A Scalable Log Differencing Visualisation Applied to COBOL Refactoring

Céline Deknop[1,3], Kim Mens[1], Alexandre Bergel[2], Johan Fabry[3], Vadim Zaytsev[4]
[1]Université catholique de Louvain, Louvain-la-Neuve, Belgium
[2]Department of Computer Science (DCC), University of Chile, Santiago, Chile
[3]Raincode Labs, Brussels, Belgium
[4]Universiteit Twente, Enschede, The Netherlands
{celine.deknop, kim.mens}@uclouvain.be, abergel@dcc.uchile.cl, johan@raincode.com, vadim@grammarware.net

***Abstract*—Large code refactoring projects can consist of hundreds of refactoring rules that are applied iteratively to make code easier to maintain. Visualising the refactoring process can help engineers and stakeholders understand how chains of refactorings were applied and to gain more confidence in the produced result. An apparently suitable existing visualisation using log-based behavioural differencing suffers from scalability issues when applied to industrial-size cases. We propose an adapted visualisation tool that highlights those parts that really changed in-between iterations of a large refactoring process and collapses those parts that remain stable. We show that our alternative visualisation scales well on large logs of a process with many possible refactoring chains, of which significant parts are shared. Consequently, it allows engineers and stakeholders to quickly answer relevant questions about what happened during the refactoring process.**

***Index Terms*—Differencing, Visualisation, Logs, Refactoring, COBOL, Scalability, Industrial**

I. Introduction

The discipline of differencing, i.e. analysing changes between two versions of an artefact, is arguably well known to anyone working in computer science. To obtain such a difference, various techniques can be used on different kinds of inputs. In our context of migrating large codebases to a more maintainable version, we are interested in visualising differences between variants of a refactoring process. For this, we explored the technique of *log-based behavioural differencing*, which extracts behavioural information from logs produced by a process, to then allow for visual inspection of the differences between two variants of that process. We used the algorithm and naïve visualisation proposed by Goldstein et al. [15], and applied it to an industrial refactoring project and use case from the company Raincode Labs.

Raincode Labs is an independent compiler company that provides services for migration and modernisation of legacy systems. To help Raincode engineers compare two variants of a refactoring process that are configured differently, we applied Goldstein's algorithm and visualisation to their logs. The goal of the visualisation is to provide insights on the effects of changing the configuration, by highlighting the differences between the two execution logs. We found that while this naïve visualisation works well on small examples, it suffers from scalability issues that make it unsuitable to apply on the actual logs of an industrial refactoring project. This is due to the large size of the graphs encountered (up to hundreds of nodes) and to their specific nature of often extremely long linear chains of nodes (which are of less interest to the engineers).

On the positive side, the nature of these graphs provides interesting opportunities for compressing the relevant information contained in them. We therefore set out to optimise the visualisation tool so that it better fits the nature and size of the graphs, by collapsing less important points (such as the long chains), using colours to highlight key differences, and using line and node sizes to indicate the importance of certain nodes and edges. This paper presents the result of our efforts.

More specifically, the project kind to which we apply our visualisation is called *PACBASE migration* [28]. PACBASE is an aging fourth generation language [38] that allows engineers to use concise macros to generate COBOL code instead of developing in COBOL directly. PACBASE support having ended in 2015, reliance on PACBASE has turned into a liability for companies. Ideally the language would be retired, and companies could maintain the generated COBOL code. Yet, the COBOL code generated by PACBASE is arguably not readable by humans, and rewriting it from scratch is not feasible in practice. Raincode's PACBASE migration refactors PACBASE-generated COBOL code to human-readable COBOL using a set of refactoring rules that is applies iteratively on the codebase (more on this in Section IV).

The exact set of refactoring rules to apply to a COBOL portfolio is a configuration that has to be determined in collaboration with the customer. This task is difficult: there are 140 rules available in total, and understanding their exact effects and interactions requires expert knowledge that the customer does not have. We see this difficulty as an opportunity for a differencing tool to show how a change in the configuration influences the overall refactoring. To help Raincode engineers in guiding the customer through the rule-selection phase of the refactoring project, we developed such a tool to visualise the influence of activating or deactivating a refactoring rule.

We validated this new visualisation tool by applying it to an actual PACBASE migration project and verifying that the produced visualisation indeed highlights the things we intend it to, that it remains sufficiently small to be human readable, and that the algorithm is sufficiently fast to calculate the visualisation. Finally, we performed a user study with Raincode engineers to collect feedback on the suitability of the tool and ideas for future improvements.

To summarise, the main contributions of this paper are:

- the analysis and use of the *log-based behavioural differencing algorithm* of Goldstein et al. [15] and the highlighting of some of its limitations when applied to log files of a specific nature;
- an improvement of the visualisation proposed by Goldstein et al., by post-processing the output of their log differencing algorithm to improve the readability of the visualisation;
- an identification of an industrial scenario of refactoring large COBOL codebases where this visualisation can significantly improve practitioners productivity;
- a validation of the visualisation on such an industrial scenario with industrial practitioners.

