Response to the Reviewer's Comments

Paper: FEVER: An Approach to Analyze Feature-Oriented Changes and Artefact Co-Evolution in Highly Configurable Systems

July 3, 2017

Dear Editors,

We thank the reviewers for the care and effort that was put in their remarks. This is well appreciated. We provide here responses to all remarks and comments. We start with the general remarks, and continue by responding to each individual comments.

1 Common comments

While we address comments from reviewers separately, let us first address the main comments.

> 1. Please discuss the impact on generalizability of the choice to rely entirely on kconfig and ignoring "Source" statements in kconfig.(R3)

To address this point, we updated Section 8.1 on the limitations of the file-based differencing approach: *This limitation has practical implications on our work. For instance, knowing that we do not consider cross artefact relationship (such as "source" statement in Kconfig files), what we observe are changes done locally to features - at a file level. As a result, certain interpretation of the changes are not possible. For instance, based solely on FEVER data, one cannot identify how the available configurations of the Linux kernel evolved. This requires an understanding of how the entire set of Kconfig files and features has changed - this amounts to semantic differencing. FEVER captures textual changes performed by developers.*

We provide further information on the generalizability of the approach as answers to Reviewer 3 comment (R3.C5).

> Please provide details of the manual analysis that was used to answer RQ1 and RQ2. (see R3)

We address this point in our reply to Reviewer 3 (R3.C6).

> R3 points out that a different ground truth was used in this paper compared to the original MSR paper. Please explain why this was done and address the other questions R3 has with regard to this decision.

While comparing the results obtained during the first experiment (in the MSR submission) and the extension, we found out that we had made some mistakes in the initial evaluation. Some of the extracted changes had been considered correct when not, and vice-versa. For the evaluation we considered that we should evaluate FEVER against the "most accurate" ground truth rather than the one used during the MSR experiment.

To compensate for this, we performed the second evaluation in a different manner, with two rounds (presented in the "Threats to validity" section, as a threat to "internal validity"). We adjusted our paper to reflect this in the discussion section:

Using manual analysis for validation purposes is inherently fault prone. The difference in terms of content of the samples used for the replication of our initial study highlights this. For instance, we had identified 208 added features during the initial study, but only 206 during the replication - over the same set of commits, and therefore the same set of changes. While in some cases (e.g. for file-feature mapping), the differences can be explained by a better ability to track some changes and therefore we simply have more information, in other cases, this is due to human error when reviewing the content of commits. For the evaluations performed in this work (both the replication and the new evaluation), the manual review of commits was performed twice - for the entire dataset, leaving a small time gap (between 2 days and a week) between the two evaluation rounds. While the errors identified in the initial evaluation lead to a significant update for some change attributes (namely "added feature references" in the code), evaluation errors occurred in less than 5% of the commits. Throughout this two step evaluation, we still observed more than 80% of the commits being matched perfectly in the FEVER database. This increases our confidence in the overall validity of our results.

R2 thinks the paper is in acceptable shape, but believe it could be better and asks for additional discussion in a few places including:

- how can the tool alleviate issues
- what is the cost of misclassification
- what details do you have about the few drops in recall
- which categories does FEVER do better or worse on than others

We provide information on those points in our replies to Reviewer 2 (comment R2.C1 to R2.C4).

2 Answers to the remarks of R1

R1.C1: I had some problems to localise the new heuristics used by the authors in the improved versions of FEVER. I kindly ask the authors to mention such changes not only at pg. 24 at the evaluation of FEVER, but also in section 3 when they describe the model. They can, for example, explicitly say something like "this heuristic improved the FEVER model of Dintzner et al 2016".

The presentation of the new heuristics might not have been as clear as we hoped. To facilitate the understanding on this part of our work, we introduced additional details in Section 4. While the heuristics are related to the model we build to extract changes, we believe those are more relevant in the section dealing with how to instantiate the model. To this end, we added in the various subsection of section 4 an additional paragraphs on how this information extraction was updated. Those paragraph give broad indiciation of what was changed to extract changes from each type of artefact. To obtain more information on the heuristics themselves, we provide the implementation of the FEVER approach. This includes 4 Java classes, one per type of artefact (Source, Mapping, Features, and file changes), describing in details how we manipulate the files.

