would be to run extensive test suites. If all tests that the original code passed still remain green for the refactored version, it should be relatively safe to deploy. However, in industry, test suites are sometimes completely absent, or lack sufficient coverage of the system under test. Manually creating new tests is expensive and time-consuming and while the discipline of automated software testing has seen a lot of interest, it cannot fully replace manual testing [4] nor has it been designed for legacy software that is not well-understood in the first place. Thus, in practice we witness code owners asking the refactoring experts to provide some kind of assurance that the two versions of the code are behaviourally equivalent and safe for deployment. Another frequent request from clients is to pinpoint critical parts of the code that they could test, but until now, refactoring experts are not able to give them full guidance.

III. Our solution

We now present our solution that compares COBOL programs before and after a refactoring. More specifically: (1) the models used to represent the code and how we obtain them, including the script to translate between those models; (2) the algorithm that compares those models, its limitations and how we overcame them; (3) a visualisation tool to highlight the differences.

A. Labelled Transition Systems (LTS)

To be able to perform trace equivalence between two programs, we transform the code into models. We use a tailored toolset to generate control flow graphs (CFGs), containing only the essential information in a program's control flow. To determine those essential elements, we consulted our partner company's refactoring experts and manually analyzed all of their 140 refactoring rules. Only 17 out of those 140 rules impact statements directly; furthermore, these 17 rules apply fairly "simple" changes, e.g. grouping variable declarations. Most of the refactoring rules are aimed at restructuring the code into something more readable, which is why we chose to abstract all statements away and focus on the control flow.

In the context of COBOL to COBOL refactoring, the most relevant constructs are therefore those that affect program structure: 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`"). As `PERFORM` and `GO TO` make use of COBOL's paragraph names, we include paragraph names in our CFGs as well.

SQL calls are found between `EXEC SQL` and `END-EXEC` keywords. SQL calls are also added to the CFGs because interactions with the database are critical: except for some prettifying, no query should be altered since the company's expertise is not in that domain.

However, the generated CFGs are not yet in the right format for trace equivalence. They tend to contain information both on the nodes and on the links and are destined mostly to be read by humans rather than to be explored by an algorithm. Therefore, we wrote a script to transform the CFGs into Labelled Transition Systems (LTS), a graph format used, amongst others, in model-based testing [31], by shifting all information from nodes to links. While this translation might seem cumbersome, it is necessary for our purpose: we not only seek the guarantees given by the trace equivalence algorithm, we also wish to visualise the results. For that, CFGs are preferred since they are more readable by humans.

LTSs have two types of links: (1) those with a specific label that are destined to be matched in the trace equivalence and (2) links that exist just to represent the flow of execution, denoted with an *INTERNAL* label. During 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.

Translating a CFG into an LTS requires to think about how to replicate the flow that will be followed by the execution of the program and translate 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 1 shows a small handcrafted COBOL code fragment, featuring most of the described constructs, and Figure 1 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 here to ease the reader's comprehension.

