Visual Assurance in Refactoring through Trace Equivalence of Control Flow Graphs

Céline Deknop*†, Johan Fabry†, Kim Mens*, Vadim Zaytsev‡
*ICTEAM, UCLouvain, Belgium; †Raincode Labs, Brussels, Belgium
‡Formal Methods & Tools, UTwente, The Netherlands
Email: kim.mens@uclouvain.be, johan@raincode.com, vadim@grammarware.net

***Abstract*—Refactoring large legacy codebases, even with industrial-strength tools, often leads to trust concerns with code owners, in particular when the codebase underwent significant changes. To provide more assurance to code owners, we integrate visual analytics into the refactoring process. This method involves transforming code into control flow graphs before and after refactoring, followed by trace equivalence analysis on these graphs. An innovative visualisation tool provides not only a comprehensive overview of the refactorings' impact across all files, but also offers detailed insights into the trace equivalence at individual file level. By presenting clear visual evidence of code equivalence before and after refactoring, our visualisation narrows the trust gap, offering refactoring experts and code owners a transparent and understandable view of the changes. We apply this visualisation on an industrial use case and discuss its effectiveness with refactoring experts.**

***Index Terms*—visualisation, trace equivalence, large-scale code refactoring, control flow graph, industrial validation, COBOL**

I. Introduction

Although many automated refactoring tools have been proposed [1]–[3], developers are reluctant to use such tools [4], [5]. One reason cited as an obstacle to the use automated tools is that they are a black box and therefore their output is not trusted. We propose a tool to compare two versions of a program (before and after refactoring) and show whether the control flows of both programs are trace equivalent. Our visualisation tool then shows both a summary of the trace equivalence for every refactored file and highlights details in every equivalence graph. This allows a developer to either get an overall idea of the effect of the refactoring process or to manually analyse the differences in both versions of a single file. We apply these tools to an industrial use case and discuss the results of the validation we performed.

II. Goals of the Tool

One of the services that Raincode Labs offers is to refactor generated COBOL programs into maintainable and human-readable code [6]. For this, they use an automated tool that iteratively applies around 140 different small refactoring rules, increasingly improving the code. Although this refactoring process has been perfected for 15+ years, in-house experts find that code owners struggle to trust the refactored code. While code owners are involved in the early stages of the projects to determine the configuration of the refactoring tool, the process itself behaves as a black box to them. Since the code tends to get refactored substantially, it is hard for them to recognise and trust the refactored code. For this reason, the in-house experts could benefit from a tool allowing them to facilitate communication with their clients and provide them a visual support that explains in detail the effects of the automated refactoring [7]. Previous work has been done in the direction of specifying how to make refactorings behaviour-preserving in diverse contexts [8]. Comparing two versions of a program through trace equivalence to prove that their behaviour remains the same, is also not novel [9]. However, we take it a step further by creating a visualisation tool to highlight where and why the refactored code is not equivalent.

We established two main goals for our visualisation tool. The first is to be able to offer a comprehensive global overview of a refactoring project, showing at a glance the status of all refactored files within the project. This overview needs to be scalable and usable with thousands of files (the scale of our industrial data). The second goal is to provide users with a detailed visualisation of every analysed file individually. This detailed view includes graph models of the code before and after refactoring. The emphasis is on making these visualisations as informative as possible. Colour coding is used to highlight important elements in the graphs, making it easy for tool users to identify where and why the code was considered equivalent or not. The graphical layout of the graphs strives to emphasise key information effectively. Uninteresting nodes and links are hidden whenever possible. Finally, the visualisation is interactive to further facilitate the exploration and analysis process.

III. Tool architecture

The tool is divided in two parts: comparing models of the code through trace equivalence (III-A) and visualising the results (III-B) to enable a more detailed analysis.

A. Trace equivalence

A detailed explanation of the trace equivalence algorithm is beyond the scope of this paper. In summary, the algorithm takes as input two graph models. The graphs are composed of nodes that represent key code structures found in the (refactored) programs, and links representing the flow of

This project was made possible by the AppliedPhD funding of Innoviris.

execution of these programs. For example, a node representing an `IF` statement would have two outgoing links corresponding to both the condition and its negation.

The trace equivalence algorithm explores two graphs, trying to match their *traces* (or paths). If all paths in the graph representing pre-refactored code have an equivalent path in the post-refactored graph, the program's control flow behaviour was preserved. In its basic version, the algorithm provides a binary answer: the models are trace equivalent or they are not.