The paper is structured as follows. Section II presents the log-based differencing algorithm of Goldstein et al. and its visualisation. Section III discusses the limitations of the existing algorithm and visualisation when applied to our industrial use case and introduces our improved visualisation. Section IV validates the work by showing the new visualisation tool at work and by analysing its adequacy and performance. Section V discusses related work and Section VI concludes the paper and presents avenues for future improvements.

II. Existing Algorithm and Visualisation

In this section we present Goldstein et al.'s work on log-based behavioural differencing [15] and comment on its relevance to our industrial use case, before explaining the algorithm in more detail and showing its results.

A. Motivation

Since Raincode engineers wish to help their customers understand why they should or should not pick 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.

With that in mind, we took interest in Goldstein et al.'s *log-based behavioural differencing* algorithm [15]. The goal of this algorithm 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?

The goal of this algorithm is 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 analyzed.

For example, in the context of analysing the use of an online store, p_1 could represent an old version of the website, while p_2 would represent a new version with more modern features. The goal of the differencing algorithm is then to highlight what new paths the client is following. This could then be analysed to find out if the navigation is still clear.

The proposed approach takes textual log messages representing the behaviour of p_1 and p_2 and turns each into a Finite-State Automaton (FSA). The two resulting automata are then compared with the differencing algorithm, and changes are highlighted using a visual representation.

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.

B. Log-based behavioural differencing

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 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 an online store. 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 contain the timestamps and the page visits, while dropping the rest of the data to ensure having the simplest possible model. Figure 1 shows a truncated example of the normalised log files obtained this way.

FSA models are then extracted from these normalised logs using the *kTails* [8] algorithm[1]. Many tools applying this algorithm already exist, and Goldstein et al. chose Perfume [6] 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 [18]. The file representing a 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.

[1]A detailed discussion of what *kTails* is and why it was chosen can be found in Goldstein et al.'s paper [15] .

```
Creditentials Page 13:53:37.281
LoggedIn Homepage 13:54:27.841
Search Page 13:55:02.182
AddItem Page 13:55:57.283
...
Checkout Page 14:15:29.724
Logout Page 14:16::37.528
```

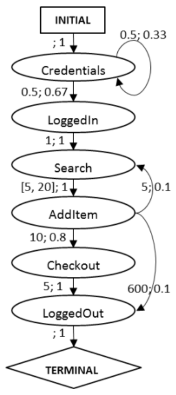
(a) Log file p_1

```
Creditentials Page 10:32:02.356
PswReset Page 10:33:05.628
LoggedIn Homepage 10:35:18.724
Search Page 10:35:02.827
AddItem Page 10:36:03.924
...
Logout Page 10:50:12.656
```

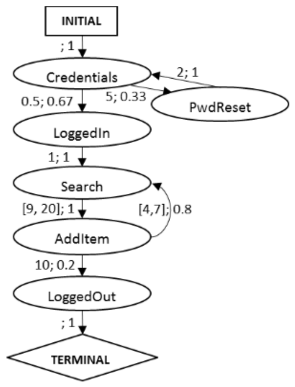
(b) Log file p_2

Fig. 1: Log files for two executions of the shopping process, example from [15]

Figure 2 shows a graphical example of such a FSA for each of the two logs of our shopping cart example. The first word on each log line is used as node ID, subsequent lines represent transitions from one node to a 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.



(a) FSA for log file p_1



(b) FSA for log file p_2

Fig. 2: FSA graph representation of both logs to compare

Once both FSA models have been calculated, we can compute the difference 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.

We now describe a simplified version of the algorithm used to compute the difference between two FSA models, redirecting to Goldstein et al. [15] for details. First, the common paths starting from the initial node and ending at the terminal node are computed. A path is common between two models if all of its nodes have the exact same label. *Common nodes* are defined as those having the same label and being part of at least one common path. There is also a special case: a node is considered common if it has a label that is unique to its model and happens in both compared models. In our online store example, all nodes labels are unique for each model (no nodes are duplicated), but the nodes `PwdReset` and `Checkout` are not present in both models, and therefore not common. *Added nodes* are nodes that are present in the second model and not common, while *removed nodes* are present in the first model but not in the set of common nodes. The diff contains all the common, added and removed nodes, along with all the edges from the second model, and the edges from the first model that connect the removed nodes to the rest of the graph.

C. Visualisation

The last step is to generate the visualisation. As shown in Figure 3, Goldstein et al.'s visualisation stays relatively close to the Graphviz images of the input FSA models. Nodes are differentiated by their border style: common nodes have solid borders, removed nodes have a dashed border and added nodes a double solid border. The edge label keeps the same structure with the execution time, a semicolon and then the transition probability; however, it now shows the evolution of these values denoted by an arrow from the old value to the new one. Consider for example the edge from node AddItem to Search, we can see that the execution time went from 5 in p_1 to a value ranging between 4 and 7 in p_2, and that the transition probability increased from 0.1 to 0.8.