Listing 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.
```

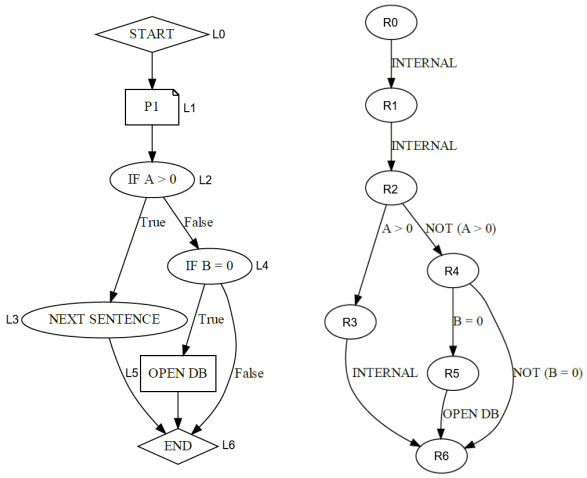

Fig. 1: CFG and LTS representation of the code of Listing 1.

The various `PERFORM` statements have to be carefully designed. `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 case of a `THROUGH` clause, one jump results in executing everything in paragraphs `A` and `B` and everything in between.

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). Note that the logic is the same for a `PERFORM THROUGH`, the placement of the *GO BACK* link being the only thing that changes. A CFG and corresponding LTS of this can be seen on Figure 2. 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...

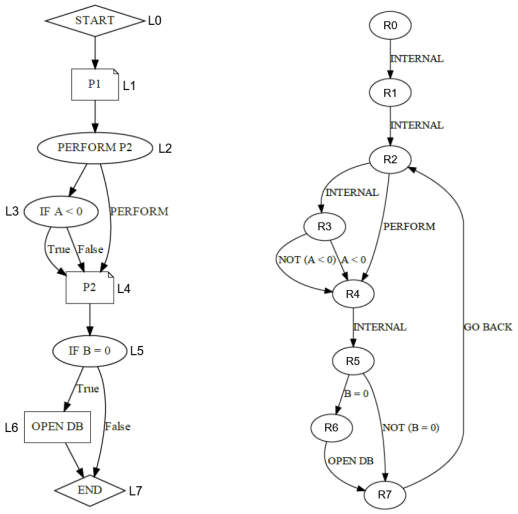

Fig. 2: CFG and LTS representation of a `PERFORM` statement.

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 3 (CFG) and Figure 4 (LTS).

B. Trace equivalence

The original algorithm for calculating trace equivalence between two graphs starts at the top (start) node of each graph and goes as follows:

1) Start at a corresponding node of each graph.
2) For each link of these nodes, look at each of their labels:
   - If it is *INTERNAL*, it can be skipped and the algorithm goes back to step 1 with the target node as start node.

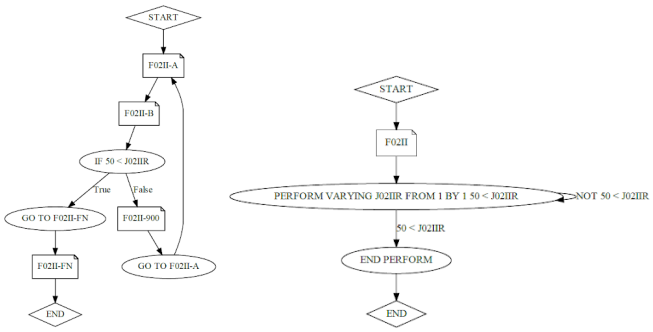

Fig. 3: CFGs for a piece of code pre- and post-refactoring.

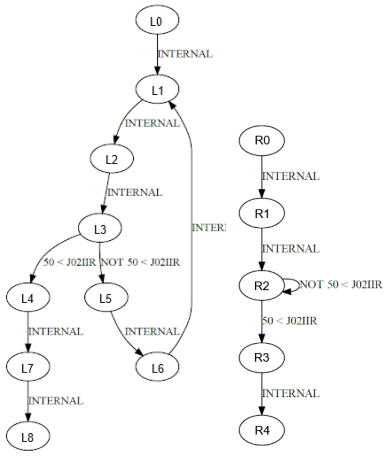

Fig. 4: LTSs corresponding to the CFGs of Figure 3.

- If it has a specific label, there must be a child on the other side having the exact same label.

3) When a link is skipped and/or matched, relaunch the algorithm recursively with its target node as start node. If no match can be found for a link, the path is invalid and therefore the graphs are not trace equivalent.
4) If a node with no out link (end node) is met or if a couple of nodes previously considered are explored again, the path we are following is valid.
5) If all possible paths are valid, the entire graph can be considered as trace equivalent.

Applied on the graphs of Figure 4, the trace algorithm will perform as follows (some steps are combined for conciseness):

1) The out links from nodes L0, L1 and L2 are skipped, meaning that node L3 is the next node. On the other side, the out links from node 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, but since we already matched them before, 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.

C. Handling conditions through semi-parsing

While this basic algorithm works for handcrafted examples, it lacks a few things to be able to handle our industrial use-case of COBOL to COBOL refactoring. The first thing missing is some awareness of how conditions work. E.g., when refactoring, in our use case 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 `NOT A < 0` and `A >= 0` are different.

This problem is even more prominent in COBOL because it supports contracted expressions (repeated variable names can be omitted), and some specific values are expressed with keywords (`ZERO`, `SPACES`, etc). The following list contains four variants of a condition that are all semantically equivalent but would not be detected as such by the base algorithm:

- `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.

Contracted expressions are very relevant in our case since the generated COBOL tends to shorten conditions, choosing not to repeat variable names and operators where possible, yet the refactoring produces expanded, more human-readable versions. This means that we encounter such false mismatches fairly often.

To address this issue, we wrote a linear semi-parser [34] focused on COBOL conditions only, allowing us to handle all cases mentioned above and more in a time-effective manner. It is based on a COBOL-compatible tokeniser and a parsing automaton shown in Figure 5. The semi-parser works in linear time and expands all contracted expressions to their full canonical form, allowing us to compare them later on. Our way of dealing with contracted expressions is an example of how we use semantics of the chosen language (COBOL) inside our differencing algorithm. A counterexample with some creative use of parentheses 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.

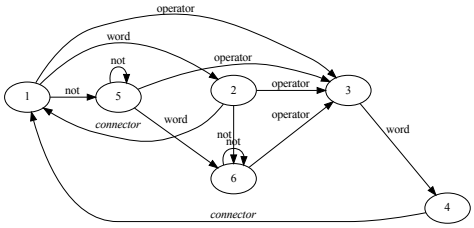

Fig. 5: The parsing automaton for contracted expressions: connectors are AND and OR, and operators are =, >=, etc.

D. Error recovery

Our trace equivalence algorithm also supports 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 equivalent to an empty statement. 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.

To handle such cases better, when no match is found for a link, the trace equivalence algorithm enters 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 labeled *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 afterwards) and the executions continues as before.

E. Special cases

Even equipped with the contracted expressions parser and the error recovery mode, the trace equivalence algorithm still occasionally produces unsatisfactory results. This is due to the fact that it needs to match links one to one, making the analysis of some more elaborate refactorings impossible. In particular, there are two specific cases that the COBOL generator tends to create and 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 2 (pre-refactoring) and Listing 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 code listings.

Listing 2: Nested IFs before refactoring.

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

Listing 3: Refactored nested IFs.

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

To handle this case, 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.

A 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 IFs are matched to the one from the EVALUATE. A concrete example of this refactoring pattern can be seen on Listing 4 (pre-refactoring) and Listing 5 (post-refactoring).

Listing 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.
```