To create a more fine-grained answer, the basic algorithm is modified to record every step of the trace equivalence algorithm. We start by marking the transitions between the states. Before starting the trace equivalence, every transition is marked as 'unexplored'. During its execution, the algorithm marks the transitions. If a link has an equivalent in the other path, it is marked as 'matched'. If there is no equivalent for a link, we allow the algorithm to skip up to X links (configurable) to look for another match further down the path. If a match can be found that way, the link is marked 'skipped'. If no match is found, the link is marked as 'unmatched'.

The graphs obtained from our industrial data, are cyclic. To handle cycles and achieve as much precision as possible in our visualisation, we simply continue to add marks as long as they are needed whenever we meet a transition again. This means that any transition can be marked with all of the four "base marks": 'unexplored', 'matched', 'unsure' or 'matched', or a combination of the last three (the 'unexplored' mark is always removed when the algorithm explores a node).

B. Visualisation

Our visualisation tool consists of two views: an *overview* and a *detailed* view. Figure 1 depicts the *overview* of the entire refactoring project. On the left-hand side are two fields used to specify the files that will be shown. Clicking on the *visualise* button will populate the right-hand side. This left panel also allows the user to order the files displayed on the right: alphabetically or by most/least links marked as matched.

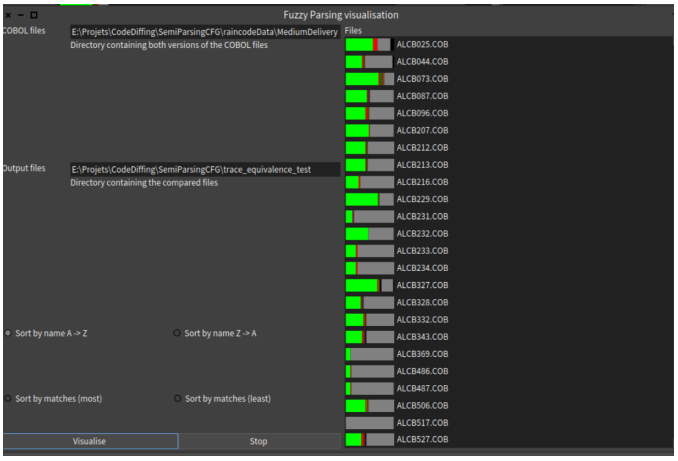

Fig. 1: *Overview* of the trace equivalence visualisation tool.

On the right-hand side of the *Overview* are the files from the refactoring process. For each file, we display both its name and a summary of the result of the trace equivalence analysis. We calculate the percentages of every match type over both models, and display a bar divided into areas of colours corresponding to their percentages.

To obtain the *detailed view*, users simply select a file to visualise from the list on the right. This opens a new window showing the two control-flow graphs corresponding to the file pre- and post-refactoring side-by-side (pre-refactoring on the left, post- on the right). Figure 2 shows the *detailed view* on small example of a file being refactored, for a simple case where the models are trace equivalent. To avoid clutter, we denoted nodes that can result in the end of the execution with a thick black border (instead of a link towards an *end* node).

Fig. 2: *Detailed view* of a small COBOL file.

Our visualisation tool is interactive: when nodes get selected in one model, matching nodes in the other model get highlighted. When trace equivalence explores execution paths, it does so in both models simultaneously. In essence, it creates groups of nodes that are matched together on either model, i.e. nodes from the pre-refactoring graph are matched with nodes on the post-refactoring graph and vice-versa. We distinguish *perfect matches*, states that are exactly the same, from *group matches*, which are nodes that have been explored together but cannot be linked specifically to a corresponding node in the other graph.

When presented with a detailed view, a user can interact with it by hovering the mouse over different nodes and links. For a node, it shows its label, e.g. *PERFORM F94FT*. For a link, it reminds the user of the meaning of its colour (e.g., for all green links, tooltips are *'matched'*). It also shows the *match* or *match group* for a node. If nodes are highlighted in light blue, they do not have an exact equivalent in the other graph (*group match*). If they are highlighted in dark blue, they are a *perfect match* to each other. Examples of group match (left) and perfect match (right) are shown in Figure 3.