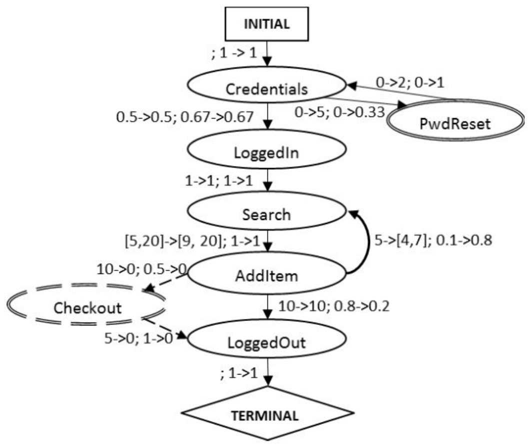


Fig. 3: Result of log differencing for the shopping process

III. Application of the Algorithm to our Industrial Case

A. Direct application

To apply the algorithm described in Section II to our industrial setting, the first step is to decide what to extract from the available logs. Since we are interested in modeling 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 a refactoring is something that is already available internally at Raincode but very rarely used in practice.

To create the FSAs, we followed the method of Goldstein et al. and found a tool based on the *kTails* algorithm. By 2021, Perfume has become abandoned and impossible to run, so we used a substitute called Synoptic [7], created by the same team. With our FSA models generated, we then reimplemented the model differencing algorithm described in the previous section using Pharo [4], [27]. The latter choice was mainly motivated by its companion visualisation framework Roassal [3], [5].

Concerning visualisation, we stayed relatively close to what was proposed in Goldstein et al.'s original paper (as shown in Figure 3), with nodes representing rules linked together by arrows representing the probability to execute 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 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 size of graphs as it would not fit on a single page or screen.

B. Limitations of the visualisation

It becomes clear from looking at Figure 4, that our data seems to be of a different nature than what was presented by Goldstein et al. [15]. 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 refactoring 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

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

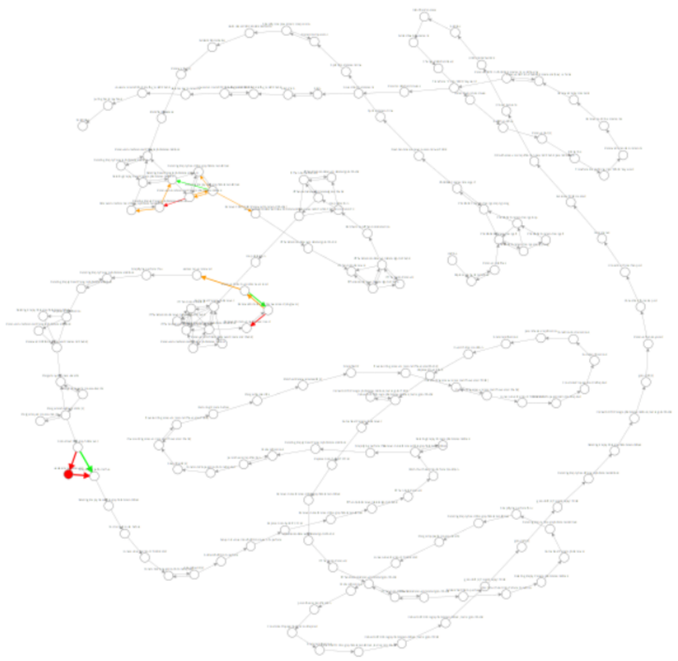


Fig. 4: First differencing output for one program

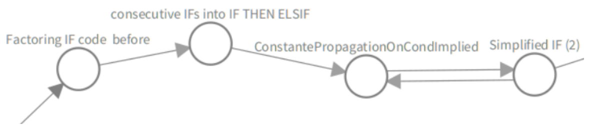

consecutive IFs into IF THEN ELSIF
Factoring IF code before
ConstantePropagationOnCondImplied
Simplified IF (2)

Fig. 5: A chain of nodes with no changes in the transition probabilities

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. 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 a 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.

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 6a. The second kind of change, like the one depicted in Figure 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 kind of structures we would like to be highlighted when looking at our visualisation.

C. Adapting the visualisation

Based on these observations, we decided to address the readability issue by creating a merge algorithm to apply after the computation of the original differencing algorithm. The merge 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 applied iteratively throughout the graph until there are no valid nodes left to merge. Two linked nodes are merged if they fit the following criteria:

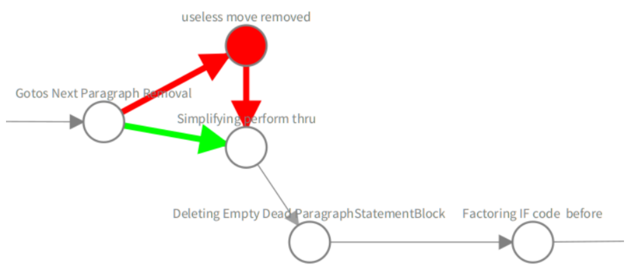
- Both nodes are common (neither added or removed, i.e. not in green or red);
- No transition edge from or to those nodes is added or removed;
- No transition edge to or from those nodes has seen a probability change between the two variants of the process (i.e. in orange).