Listing 5: Refactored successive IFs.

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

These structures are currently the only special cases that we have implemented. While other such cases could have been identified and implemented, the risk of making our tool less general may outweigh the benefits of having a few more matches.

F. Visualisation

To present the results of the trace equivalence algorithm to code owners, we created a visualisation tool using Roassal [3]. It is composed of two views: the first view, not shown here due to space limitations, shows an overview of the entire refactoring project, consisting of a list of all files that have been analysed, colour-coded to indicate the percentage of links that have been explored and detected as equivalent by our algorithm. This provides a general overall idea of how the refactoring went, and helps users to find the files that are most important to check, allowing them to order the files by most/least nodes matched or alphabetical order.

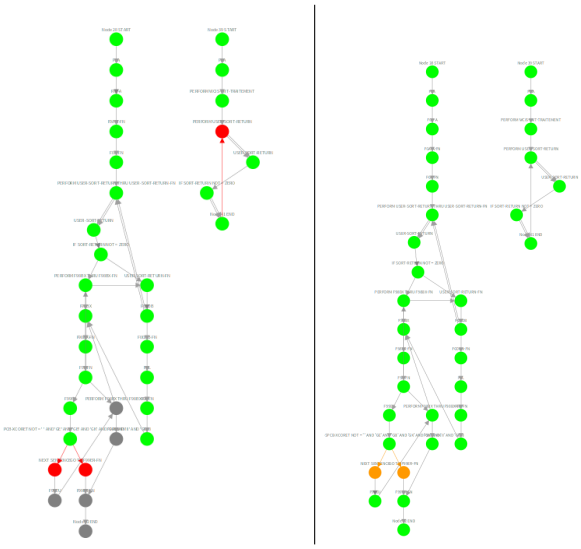

Fig. 6: Visualisation of the graphs from an example program before and after refactoring without (left) and with (right) error recovery activated.