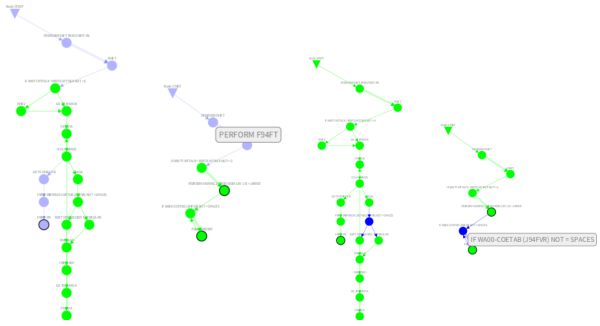

Fig. 3: Highlighting of a group match (left) and perfect match (right) on the example of Figure 2

While the example shown so far is fairly small in size to help in understanding the visualisation tool (40 LOC and no more than 20 nodes for the larger pre-refactoring model), larger files quickly encounter scaling issues when visualising their *detailed view*. For example, Figure 4 is an example of a full-size file displayed as-is in our tool.

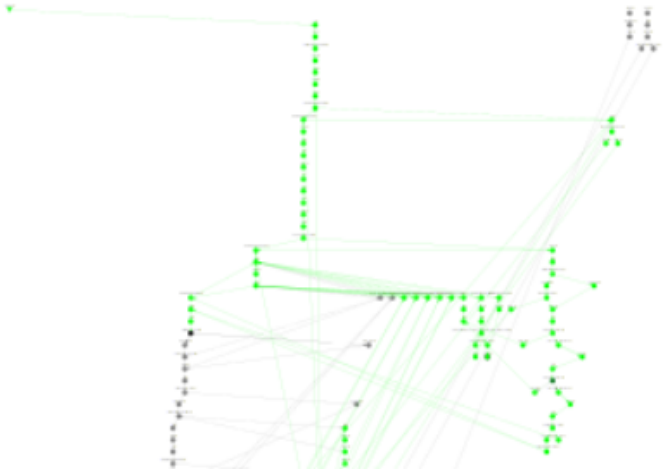

Fig. 4: *Detailed view* on a pre-refactored version, no post-process (image cut off due to lack of space).

To address this scaling issue, we created a post-process algorithm to 'fuse' together nodes that are either matching (green) or unexplored (gray). Two contiguous nodes of a same colour are grouped if they do not have any links that are marked either unmatched or skipped. This fusing algorithm allows us to greatly reduce the amount of nodes displayed, as illustrated in Figure 5 (which is the fused version of Figure 4).

IV. Industrial validation

A. Industrial context

Raincode Labs performs various services, COBOL refactoring being only one of them. The company is fairly small, around 40 employees. Because of this, only two employees are in charge of the COBOL refactoring projects we described in this paper. Both were instrumental in defining the goals of the tool as well as performing the industrial validation. One of them (referred to as P_C) was more Customer-oriented, focused

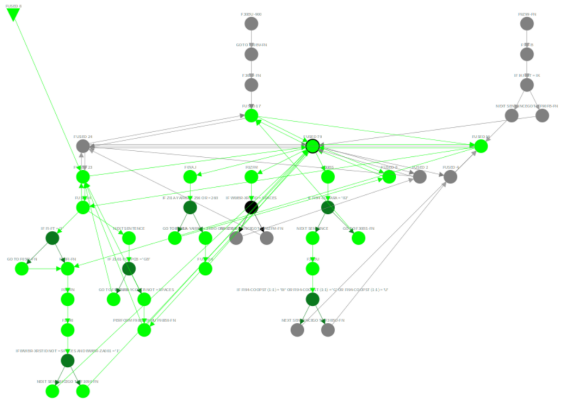

Fig. 5: Detailed view of Figure 4, with post-process.

on communication with the clients to configure the refactoring process, while the other one (P_T) is more Technical, the main developer of the refactoring process itself.

B. Interview set-up

In the first phase of our validation interview, we started by contextualising the tool. We listed the code structures that we extracted from the COBOL code and how we arranged them into control-flow graphs. We told the participants how we compared graphs using trace equivalence, giving an intuition of the algorithm. We explained that they would use a visualisation tool to understand the result of the trace equivalence, and prompted them to read the tool's brief user manual.

During their use of the visualisation, we asked them to "think aloud" in order for us to collect as much feedback as possible from the interview. We recorded both the screen of the computer used and the sound of our conversation.

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

In the third phase, we turned fusing *on* and let the participants browse freely in files taken from a previous refactoring project. We concluded the interviews by going over a set of pre-defined questions and asked them if they had any further remarks or feedback to give (a video is available at https://zenodo.org/records/10162365).