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

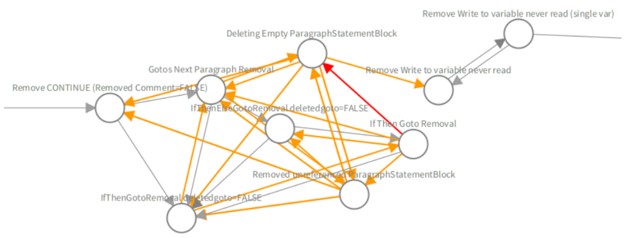
```
merge(graph){
    do {
        // Keep track of whether or not we merged
        beingMerged = false
        // Keep track of visited nodes
        visitedNodes = new Array()
        // Start at the top
        mergeRec(graph.initialNode(), visitedNodes)
    } while(beingMerged)
}

mergeRec(currentNode, visitedNodes){
    // Merge anything that we can in the current node
    foreach(child : currentNode.children()){
        if(canMerge(currentNode, child)){
            mergeNodes(currentNode, child)
            beingMerged = true
        }
    }
    //Add the current node to the visited ones
    visitedNodes.add(currentNode)
    foreach(child : currentNode.children()){
        if(child not in visitedNodes){
            //Recursively go through each unexplored node
            mergeRec(child, visitedNode)
        }
    }
}
```

Fig. 6: Two types of clustered changes


useless move removed
Gotos Next Paragraph Removal
Simplifying perform thru
Deleting Empty Dead ParagraphStatementBlock
Factoring IF code before

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


Remove Write to variable never read (single var)
Deleting Empty ParagraphStatementBlock
Gotos Next Paragraph Removal
Remove Write to variable never read
Remove CONTINUE (Removed Comment=FALSE)
IfThenElseGotoRemoval deletedgoto=FALSE
If Then Goto Removal
Removed unreferenced ParagraphStatementBlock
IfThenGotoRemoval deletedgoto=FALSE

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

The merge 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 would be collapsed into one single merged node, while only the two last nodes of Figure 6a would be merged, leaving the *Simplifying perform thru* node visible because it is has two incoming links: one removed and the other added.

Recall the initial example of Figure 3: our algorithm would not merge 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 merged nodes visually by showing them in gray rather than the white for the other common nodes. As label, 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 merged node’s size is proportional to the amount of nodes it contains, enabling a more “at-a-glance” analysis.

Our merging algorithm helped us shrink the graph from an average of 120 to 30 nodes (more in-depths metrics are presented in Section IV). As described, only plain white unchanged nodes were merged, 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 merged node’s internal details can still be inspected in a new window when clicking on it. As an example, the result of our merging algorithm on the graph of Figure 4 can be seen in Figure 7.

IV. Validation

In this section we first detail the process of a migration project at Raincode. We then present a few metrics regarding the results produced by our visualisation tool. Second, we analyse the execution time of the model creation and the

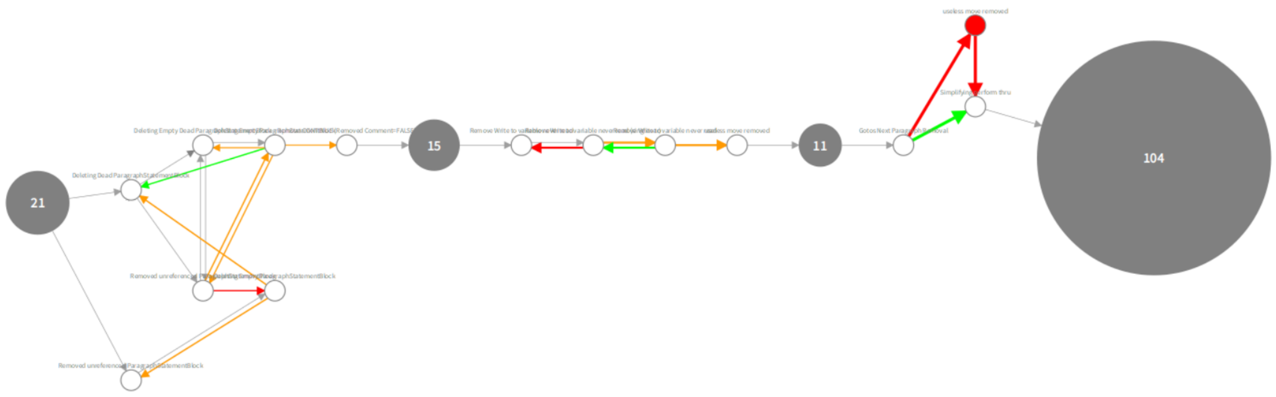
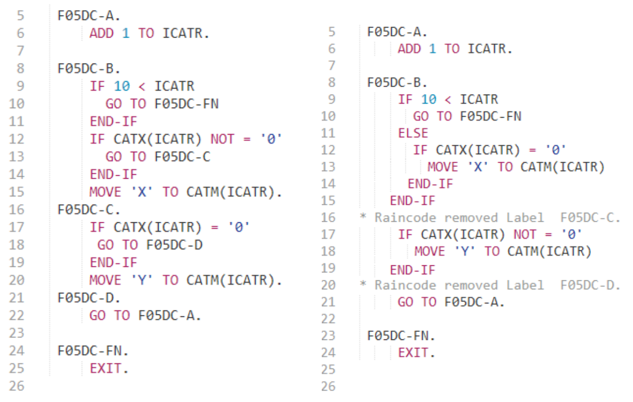