3 Answers to the remarks of R2

> R2.C1: In the abstract the authors talk about the issues of not knowing about the co-evolving features: compilation errors, invalid products, or dead code. It would be great if the authors could add a discussion section in the end, where they could discuss how their tool can help alleviate these issues.
>
> R2.C2: The authors have an accuracy of around 85%. While this is great, there is a misclassification of 15%. It would be great if the authors could candidly describe the cost of misclassification on the different stakeholders (both from the aspect of false positives and false negatives).

We chose to reply to those two comments by adding a paragraph at the end of section 6 on "FEVER usage scenarios". We present how FEVER can be of use for the challenges mentions, and relate this to the accuracy of the tool in general, with what we believe can be achieved with its potential risks. The paragraph reads as follows: *The implementation of variant-rich system is known to be challenging. Features and their relationships, if misunderstood, can lead to issues such as dead code, invalid products, or compilation errors. To mitigate such problems, researchers have to identify the issues, find a mean to fix them, and apply them on a number of cases for validation. We argue here that tools such as FEVER are a convenient way of identifying what changes occurred in commits with respect to features. Once a research has found a scenario where the studied error occurs, one can easily find other scenarios, with similar changes, and observe if the error occurred as well. FEVER by itself cannot mitigate such problem, but constitute a great tool to facilitate research in those domain: by easing the search for problematic situation, and providing quantitative estimate of the occurrences of problematic evolution scenarios. Given the current accuracy of the FEVER prototype (85%), a manual review of the changes is necessary to guarantee that the retrieved changes are all correct. However, it is sufficient to reduce efficiently the number of commits that must be reviewed, and provide a solid starting point for further manual analysis, as was done by Sampaio et al.[]. Conversely, should a developer run a query on FEVER and get no results, there is a small chance that FEVER may have failed to extract such changes (false negative). In such a situation, the developer might have to rely on Git query instead, but FEVER already provide some information: the type of change sought by the developer is not common, or the implementation used to support those specific features/constructs are not what is commonly used in the Linux kernel.*

R2.C3: Table 8 is great, but it would be good to see the results (at least in an appendix) per category of commits so that we can understand when and under what circumstances, FEVER works great an where it performs poorly. Such a breakdown could also give us insight on how to improve FEVER. This goes back to my first comment.

To answer this question, we added a paragraph in Section 8.1 - Threats to Validity, on the feature-oriented change extraction process. We include this paragraph as part of the "limitations" of the current approach. Although we do not provide the breakdown as suggested, we believe that this is sufficient to shade additional light on the capabilities of the FEVER approach.

The paragraph reads as follows:

FEVER change extraction approach is based on heuristics to parse an analyze changes operated on commits performed by developpers. Our evaluation shows that FEVER captures feature-related changes with a relatively high accuracy (87.2% of commits extracted completely correctly). In the remaining commits, at least one information was inaccurately reported by FEVER. FEVER may fail in when changes to artefacts deviate from the "usual" development practices (naming convention, feature-file mapping approach and so on). Such cases are when dealing with architecture specific features, where the link between features and artefacts in Makefiles relies on variable values rather than straight forward foldering structures - as is the case for sub-architectures of the ARM main architecture. On some occasion, the object file included in Makefile by default in the compilation process is not the usual Linux "obj-y" list. In such cases, FEVER is not necessarily able to determine that those artefacts are associated with the feature that condition the inclusion of the Makefile. Errors in the code changes are mostly due to the problems when assigning code changes to block changes - we can identify if a block has changed (added or removed), but finding how the code inside the code was modified remains a challenge. This is particularly true when we observe series nested #ifdef statements, each containing a single line of code. Finally, we have some difficulties assessing whether a symbol in the code is a reference to a feature or not. A number of C macros in the implementation may come as false positives. Despite such shortcomings, occuring when developers do not, or are not able to, following the usual development guidelines, FEVER still produces correct results for 87.2% of the commits in our sample.

R2.C4: In section 5, it would be great if the authors could also candidly discuss the drop in recall in their new approach in a few cases.