A detailed view of any file can then be rendered by clicking on its name. Two CFGs are shown side-by-side, with colour-coding in green, orange, red and gray for links and nodes that were respectively marked as equivalent, unsure, non-equivalent and unexplored. Figure 6 shows the result of applying our trace equivalence algorithm to a small graph taken from our examples, running with and without error recovery.

On the left, we can see some red links and nodes, followed by gray unexplored ones, indicating where the algorithm stopped and considered the graphs nonequivalent. This is caused by an `IF` statement that was removed by the refactoring. In this specific case, there are unexplored nodes only on the pre-refactored version because the extraneous `IF` comes right before the end of the program, causing only a *GO BACK* link to be unmatched on the post-refactoring side.

On the right, we can see the same graphs after running with error recovery. The previously red elements on the pre-refactoring side are now orange because they were skipped (no node on the right-hand side had to be skipped, hence they are all green), and the rest of the nodes and links are green because the algorithm found a match further down.

Since our graphs tend to grow fairly large, we chose to group together nodes and links that are related to ease the reading. The result of applying this on the graphs shown in the previous figure can be seen on Figure 7. If possible, all green or gray (matched or unexplored) nodes that are linked together are merged into a single node, unless they have an incoming or outgoing link that is unmatched or skipped. This allows the user to focus on the parts of the code that may contain problems and gives a less cluttered view.

The goal of our tools is to be easy to use with a simple interface and well-known colour coding, as well as being actionable either by a refactoring expert or the code owner

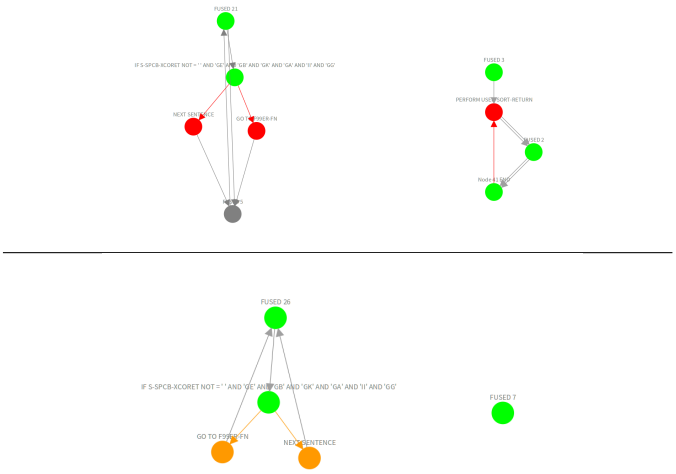

Fig. 7: Visualisation of the fused graphs from an example program before and after refactoring, without (top) and with (bottom) error recovery.

after receiving their 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 human developers, and are arguably more natural for them to interpret.

G. Summary

To summarise, we have presented our technique to extract CFGs from COBOL programs and then transform them into LTSs to use in our tools. We then detailed the base version of our trace equivalence algorithm, some of its limitations and our solutions to those limitations: semi-parsing of conditions, error recovery, and two implemented special cases. Consequently, whereas the overall approach to perform the differencing is largely language agnostic, some of these solutions rely on knowing the exact semantics of COBOL. Finally, we showed the visualisation tool that we created to analyse our results.

IV. Validation

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

A. 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 2'680 LOC, after refactoring it became 1'719; 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

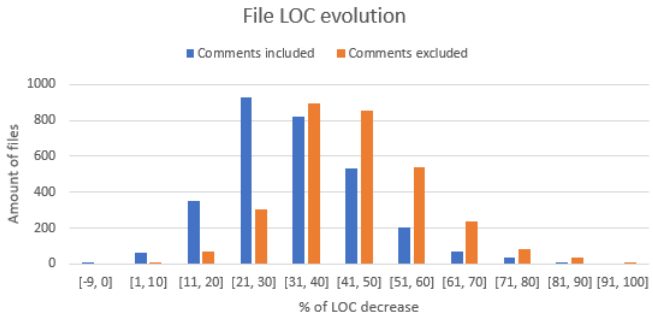

Fig. 8: Percentage of decrease in LOC in files after the refactoring, with and without comments included.