useless move removed
Simplifying perform thru
Deleting Empty Dead Paragraph
Removed Comment=FALSE
Remove Write to variable never read
useless move removed
Gotos Next Paragraph Removal
Deleting Dead ParagraphStatementBlock
Removed unreferenced ParagraphStatementBlock
Removed unreferenced ParagraphStatementBlock
21
15
11
104

Fig. 7: Adapted visualisation after merging nodes, for the same input as Figure 4

differencing and merge algorithms. Third, we discuss the size of the input logs and graphs and compare the size of the resulting graphs, for both the original and merged graphs. Fourth, we present how we validated our visualisation tool with two Raincode engineers involved in PACBASE migration projects, and analyse the results of this user study.

A. *Migration projects at Raincode*

A detailed description of the different steps of a PACBASE migration project is given by Deknop and al. [11]; we present only an illustration here. The migration is performed by iteratively applying a set of rules, each refactoring the code just a bit, as exemplified in Figure 8.



```
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) Before applying rule

```
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) After applying rule

Fig. 8: Snapshots of the migration process

Even though Raincode has a basic “one size fits all” configuration of rules, their customers often prefer a personalised experience. The first step of a migration project is thus to pick and choose the exact set of refactoring rules that will be used to transform their code. The engineers present the different rules to the customer, who can choose to activate them or not. Since a plain description of a rule is often too abstract for the customer, a small portion (around 30) of the customer’s programs is used as an example on which the effect of the rules is demonstrated. The chosen set of rules is applied to these programs, and the output is presented to the customer so that they can verify if the refactored results fit their company’s standards. A few iterations of this may occur, adding and removing rules until the customer is fully satisfied.

Once the set of rules has been chosen, a second preliminary step starts: the customer has to be convinced that the refactoring process is safe and will not introduce any bug. For this, Raincode creates an illustrative subset of the refactored files so that all the rules that got triggered during the complete process are represented. Note that some rules selected by the customer might not have been triggered at all, simply because their preconditions were never met. The customer then tests these files thoroughly to ensure that the refactoring rules did not break anything.

After this, the project enters the main refactoring phase where all artifacts to be transformed are sent to Raincode, pre-analysed and processed. This phase takes on average two weeks and is concluded by a *delivery*. During this time, the customer may keep editing some of the files that are being refactored. When receiving the transformed codebase, the customer would then want the additional changes they made in parallel to be integrated as well.

This triggers a new phase of the migration project, called a *redelivery*. The customer sends to Raincode the new versions of the files that have been edited so that they can be refactored as well. Again, the files are pre-analysed, processed and sent over. This last step is repeated as many times as needed.

B. *Use of the visualisation tool during the migration process*

We identified two steps in the migration project where we believed that our visualisation tool would be useful: during the very first rule-selection step, as well as during redelivery. We expand only on the first use case here for brevity.

The goal of this first step is for the customer to tailor the refactoring rules to their preference and company’s coding style. When presented with all the rules, they will consider some rules as absolutely necessary while feeling less strong about others. It is the role of Raincode's engineers to help guide the customer's choices. However, while a rule's drawbacks or benefits can be clear by looking at it in isolation, it is harder to assess the impact of a rule in the context of a full migration, when combined with many other rules.

The refactoring process being iterative in nature, the execution of a rule A might be a precondition for a rule B to be fired. Deactivating rule A could thus essentially deactivate rule B as well: even though it is still in the chosen set, it might never get triggered during the process. While the relationships between some rules are fairly clear, even for Raincode engineers it is impossible to predict all interactions between all 140 rules. This can be problematic if a customer really wants a specific rule to be executed, without realising that changes to other rules might affect it, thus failing to understand why the rule is no longer executed.

This is where our visualisation tool comes in: the idea is to take the logs of two variants of the process on the small set of programs, each variant using a slightly modified set of selected rules. Armed with the diff graphs along with the knowledge of what changed in the rule-set between the two variants, the customer can better visualise the possible effect of the changes made to the rule-set. For example, it would be directly visible if the removal of one rule had an impact that was bigger than expected. With this knowledge, the customer can then make a more informed decision about what modifications to make next to the set of selected rules. The tool helps Raincode engineers in their guidance by giving them a more visual support, but also because it can provide them with concrete examples of why a rule should be left activated (or not) when it has a high impact on other rules or even on the entire migration process.

C. *Metrics*

Since Raincode did not have an ongoing migration project at the time of writing, 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 Section II) 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.

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 merge 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 grows 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 I) to only 119 nodes in our input graphs.