We adjusted the description of the results obtained in Section 5. It now contains the following paragraph:

If we compare the results obtained during the complete evaluation with the results obtained during the replication of our first evaluation, we note that, for a number of change attributes, the accuracy dropped by small yet noticeable amounts (for precision and recall alike). This is due to the increase in our sample size and a more thorough sampling approach. We considered for the complete evaluation a larger, more representative sets of commits in the Linux kernel as explained in Section 5. As we observe more changes, we see more commits containing changes deviating from what FEVER is able to parse. We can say that, given the increase in

our sample and the a more balanced sampling techniques, this evaluation is more precise than what we had provided in the past.

> R2.C5: In S2 and S3, there is a lot of research that has already been done. Can the authors place how their approach will compare/compliment against the state of the art in Section 6? Currently there is only a description of how FEVER can be used in these scenarios.

We adjusted Section 6 presenting how FEVER can be of use to address several scenarios.

The scenario S2 refers to a developer introducing a new feature and being guided by past changes. We completed this scenario by adding the following information: *In this scenario, FEVER is used as a recommender system to guide the implementation of a new component. Relying on previous activities to guide further development is a common approach to ease software evolution. We can name Hipikat [Cubranic and Murphy, 2003], CodeBook [Begel et al., 2010] as tools aiming for such facilities. However, such approaches do not take into account the deep structure of the implementation in such the way that FEVER does, i.e., by breaking artefact changes by feature. This degree of granularity is particularly interesting for variant-rich system. For such scenarios, we believe that the information obtained by FEVER would be a valuable addition to existing approaches such as CodeBook, rather than a replacement*

Scenario S3 is a bug triage situation. We extended the scenario description as follows: *Bug triage is a known challenge for large software system development. The number of bug report, and the number of developer make appropriate bug assigment to developers difficult. To alievate some of those issue, several approaches have been designed to facilitate the identification of experts capable of fixing a bug [Ahsan et al., 2009; Matter et al., 2009]. Most of those solution rely on previous fixes to determine, based on bug report content, who is the most likely to be able to provide an answer to a bug report. What we propose is to take into account a feature-based expertise, and relate the bug report content with specific features, in order to determine who is the best suited to fix bugs related to that feature. This provides an additional type of information, based on fine-grained artefact changes, which can be particularly useful for artefacts at the limit of a subsystems where more than one team may be considered as potential fixers [Guo et al., 2011]. We do not claim that FEVER could replace existing approaches, but feature-related evolution information could be added to increase the accuracy of existing techniques.*

> R2.C6: In section 7, it would be great if the authors could add a "so-what" subsection to RQ3 and 4, where they could discuss the take aways for the results. For example, if the linux dev team is going to read these results, what should they change? What should researchers do? I was hoping to see this in the conclusion section at least, but the conclusion ended up being a summary of results an future directions. Either a new subsection/section or in the conclusion would be a good place for this discussion.

To address this point, we chose to extend section 7.4 "On Co-Evolution in Linux". In this section, we already suggest how our results might affect the future of variability management tools. We added information to futher strengthen our point. The last paragraph of this

section now reads as follows: *Hellebrand et al. noted that, in an industrial context, and for highly con- figurable systems, the evolution trends were leading towards less co-evolution of artefacts (source and model artefacts in their case [Hellebrand et al., 2014]). Such observation is consistent with the idea that common evolution scenarios should not require many modifications in many artefacts of different nature. In the Linux kernel, we have shown that co-evolution of heterogeneous artefacts only occurred in 30% of commits, and only for 25% of the developers. Consider- ing that those ratios are relatively stable overtime, we can assume that those are the results of choices in the Linux architecture, development practices and the choice of technology to support variability (Kconfig/Makefile/pre- processor annotation). The data we gathered constitute a base line for further studies on co-evolution in Linux. Further changes to the implementation tech- niques used to support variability implementation should not increase artefact co-evolution beyond what we observed in our study. Additional studies on feature-oriented co-evolution on variant-rich software systems, beyond the Linux kernel, would allow us to see if other mature variant- rich system evolve with similar ratio of co-evolution. With more points for comparison will be in a better position to assess whether this ratio of co- evolution is optimal or not.*

4 Answers to the remarks of R3

> R3.C1: One of my main comments is about the generalizability of the approach, as it entirely relies on Kconfig as well as on the kinds of mappings allowed by Makefiles. I.e., how representative is Kconfig for the entire space of configuration models/technologies? In fact, what does this space cover? It would be interesting to read a discussion on how FEVER would generalize to one of the other state-of-the-art configuration languages. Also, the Makefile mapping using "obj-..." is a Linux idiom, what do other projects do and could such mappings be supported as well?