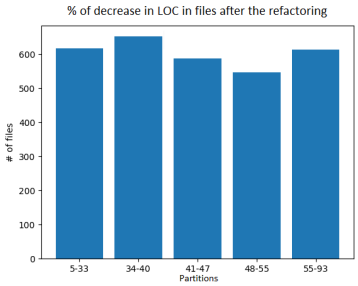

Fig. 9: 5 partitions for the files

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 diminution 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 8.

The most common diminution is of 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 indeed was 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 comments. Thus, we normalised the previous data, excluding comments from our LOC count to get the distribution shown in orange on Figure 8. The lowest amount of LOC diminution is now 5%, the highest 93%, and the most common range between 31% and 40%.

This allowed us to separate the files in five partitions regarding their percentage of diminution, each partition of more or less equal size, containing between 500 and 600 files each (see Figure 9). For our manual analysis, we then picked 5 files at random in each of these 5 partitions.

B. 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 analysed the output of the algorithm via logs from the trace equivalence, as well as using our visualisation tool. We used the unfused version of the graphs in our tool as much as possible to get the deepest and most precise analysis, but we did use the fused version whenever graphs were too big to analyse manually (over 250 nodes, which happened in 10 of the 25 selected files).

In the first phase, our first step was to check if the files could be shown to be trace equivalent. If so, we checked the visualisation to ensure that its output looked correct. 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 were 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. Due to space constraints, we are not able to detail all our observations in this paper, but we will discuss our most interesting findings and trends that emerged.

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 proven 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 our partner company: 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-factorisation 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.

C. Automated analysis

For our automated analysis, we ran the trace equivalence algorithm on all 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 error recovery, only 139 of the 3014 files were shown to be equivalent (before and after refactoring). This number increases to 902 with error recovery. Unfortunately, this still means that we can show full equivalence only in respectively 4.6% and 29.9% 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 version.

Figure 10 shows the amount of links explored by the trace equivalence 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 augmented in 2843 cases (94.3%), and the number of matched links augmented in 2128 cases (70.6%). We also noted that the number of links 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 error recovery on, 912 or 30.2% of the files have more than 40% of their links shown as equivalent, giving an indication that the refactoring was behaviour-preserving.

D. Threats to validity

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

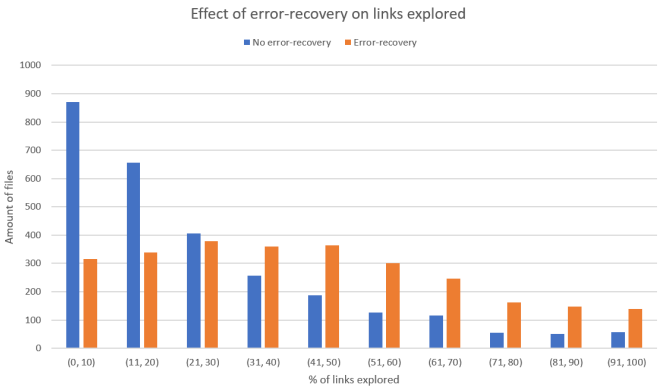

Fig. 10: Percentage of links explored by the trace equivalence algorithm with and without error recovery.

- 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, the refactoring experts could have been a better choice to act as proxy for their code owners. However, the researcher used its 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 supports the fact that code owner 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 generalizable to other programming languages or systems. However, we did pay special attention to make most parts of our tools 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 generalizability of the results.

V. Best Practices and Lessons Learned

This section presents the lessons we learned along the way while creating our trace equivalence and visualisation tools, the use case for which they are the best fit and some of the limitations of our tools, approach and methodology.

The technique is most suited to point out areas of concern in refactored code that should be tested more thoroughly before releasing it, rather than asking someone to blindly trust that the new code is behaviourally equivalent to the original one. 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. 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.

It is worth noting that undoubtedly we could increase further the raw percentage of files that can be shown to be trace equivalent, by implementing additional special cases in the algorithm, like we did in subsection III-E. But 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 lower our confidence in its capabilities.