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

TABLE I: 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 amount of nodes by just a bit, from an average of 119 to one of 123. As illustrated by Figure 4, this amount of nodes is overwhelming to be visually analyzed. The second column of Table II shows the improvements gained by the merge algorithm.

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

TABLE II: Graph size in nodes

Merging reduces the average amount of nodes by around 75%, resulting in much more readable graphs. The minimum graph size consisting of a single merged 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 merged node representing no changes.

D. *Interview with the engineers*

Given the size and style of the company, only two persons are working on PACBASE migration at Raincode. One of them is more customer-oriented, having as main responsibility to assist in sales and to 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 background. 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.

1) Interview set-up: Since we had only two engineers available for our study, we conducted a semi-structured interview with each of them individually (a full interview taking around 45 minutes).[3] Before the interview, they received a one-page manual describing each component in our visualisation tool

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

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 their own suggestions along with their opinions. Our accompanying repository [12] 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.

2) Methodology: To structure the presentation of our results, we took inspiration from the work of Sillito et al. [29]. 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 in 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);
- Questions about the **meaning** of a visual component (they either asked us directly or used the one page manual to look for the response);
- 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 merging algorithm is an improvement over the non-merged graphs we asked them direct questions about this aspect.

3) Results of the study: Due to space considerations, we cannot include all results of the study in this text. We refer to the file `InterviewCommonQuestions.csv` in our accompanying repository [12] for all questions highlighted in the following descriptions. We limit ourselves to presenting the most significant items here.

a) 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?"*

b) 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?"*

c) 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?"*

d) 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.

e) Open questions: For the open questions we asked our participants, we were mostly interested in knowing whether they perceived our merged graphs as more useful than the full graphs. For this, the participants were requested to open a non-merged version of a graph of which they had previously analysed the merged version. We then asked them questions regarding the similarity and differences of the information shown in this graph with respect to the previous one.

4) Analysis of the results:

a) Domain: A 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 therefore are 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, Raincode's refactoring process has two levels of logs: a more generic level concerning the triggering of a rule (but that can be applied at multiple places in the code), and a more low-level one that represents the application of a specific rule to a specific paragraph in the code. Both levels can be analysed with our tool, but in the specific instance of this interview, for the sake of time we were only looking at the, more complex, lower level. Both engineers thus needed to know the exact level of detail we used as input for the tool in order to proceed 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.

b) Tool usage: We observed that the most performed actions were moving nodes and zooming into the graphs. This may suggest that despite our merge 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's 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.

c) 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 to varying probabilities, causing both engineers to need clarification of this description. Second, the use of the gray 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.

d) Future work: As described previously, in our specific use case our tool can help analysis at two levels. While we chose to have our validation only showing the most fine-grained level due to time constraints, another round of interviews on the 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.

As for feature requests, we want to highlight two that seemed quite interesting. P_C would like to have, attached to a merged 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".

e) Open questions: We focus here on the comparison between the non-merged and merged graphs. When shown a non-merged version of a merged graph they had analysed before, we asked the engineers if, at a first glance, this was something they saw 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 merged node would be sufficient for his needs. When asked if they thought that the merged graphs were an improvement over the non-merged ones, they both agreed that the merged graphs were a strict improvement. They also said that no features from unmerged graphs would be missed.

5) Conclusions: To conclude, we observed that, even if they had fairly different points of view due to their background, both engineers used the tool in a very similar fashion (excluding idiosyncrasies of who prefers scrolling and who prefers dragging): when presented with our visualisation, they look at the graph from left to right, stopping on clusters of change and guessing at what might have cause 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 reflecting 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. P_T however, 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, this to ensure nothing gets obsolete.

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.

Finally, we want to note that our merge 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 merged graphs.

V. Related Work

The visualisation proposed in this paper was based on Goldstein's log-based behavioural differencing algorithm. Other papers presenting algorithms that perform differencing in specialised or advanced ways, though a rare find, still exist. The one closest to our current interest is Kim and Notkin's *LSdiff* [20] (Logical Structural DIFFerencing), an approach aiming at representing structural changes in a very concise manner, focusing on allowing the developer to understand the semantics of the changes. However, that approach seems more suited for object-oriented code, and less for our COBOL use case. Other approaches exist that focus on the object-oriented paradigm, such as cal-cDiff [2] and Diff-CatchUp [36].

Other tools like REdiffs [17] or Ref-Finder [19] focus on untangling changes due to refactoring from other changes made to functionalities. Whereas such tools could prove useful in the context of analysing a redelivery, in our industrial migration process it would require explicitly encoding all 140 possible refactoring rules and maintain the tool as they evolve. Our approach, in contrast, is agnostic of the particular refactoring rules as it looks at the logs only.

Since we are not differencing code but models, we also took interest in work happening in the software modelling community, and studied tools that perform clear and efficient differencing on specific kinds of models. Many of those exist for widely-used models like UML (e.g., UMLDiff [35]), activity diagrams (e.g., ADDiff [25]) or feature models (e.g., in FAMILIAR [1], [32]). Witnessing the abundance of many different tools for each kind of model, Zhenchang Xing [34] also proposed an approach to allow for a more generic way to difference models. Here again, most tools focus on the object oriented paradigm, though we did take inspiration from the different visualisations these propose.