We address part of this concern in Section 8.1. when we mention eCos and the CDL language. We agree on the points raised by Reviewer 3, and endeavour to make such dependencies on Linux clearer. We adjusted this section to read as follows: *As mentioned in earlier this section, the heuristics used to identify feature names and usage in the different artefacts are based on development practices. For instance, in other systems, it is unlikely to find that feature names are prefixed like as they are in the context of the Linux kernel. Similarly, the association between features and file might not be achieved using Makefile, and even in this case, they are other ways to do so without the usage of lists as done in the kernel. The mechanisms used to implement variability in the system must be known in order to be able to apply a FEVER-like approach to analyze feature evolution. While they might differ wildy from system to system, we argue that such mechanisms exist and should be documented. Therefore, it should be possible to adapt the FEVER approach for any type of systems.*

We can explore more in depth how adapting FEVER could be done. We did not include this in the paper, as we think this is implementation-specific.

Kconfig and CDL have something in common: constraints are define per features (as opposed to the FODA notation, where cross-tree constraints are not assigned to a specific feature). This being said, what we have is a "feature" with some "attributes", where some attributes represent cross-tree constraints. If we deviate from this (and move toward the

FODA notation for instance), the problem becomes very different. The EMF change model we use is no longer relevant.

Then, we also rely on Kconfig attributes and use them in our change model. Those are not necessarily generalizable (select statement for instance is likely to be very specific to Linux). But when we look at the CDL language, we see that we can represent a CDL feature with our change model, we would simply leave out some of the Linux specific attribute empty. That's not very elegant, but it would work. The challenge here is to build CDL parser, which already exists, and use it to instanciate the EMF model used for comparison. This in itself is not very hard: the class implementing the EMF model instantiation is less than 500 lines long (with comments).

Regarding the generalizability of the mapping, the situation is quite similar but more favorable to us. The EMF model used for the mapping is basically the pair ¡feature,target¿. This is very generic, and can represent most of the mapping that we can think of. The question is then how to recover the mapping. This is done by our Makefile parser, which, like the Kconfig parser, is a relatively small piece of code.

So yes, the current FEVER prototype contains a lot of references to Linux specific details. However, the overall approach (using models for diff, and feature identification) is in theory quite generic. The information we capture to do so should also be quite generic (features with attributes, code blocks, mapping). The EMF models are more Linux specific, but we could have renamed our attributes or merge some (depends and select for instance) to have a more generic model. We consider that would have been a premature optimization.

We maintain that the approach can be adapted to be applied to more systems than Linux, and beyond the Kconfig/Makefile/IFDEF technologies with a bit of work. The overall concepts remain the same.

> R3.C2:It took me a while to realize that a TimeLine object basically ties together all changes (commits) to all artefacts/models/mappings involved in a particular feature, which forms the core of the approach. It is important to stress this from the beginning, as this remains implicit for a long time. I did not like the term "TimeLine", as it does not express the link with a specific "feature".

We agree that the notion of **TimeLine** is not presented as soon as it should have been. We extended the description of the FEVER change meta-model (page 9) to add the missing description of **TimeLine** entities. The paragraph was extended with the following sentences: *Finally, **Edit** entities pertaining to the same feature are linked together through **TimeLine** entity. This grouping changes per feature using **TimeLine** entities is done over multiple commits (a complete release in our experiment). Therefore, the **TimeLine** of a feature aggregates all changes that occurred to that feature over time.*

> R3.C3: Furthermore, it was not entirely clear what "playing a role in thebehaviour of another feature" actually means in the case of FeatureEdits. Eventually, it seems to refer to any select/depend expression in the feature model, is that correct?