We found that the trace equivalence algorithm excels when comparing structural changes, such as those shown on Figure 4. 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-case for our tool, which is the case for our case study, since the main focus of many of the applied refactorings is to improve code readability by removing many of the generated `GO TO` into more high-level human-readable control constructs. The technique we presented in this paper can be generalized further to any use case where proving equivalence between two programs that have been heavily restructured could be useful, which should be the case for other (automated) refactoring tools in other programming languages.

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 the migration company to pick the most relevant language constructs.
- *Understand the language*
  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), 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 that take the execution out of range of the current execution context, eliminate it.
- *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.

VI. Related work

A substantial body of academic research exists on refactoring of legacy systems with different objectives. Liu et al. [21] performed feature-oriented horizontal decomposition of the code, Bayer et al. [2] aim to produce product lines, Ribeiro and Borba [28] made a tool to recommend refactorings for an already existing product line, Raghavan [27] proposed to separate common interfaces from components, Chiang and Lee [7] moved towards distributed architecture by putting legacy code in wrappers, Kolb et al. [16] specifically wanted improved reusability of code elements without damaging their performance, Khatchadourian and Matsuhara [15] wanted to change the existing code to make full use of newly introduced language features, etc.

Some of this work, like ours, had differencing as their final goal, even though the actual comparison can be done very differently, as was the case for Ying and Miller [33] who tested the pre- and post-refactoring versions only for performance. One of the projects closest to ours was the one by Schuts et al. [29] who extracted Mealy machine models from a legacy system and its refactored version, in order to demonstrate their equivalence to their industrial partner.

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++ [12], Java [12], [26], [30], [32], JavaScript [30], Kotlin [18] and Python [1], [10].

Comparison of control-flow graphs was done by Bruschi et al. [5] in the context of detecting self-mutating malware, by Le and Pattison [19] for the sake of patch verification, Nagarajan et al. [24] as a countermeasure for heavy-flow obfuscation techniques. Lim [20] compared control-flow graphs on the level of entire blocks, which is a rougher version of our error recovery process. Comparison of labelled transition systems is an even older topic, well-investigated at least since the 1980s in the testing context [8].

Modern approaches are numerous and include variants of bisimulation and singleton failures metrics. For example,

Ordóñez-Camacho et al. [6] 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 context of operations languages used in the domain of spacecraft mission planning.

VII. Future work

As future work, we will improve our tools further based on the lessons learned from our validations. Although space limitations did not allow us to include it in this paper, we did perform a user validation with experts from our partner company. Their feedback was encouraging, confirming that the visualisation tool could be used to help guide their users to the parts that need to be tested. We could expand the custom cases of factorisation of `IF` and `IF/EVALUATE` to make the algorithm more flexible by ignoring links that do not interfere with the logical execution, or further implement special cases to improve our amount of matched nodes. We could improve our condition semi-parser to handle swaps in variable order, e.g. by comparing the different parts of a conditional expression as a set and not a string. We observed a few cases where equivalence checking failed due to a partial rename of variables. The algorithm could be improved to take such renamings into account. After having made all these improvements, we will compare the effect they have on the percentage of nodes matched and on our users.

In the context of our specific use case, where the company iteratively applies a series of refactoring rules to the original code, we could correlate for which files we fail to show equivalence with what refactoring rules were applied to refactor these files. From this we could learn what particular refactoring rules cause code changes for which it seems harder to prove trace equivalence. We could then ask the company to refine their refactoring rules to make them less complex or more structure-focused; or at the very least we could warn code owners when such refactoring rules were applied where the new code may require more intensive testing.

To demonstrate our work to be generalisable, we could apply or adapt the developed tools to another automated refactoring project, other use cases, or even other programming languages, to see if we can replicate the results shown in this paper.

VIII. Conclusion

In this paper, we introduced our use case of automated large-scale codebase refactoring and showed how it would benefit from tools allowing to show that the behaviour of a program is equivalent before and after the refactoring, helping code owners to trust the refactored version of their programs.