We also explored different techniques used when performing *data* differencing. From the starting point of the Hunt-McIlroy algorithm treating simple text, to its extension to treat binary data [33] when there is need of differencing more heterogeneous artefacts. Afterwards, many different and modern techniques were developed, including those based on control flow graphs [21], program dependence graphs [16] and other tools making use of ASTs or at least parse trees as with GumTree [13] or cdiff [37]. We are also exploring the idea of enriching the initial data format with infrastructures such as *srcML*, and how it can be applied to differencing [24] using its corresponding tool *srcDiff* [10], [31]. While we agree that some of those techniques could be useful in the next stages of this research project, in this paper we wanted to focus on the logs to provide new insights to the users of our tool.

There is some work on log differencing as such, mostly focused on business processes or on relating actual log entries to logging code that produced them: Li et al [23] have investigated log diffing with the intention to detect unwanted duplicate log messages, such as those stemming from cloning logging code without appropriate adjustments. They detected 5 different patterns of harmful log message clones, and developed a tool called DLFinder [22]. Gholamian and Ward [14] develop 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 [30] 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 ours is the paper by Bolt et al. [9], 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 [26] which in software engineering would be analogous to grammatical inference.

Since Goldstein's work came closest to what we needed, we decided to build our own visualisation on top of that algorithm.

VI. Conclusion

In this paper, we presented an improvement to an existing algorithm to perform log-based behavioural differencing [15], in order to improve its scalability. This is because when applying the original algorithm to an industrial refactoring process, we found significant limitations in the original visualisation. To overcome those limitations, we extended the algorithm with a merging phase and implemented a new visualisation. We then validated that log-based behavioural differencing could indeed be applied to our industrial use case and that our merge algorithm is an improvement on the output from the original algorithm. Our extension is compatible with any textual logs, just as the original. However, to benefit the most of our merging algorithm, the logs need to be of a significant size and the process itself needs to have many possible executions chains: it should be able divert for a bit then join again later, in order to create the clusters of changes we described.

Our improved algorithm reduces the output size of the original algorithm by 75%. This allows for an easier analysis, while not hiding any information to the user, as confirmed by both industrial participants of our user study. Regarding efficiency, our algorithm adds only a minimal overhead to the computation time: for our industrial case of 39 logs of around 282 lines each, the original algorithm takes 0.77 seconds and our extension adds 0.58 seconds. This is assuming that the FSAs models have already been generated (our extension did not impact the generation of those models).

The industrial user study confirmed that both participants would be interested to use our tool in their specific context: respectively for explanation support and for debugging during an industrial COBOL migration project. Moreover, they confirmed that the tool might prove useful and that it provides them new insights they cannot get otherwise.

For future work, fine-tuning of the layout would be the next step, as well as having a more user-friendly interface for the creation of the models and the differencing itself. We would also like to conduct another round of validation at the higher level of abstraction available in Raincode's logs. Finally, more validation with a higher number of participants and on other processes (like our online shopping example, or other examples such as traffic navigation) would help validate on what kinds of processes our tool is most useful.

Acknowledgements

We thank Raincode Labs for its support and Raincode's migration engineers Yannick Barthol and Boris Pereira for their collaboration as our validation subjects. We also thank Innoviris the research funding agency of the Brussels-Capital Region, for providing the funds necessary to conduct our CodeDiffNG Applied PhD research project. We are also grateful to Lam Research and the ANID FONDECYT Regular 1200067 for partially sponsoring the work presented in this paper. Final thanks go out to Benoît Duhoux for proofreading this paper.

References

[1] 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)*, ser. LNCS, vol. 7328. Springer, 2012, pp. 629–645, DOI: 10.1007/978-3-642-31095-9_41.

[2] T. Apiwattanapong, A. Orso, and M. J. Harrold, "A Differencing Algorithm for Object-Oriented Programs," in *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE)*. IEEE, 2004, p. 2–13, DOI: 10.5555/1025115.1025202.

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

[4] A. Bergel, D. Cassou, S. Ducasse, and J. Laval, *Deep Into Pharo*. Square Bracket Associates, 2013. [Online]. Available: http://books.pharo.org/deep-into-pharo/

[5] A. Bergel *et al.*, "*Roassal homepage*," Online: http://agilevisualization.com/, 2021.

[6] I. Beschastnikh *et al.*, "*Perfume frontend GitHub*," Online: https://github.com/ModelInference/perfume-frontend, 2021.

[7] ——, "*Synoptic GitHub page*," Online: https://github.com/ModelInference/synoptic, 2021.

[8] A. W. Biermann and J. A. Feldman, "On the Synthesis of Finite-State Machines from Samples of Their Behavior," *IEEE Transactions on Computers*, vol. C-21, no. 6, pp. 592–597, 1972, DOI: 10.1109/TC.1972.5009015.

[9] 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," *Information Systems*, vol. 74, pp. 53–66, 2018, information Systems Engineering: selected papers from CAiSE 2016.