The intended meaning of "playing a role" is a larger than what you imply. We added a sentence after this one in the paper to further detail what is meant. It reads as follows:

For instance, in the first case, Feature B plays a role in the implementation of A if we can find an #ifdef block refering to B in a source file mapped to Feature A. Similarly, Feature B plays a role in the definition of feature A if Feature B appears anywhere in the definition of Feature A in the variability model (as part of a default value, depends or select statement or any other attribute)

So, in many cases, indeed, a feature will play a role in the definition of another through select/depends expression - since those are very frequent. But we don't exclude cases where the role played by the feature is located in the visibility of the feature, or the condition of a default value.

> R3.C4: Fig. 5 needs a better legend to clearly explain the color coding. Why do 2 nodes contain the "APDS9300" label? I also would not hide information into the small nodes, as it is difficult to try and match this figure with earlier code listings.

In this instance, we observe three nodes with APDS feature name: one for the FeatureEdit object, one for the FeatureDesc object and one for the TimeLine object. We are not certain that a color coding would provide more information. We propose here an updated caption for the image providing a more thorough description of all entities in the figure. *FEVER representation of commit 03eff7b60d - all entities and relationships. For readability purposes,* ***ArtefactEdits*** *are represented by small unlabelled gray dots. From top to bottom, they represent edits to the following files: a documentation file, the source file containing the behavior of feature APDS9300, the Makefile containing the new mapping, and the Kconfig file containing the new feature declaration. On the left hand side, we see three commits. On the right hand side, we see three feature* ***TimeLine*** *entities, one for each feature that was adjusted in the commit. In the middle, from top to bottom we see two source edits (labeled "ADDED") indicating that two #ifdef blocks were added, one* ***MappingEdit****, labeled "apds9300.o", then a* ***FeatureEdit*** *entity indicating taht feature APDS9300 was changed, and a* ***FeatureDesc*** *entity containing a detailed description of how the feature "is" after the change*

> R3.C5: I did not fully understand the rationale and repercussions of ignoring "source" statements in the Kconfig files. Why is this required, and what are the risks? Are certain changes not recorded by FEVER? It seems so ("FEVER will not capture such complex changes."), but I did not fully understand this.

We consider here changes performed in a small number of files (compared to the complete kernel). The "source" statements refer to files which might not have been touched, and therefore for which we may not have any information readily available in the commit. When try to parse the Kconfig files, attempting to resolve those based on the touched file is likely to fail. In most cases, only one Kconfig file is touched, so the "sourced" files, or "sourcing" files are not present in the commit. We remove it to avoid attempting to resolve references to files we do not have.

The risks here are missing co-evolving Kconfig files, or least the fact that they are related. There are cases where a new Kconfig is created, sourced, and then a set of existing features are moved to the new file (a form of refactoring if you will). By ignoring the source statement, we prevent ourselves from ever finding out that this is a refactoring. All we can see is the removal of features somewhere, and their apparition in a new file (which we will track).

Another drawback is the loss of semantic information. Because we ignore source statements, we cannot reflect on how constraints are propagated from one Kconfig file to the next. So the changes we extract are really textual changes - we perform a semi-structured diff. We get changes performed by developers, but we miss the meaning of those changes with respect to the set of valid configurations.

As long as we are interested in the changes performed by developers on the artefacts, removing the source statement should have very little impact. This is also a reason why we focus on changes performed by developers.

> R3.C6: How was the manual analysis of Git commits for RQ1 and RQ2 done? One complexity is that code changes hidden inside a conditional block might be shown without sufficient context, such that it is not clear that the changed lines are within an "#ifdef". How was this risk addressed?

To validate the changes in the code - both the code itself and its containing blocks, we compared the FEVER database with the GitK commits. In a number of cases, the results from FEVER "appeared" to be different from the GitK view of the change because of the missing context. This forced us to be particularly careful about context - mostly because it causes us to consider FEVER data as inaccurate when it is, in fact, right.

We should also note that using CPPSTAT really helped here. CPPSTAT expands nested conditions, and we rely on those to identify touched features. This allows us to identify wether the change described by FEVER is supposed to be in a nested #ifdef or not (and therefore search for nesting, or its absence).

But more generally, we need a large context to understand changes in all types of artefacts: adding a feature within a "menu", or "if" in Kconfig (which may alter its dependencies), adding a mapping within a large "if" statement in a Makefile (affecting the effective mapping), or, as you pointed out, code changes in very large "ifdef" blocks.