C. Methodology

To structure the results of our user validation, we take inspiration from Sillito et al. [10]. We used the vocal description of the actions performed by our participants to analyse not only the answers they gave to our questions, but also the questions that arose from them. We divide our observations in four categories:

1) Questions about the **domain** (input, output, inner representation of data in the tool);

2) Questions about **technical execution** (related to how we extracted things from the COBOL files or how we computed the execution paths);
3) Questions about the **meaning** of a visual component (they either asked us directly or used the user manual to look for the answer);
4) **Ideas** for future work (when they asked if some feature would be possible to add or implement).

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

D. Results of the validation

We highlight here the most relevant feedback that we received.

1) Domain: We had only very few domain questions, P_C being the only one to ask during phase 2: *"Are you sure the graphs are not flipped? Since the right graph is bigger than the left, I would expect that one to be the pre-refactoring"*. P_C's confusion about which graph corresponds to which version of the program is easily answered: while it is true that the amount of nodes found in the model before refactoring is always larger than the amount of nodes after refactoring, this is no longer true in the visualisation when the fusing algorithm is applied. This answer satisfied him.

We can explain the low amount of domain questions by two factors. First, our validation participants are experts in their field and therefore very familiar with the domain. Second, the fact that we took time to contextualise how we obtained the visualisation that was presented, helped in avoiding most questions regarding the input or output of our tool.

2) Technical execution: P_T had two technical questions. One referred to the constructs that we extracted from COBOL: *"Do you handle the periods?"*; the other was about our handling of the way COBOL works *"Can you deal with the fact that a* `PERFORM` *comes back, but only if the execution does not jump outside of its boundaries?"*

The respective orientation of both participants explains why only P_T had technical questions. When P_C looked at our visualisation, he saw exactly what he expected and assumed that we did our due diligence. For P_T, since he has been working on the automated refactoring process for years, he is deeply aware of the challenges of COBOL and wanted to make sure that we did as well so he could trust our tool. We answered both his technical questions satisfactorily, allowing him to move forward in the validation with more confidence.

3) Meaning of visual components: P_C had a question regarding the meaning of the large border denoting and ending node : *"Does this mean that the execution* always *stops there, or that it* can *stop there?"*. We explained that, if the node had no outgoing link, it meant that the execution always ended while if the node had (at least) one outgoing link, it meant that the execution could end there. This answer satisfied P_C and could be included in a new iteration of the user manual to avoid all confusion.

P_T felt overwhelmed when reading the user manual and when first interacting with the full graphs; he had to refer to the manual several times to remember the meaning of colours, although he did not ask us direct questions. However, after a few minutes of using the tool, he was more confident and did not express further confusion.

4) Ideas of future work: P_C had two ideas about *How to use the tool at a smaller scale so it is less overwhelming to analyse its output?"*. First, he expects the output to be more accessible if we focused not on the entire automated refactoring process, but on the effects of a single refactoring rule. It is possible for Raincode Labs to run the process with a very limited amount of rules activated, and P_C thinks this visualisation could help convince a client that a specific rule will not alter the behaviour of the code. The ability to say that an automated process showed the two versions to be equivalent would strongly benefit P_C's interactions with clients.

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

During his interview, P_T conducted a detailed analysis of a node flagged as unmatched. His analysis started in our tool. He opened the first file, scrolled to a red (unmatched) node on the pre-refactoring side and decided that he wanted to understand why it was marked as not matching. He picked a red node and was able, using our graph, to quickly find the point of interest in the corresponding code files. He came to the following conclusion: the refactoring process completely deleted the IF statement because it was simply *doing nothing* (his wording). This entire process, from picking the node to analyse to opening the files and coming up with the answer, took him around five minutes.

Indeed, the flagged node was an `IF` statement that only contained jumps, both leading the execution to the same point, meaning that the computation of the statement did not, in fact, perform anything useful. This is why the statement simply got deleted in the refactored version of the code.

After this analysis, P_T asked why our tool was flagging this as unmatching. We explained that, due to the partial nature of our models, we have no way to know that the if statement did not contain any other statement than the jumps, which is why we flag it. He then asked us *"Why not implement a few more or the specifics of our process in order to not have so many things flagged as unmached?"* and we explained our reasoning for not explicitly wanting to reimplement all refactoring rules in our tool, in order to keep it more generic. He was not fully satisfied with this answer, arguing that he would be interested in seeing at least this case implemented and the effect it would have on the output since he felt this specific situation would occur frequently.