[10] M. Decker, M. Collard, L. Volkert, and J. Maletic, "srcDiff: A Syntactic Differencing Approach to Improve the Understandability of Deltas," *Journal of Software: Evolution and Process*, vol. 32, no. 4, p. e2226, 10 2019.

[11] 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)*, M. Morisio, M. Torchiano, and A. Jedlitschka, Eds. Springer, 2020, pp. 270–286, DOI: 10.1007/978-3-030-64148-1_17.

[12] C. Deknop, "Validation data on github," Online: https://github.com/CelineDknp/PACBASEValidationData, 2021.

[13] 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, p. 313–324. [Online]. Available: https://doi.org/10.1145/2642937.2642982

[14] 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*, ser. SAC. ACM, 2020, p. 82–91.

[15] 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.

[16] A. Hamid and V. Zaytsev, "Detecting Refactorable Clones by Slicing Program Dependence Graphs," in *Post-proceedings of the Seventh Seminar in Series on Advanced Techniques and Tools for Software Evolution (SATToSE 2014)*, ser. CEUR Workshop Proceedings, vol. 1354. CEUR-WS.org, 2015, pp. 37–48.

[17] 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.

[18] E. G. John Ellson *et al.*, "*Graphviz homepage*," Online: https://graphviz.org/about/, 2021.

[19] 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, DOI: 10.1145/1882291.1882353.

[20] M. Kim and D. Notkin, "Discovering and Representing Systematic Code Changes," in *Proceedings of the 31st International Conference on Software Engineering (ICSE)*. IEEE, 2009, p. 309–319, DOI: 10.1109/ICSE.2009.5070531.

[21] J. W. Laski and W. Szermer, "Identification of Program Modifications and Its Applications in Software Maintenance," in *Proceedings of the Eighth Conference on Software Maintenance (ICSM)*. IEEE Computer Society, 1992, pp. 282–290, dOI: 10.1109/ICSM.1992.242533.

[22] Z. Li, "Characterizing and Detecting Duplicate Logging Code Smells," in *Companion Proceedings of the 41st International Conference on Software Engineering (ICSE), Student Research Competition*, 2019, pp. 147–149.

[23] 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.

[24] J. I. Maletic and M. L. Collard, "Supporting Source Code Difference Analysis," in *Proceedings of the 20th International Conference on Software Maintenance (ICSM)*. IEEE, 2004, pp. 210–219.

[25] S. Maoz, J. O. Ringert, and B. Rumpe, "ADDiff: Semantic Differencing for Activity Diagrams," in *Proceedings of the 19th Symposium on the Foundations of Software Engineering and the 13rd European Software Engineering Conference (FSE)*, T. Gyimóthy and A. Zeller, Eds. ACM, 2011, pp. 179–189.

[26] 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*, J. C. Trujillo, K. C. Davis, X. Du, Z. Li, T. W. Ling, G. Li, and M. L. Lee, Eds. Cham: Springer, 2018, pp. 449–459.

[27] Pharo consortium, "*Pharo homepage*," Online: https://pharo.org, 2021.

[28] Raincode Labs, "PACBASE Migration: Flexible Process," https://www.raincodelabs.com/pacbase/, 2021.

[29] J. Sillito, G. C. Murphy, and K. De Volder, "Asking and answering questions during a programming change task," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 434–451, 2008.

[30] B. A. Tama and M. Comuzzi, "An Empirical Comparison of Classification Techniques for Next Event Prediction using Business Process Event Logs," *Expert Systems with Applications*, vol. 129, pp. 233–245, 2019.

[31] G. de la Torre, R. Robbes, and A. Bergel, "Imprecisions diagnostic in source code deltas," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR. Association for Computing Machinery, 2018, p. 492–502, dOI: 10.1145/3196398.3196404.

[32] S. Urli, A. Bergel, M. Blay-Fornarino, P. Collet, and S. Mosser, "A visual support for decomposing complex feature models," in *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)*, 2015, pp. 76–85.

[33] Z. Wang, K. Pierce, and S. Mcfarling, "BMAT – A Binary Matching Tool for Stale Profile Propagation," *Journal of Instruction-Level Parallelism*, vol. 2, pp. 1–20, 06 2000. [Online]. Available: http://www.jilp.org/vol2/v2paper2.pdf

[34] Z. Xing, "Model Comparison with GenericDiff," in *Proceedings of the 25th International Conference on Automated Software Engineering (ASE)*. ACM, 2010, pp. 135–138.

[35] 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.

[36] ——, "API-Evolution Support with Diff-CatchUp," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 818–836, 2007.

[37] W. Yang, "Identifying Syntactic Differences between Two Programs," *Software Practice & Experience*, vol. 21, no. 7, p. 739–755, Jun. 1991, DOI: 10.1002/spe.4380210706.

[38] V. Zaytsev and J. Fabry, "Fourth Generation Languages are Technical Debt," International Conference on Technical Debt, Tools Track (TD-TD), 2019, extended Abstract, http://grammarware.net/text/2019/4gl-techdebt.pdf.