We start by extracting Control Flow Graphs from the code, using a semi-parsing tool. We extract only essential control-flow constructs from the code, as the refactoring rules mostly target such constructs. We then transform these CFGs into Labelled Transition Systems, taking care to respect the intricacies of the language semantics.

To compare these models, we proposed a trace equivalence algorithm that allows to match all possible paths composed of either labelled or "internal" links in two graphs given as input. We presented the algorithm, some of its limitations when applied to our industrial use case and how we customised it to overcome some of these issues.

While trace equivalence excelled when comparing programs of which the structure changed, the basic version of the algorithm did not perform well with variations in the conditions, leading us to develop a custom-made semi-parser focused on alleviating those limitations. Another limitation was the fact that the algorithm only handles matches of one link to another, rendering it inefficient when several constructs before refactoring are condensed into a single one by the automated process. We developed special cases to check equivalence for two kinds of refactorings that were common in the codebase we analysed: factorisation of `IF` conditions and translation of several `IF`s to a single `EVALUATE`. Finally, we implemented an error-recovery mechanism that allows the algorithm to skip a few links when it finds no match and continue when a match is found further on.

Since this work aims at helping code owners and refactoring experts, we also created a visualisation tool to help understand the output of our trace equivalence algorithm. It is meant to be easy to use with a simple interface and clear colors, as well as scalable with the size of our data by fusing similar nodes together. We used this tool in combination with the trace equivalence tool to validate our approach, both manually and automatically.

Our validation showed us that, even though our tools were not able to give certainty that the behaviour was preserved by the refactoring process in all cases, they still provide useful insights on where in the code further analysis or testing is needed, which is a question commonly asked to the company experts and that they were not able to answer before. Thus, our main achievement and contribution is not necessarily making a tool that matches any refactored CFG/LTS pair, but a tool that helps code owners to navigate their code and guides their attention towards code fragments that were refactored in the least trivial ways.

To conclude, we presented a tool that excels at structural differencing and showned the semantic-aware capabilities in our implemented extensions such as the condition parser and the custom handling of some special cases. We strike the delicate balance between detailed and efficient structural diffing enhanced with some semantic-aware matching based on the particularities of the use case and programming language considered. This allowed us to create a tool that remains sufficiently generic while highlighting points of attention to its users in those cases where equivalence cannot be shown.

This technique could be useful to help the analysis of other refactored code, although the extensions we created for the trace equivalence algorithm would have tio be adapted for the target programming language.

References

[1] 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. [Online]. Available: https://doi.org/10.1109/SCAM52516.2021.00025

[2] J. Bayer, J.-F. Girard, M. Würthner, 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*, ser. LNCS, vol. 1687. Springer-Verlag, 1999, pp. 446–463. [Online]. Available: https://doi.org/10.1007/3-540-48166-4_27

[3] A. Bergel, *Agile Visualization*. LULU Press, 2016. [Online]. Available: http://AgileVisualization.com

[4] 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)*. ACM, 2005, p. 571–579. [Online]. Available: https://doi.org/10.1145/1062455.1062556

[5] 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.

[6] D. O. Camacho, K. Mens, M. van den Brand, and J. Vinju, "Automated generation of program translation and verification tools using annotated grammars," *Science of Computer Programming*, vol. 75, no. 1-2, pp. 3–20, 2010.

[7] 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.

[8] R. De Nicola, "Extensional equivalences for transition systems," *Acta Informatica*, vol. 24, no. 2, pp. 211–237, 1987.

[9] S. Demeyer, S. Ducasse, and O. Nierstrasz, *Object-Oriented Reengineering Patterns*. Morgan Kaufmann and DPunkt, 2002.

[10] 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.

[11] M. Goldstein, D. Raz, and I. Segall, "Experience Report: Log-Based Behavioral Differencing," in *Proceedings of the 28th International Symposium on Software Reliability Engineering (ISSRE)*, 2017, pp. 282–293, DOI: 10.1109/ISSRE.2017.14.