5) Open questions:

- Both participants agreed that the general design of the tool was adequate. They also agreed that there was a lot of information displayed on the screen, but that this was

unavoidable due to the nature of the data and that the fusing algorithm did help make the graphs more readable. P_C stated that he always encounters such limitations when wanting to visualise graphs of this nature and size.
- Both participants said that the amount of nodes remaining unexplored made them wary to show these graphs to clients. They fear that showing a graph with 10 to 40% of the nodes unexplored would stress the client even more. However, P_C would consider using this at a smaller scale as discussed under *Ideas of future work* earlier on.
- Our participants' opinions diverged about the final results. P_C was not surprised by the results and said: *"I know how complex our process is and I did not expect that it would be possible to show trace equivalence for the entirety of the programs"*. P_T on the other hand had expected us to include more of the specificities of the automated refactoring in our tool since they feel so natural to him. He therefore expected our number of matches to be higher and stated that he thinks implementing more special cases would not introduce too much bias. As long as we do not copy the code he wrote for the refactoring, he feels that if two different persons arrive at the same result, it is a form of verification.

E. Discussion

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

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

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

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

While implementing more special cases must be done with caution, we believe that it is possible to do so without jeopardising the generalisability of the tool, at least for the case described above. For example, we could tweak our semi-parser to denote when it ignores part of the code, indicating the presence of some statements that it did not parse. With this, we could detect that the removal of the `IF` statement analysed by P_T is indeed behaviour-preserving. An implementation of this is a good avenue for future work with this tool, as would be a deeper analysis (helped by P_T) of the migration rules to identify other candidates for special cases.

V. Conclusion

We started this paper by arguing that automated refactoring tools are often considered as black boxes and therefore not easily trusted by code owners. To overcome this issue, we set out to create a tool to compare two versions of a codebase, before and after refactoring, allowing to show that its behaviour remains unaltered by refactoring and therefore augment trust.

The tool we created provides two views: one summarising the state of the entire codebase being refactored, and a detailed view focused on a single file. We model a program before and after refactoring and compare them using the technique of trace equivalence, that we extended to mark our models with the information needed for our visualisation.

Our tool was intended to scale to industrial data—thus, we made sure that its users would easily find what they are looking for through a summary for every file. We also made the graphs in the detailed view interactive, helping experts in their analysis. We also created a post-process fusing algorithm to emphasise the most relevant information only. Finally, we evaluated if the tool we created attained the objectives we established. To do so, we validated our tool on industrial data and obtained feedback from two refactoring experts. The validation participants reacted positively to the tool we designed, although some more work would be required for it to be used on concrete projects with actual customers.

References

[1] H. K. Wright, D. Jasper, M. Klimek, C. Carruth, and Z. Wan, "Large-Scale Automated Refactoring Using ClangMR," in *ICSM*. IEEE, 2013.

[2] G. Szóke, C. Nagy, R. Ferenc, and T. Gyimóthy, "Designing and Developing Automated Refactoring Transformations: An Experience Report," in *SANER*. IEEE CS, 2016, pp. 693–697.

[3] S. Thompson and H. Li, "Refactoring Tools for Functional Languages," *Journal of Functional Programming*, vol. 23, no. 3, p. 293–350, 2013.

[4] Y. Golubev, Z. Kurbatova, E. A. AlOmar, T. Bryksin, and M. W. Mkaouer, "One Thousand and One Stories: A Large-Scale Survey of Software Refactoring," in *ESEC/FSE*. ACM, 2021, pp. 1303–1313.

[5] M. Wyrich and J. Bogner, "Towards an Autonomous Bot for Automatic Source Code Refactoring," in *BotSE*, 2019, pp. 24–28.

[6] C. Deknop, J. Fabry, K. Mens, and V. Zaytsev, "Improving Software Modernisation Process by Differencing Migration Logs," in *PROFES*. Springer, 2020, pp. 270–286.

[7] C. Deknop, K. Mens, A. Bergel, J. Fabry, and V. Zaytsev, "A Scalable Log Differencing Visualisation Applied to COBOL Refactoring," in *VISSOFT*. IEEE, 2021, pp. 1–11.

[8] T. Mens, S. Demeyer, B. Du Bois, H. Stenten, and P. Van Gorp, "Refactoring: Current Research and Future Trends," *ENTCS*, vol. 82, no. 3, pp. 483–499, dec 2003.

[9] T. Wood and S. Drossopoulou, "Program Equivalence through Trace Equivalence," *Foundations of Object Oriented Languages, FOOL*, 2014.

[10] J. Sillito, G. C. Murphy, and K. De Volder, "Asking and Answering Questions during a Programming Change Task," *IEEE TSE*, 2008.