[12] I. Hemati Moghadam, M. Ó. Cinnéide, F. Zarepour, and M. A. Jahanmir, "RefDetect: A multi-language refactoring detection tool based on string alignment," *IEEE Access*, vol. 9, pp. 86 698 – 86 727, 2021. [Online]. Available: https://doi.org/10.1109/ACCESS.2021.3086689

[13] 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)*. ACM, 2022, p. 1544–1554. [Online]. Available: https://doi.org/10.1145/3540250.3558954

[14] P. Julius and J. Fredrick, "Creating Habitable Code: Lessons in Longevity from CruiseControl," in *Proceedings of the 2009 Agile Conference*, Y. Dubinsky, T. Dybå, S. Adolph, and A. S. Sidky, Eds. IEEE Computer Society, 2009, pp. 259–264. [Online]. Available: https://doi.org/10.1109/AGILE.2009.21

[15] R. Khatchadourian and H. Masuhara, "Automated refactoring of legacy java software to default methods," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE. IEEE Press, 2017, p. 82–93. [Online]. Available: https://doi.org/10.1109/ICSE.2017.16

[16] 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," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 2, p. 109–132, mar 2006.

[17] 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," *IEEE Access*, vol. 8, pp. 125 755–125 775, 2020. [Online]. Available: https://doi.org/10.1109/ACCESS.2020.3007098

[18] 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.

[19] W. Le and S. D. Pattison, "Patch verification via multiversion interprocedural control flow graphs," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 1047–1058. [Online]. Available: https://doi.org/10.1145/2568225.2568304

[20] H.-I. Lim, "Comparing control flow graphs of binary programs through match propagation," in *2014 IEEE 38th Annual Computer Software and Applications Conference (COMPSAC)*. IEEE, 2014, pp. 598–599.

[21] J. Liu, D. Batory, and C. Lengauer, "Feature oriented refactoring of legacy applications," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE. New York, NY, USA: Association for Computing Machinery, 2006, p. 112–121. [Online]. Available: https://doi.org/10.1145/1134285.1134303

[22] T. Mens and T. Tourwé, "A Survey of Software Refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, 2004. [Online]. Available: https://doi.org/10.1109/TSE.2004.1265817

[23] E. R. Murphy-Hill and A. P. Black, "Refactoring Tools: Fitness for Purpose," *IEEE Software*, vol. 25, no. 5, pp. 38–44, 2008. [Online]. Available: https://doi.org/10.1109/MS.2008.123

[24] 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.

[25] W. F. Opdyke, "Refactoring Object-Oriented Frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992.

[26] 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.

[27] 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.

[28] M. Ribeiro and P. Borba, "Recommending refactorings when restructuring variabilities in software product lines," in *Proceedings of the 2nd Workshop on Refactoring Tools*, ser. WRT. New York, NY, USA: Association for Computing Machinery, 2008. [Online]. Available: https://doi.org/10.1145/1636642.1636650

[29] M. Schuts, J. Hooman, and F. Vaandrager, "Refactoring of legacy software using model learning and equivalence checking: An industrial experience report," in *Integrated Formal Methods*, E. Ábrahám and M. Huisman, Eds. Cham: Springer International Publishing, 2016, pp. 311–325.

[30] D. Silva, J. Silva, G. Santos, R. Terra, and M. T. Valente, "RefDiff 2.0: A multi-language refactoring detection tool," *IEEE Transactions on Software Engineering*, 2020. [Online]. Available: https://doi.org/10.1109/TSE.2020.2968072

[31] J. Tretmans, *Model Based Testing with Labelled Transition Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–38. [Online]. Available: https://doi.org/10.1007/978-3-540-78917-8_1

[32] N. Tsantalis, A. Ketkar, and D. Dig, "RefactoringMiner 2.0," *IEEE Transactions on Software Engineering*, 2020.

[33] M. Ying and J. Miller, "Refactoring legacy ajax applications to improve the efficiency of the data exchange component," *Journal of Systems and Software*, vol. 86, no. 1, pp. 72–88, 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121212002129

[34] V. Zaytsev, "Formal Foundations for Semi-parsing," in *Proceedings of the IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*. IEEE, Feb. 2014, pp. 313–317. [Online]. Available: https://doi.org/10.1109/CSMR-WCRE.2014.6747184