For this experiement, we considered that checking the changes against the proper context was part of the "normal" way of doing it. There are, to the best of our knowledge, no other way to properly check our results. Hence the lack of mention on this point.

We added the following sentences to section 5.1 (evaluation method): *GitK provides a view of the list of changed files, the chunked of modified texts in each of them, with an adjustable number of lines of context for each chunk. The number of line of context provided for each chunk is particularly relevant for us since conditionally compiled code blocks can be large, and identifying in which block a change occurred may require a very large context (up to the complete file)*

> R3.C7: My main issue with RQ1/RQ2 is the discussion about "variation of population". From what I understood, the first author found a different ground truth when addressing RQ1/RQ2 for the MSR 2016 version of FEVER than when doing it for this paper's version. This is a bit unusual. First of all, having more than one person perform such a manual analysis is recommended to reduce the risk of errors in the ground truth.

We agree that this is an unconfortable situation. Given the amount of work necessary and the degree of familiarity with Linux required, it was difficult to find more people to

double check everything. To compensate for human error, we simply adjusted our protocol to include two validation passes, with a delay in-between to reduce possible errors. We provided additional information on this point when answering to the general remarks.

> R3.C8: Second, why did one have to rebuild the ground truth for this paper's replication of the MSR 2016 study instead of reusing the same ground truth? Was this only due to the issue with local macros?

To assess the correctness of the approach, it made little sense to compare our results with something that we know to wrong. Therefore, instead of comparing our results solely against what we had before, we also checked against the actual commits.

The local macros is the change that caused the largest error on our part when establishing the ground truth. It is not however, the only case. We can observe small variations of ground truth in a number of change attributes. This being said, the errors were in a small number of commits.

> R3.C9: Third, has the precision/recall of the MSR version of FEVER been adapted to deal with the incorrect identification of local macros, or are the same precision/recall values used as in the MSR 2016 paper?

We kept the information as presented in the MSR submission. We decided it was better to present the number as they are, with the discrepency in terms of ground truth and explain those in the current version of this work. The alternative would mean inconsistencies between the two versions of the work, which would be more confusing than anything else.

We believe that reporting what we had, and what we have now makes the extension of the work that much more valuable for researchers on this topic.

> R3.C10: I liked the discussion of use case scenarios. Both the scenarios for developers and researchers made sense. The exploratory analysis as well provided interesting insights, in particular regarding the feature dimensions that are being changed after feature introduction. However, one thing that was missing is a deeper analysis of which authors only make implementation changes vs. those that touch other feature dimensions as well. Are the latter developers more experienced in terms of number of prior commits? Are they the original author or maybe a maintainer?

We did not include the human aspect of feature evolution in this study. We were interested in figuring out how "important" co-evolution is in the evolution of the kernel. In a sense, we are interested in how many people deal with co-evolution rather than their profile as developers. A recent study by Avelino et al. (with Leonardo Passos) was published on this specific topic ("Assessing code authorship: The case of the Linux kernel" - I.C.O.S.S. 2017). Their work is more in line with what you suggest here.

We believe the data we collected can be of use when attempting to answer such questions. However, we did not explore this dimension of feature evolution.

R3.C11:The RQs and text refer to improvements compared to the original FEVER, but the paper only lists them for the first time on page 24. It would be better to put those changes early on, even in the introduction.

We agree that the improvements over the previous iteration of the work were not necessarily presented early enough. In accordance with comments from other reviewers (R1.C1), we included in several subsections of section 3 (presentation of the approach) a small paragraph presenting what changed between the first and last iteration of the work. By doing so, we introduce the technical aspects of the improvement earlier.

R3.C12: In general, the paper refers quite a bit to the MSR paper and earlier work. For example, the dumpconf approach could use some more details in terms of its input/output as well as what kinds of changes it is able to find. This EMSE paper should be able to stand on its own, without requiring readers to also consult earlier papers.

The paper was built on top of the previous MSR paper. We did not remove any information that was previously available. While we refer to our previous work a lot, this is due to our intent to compare our new results with the previous installement of the work.

In general, we could have given more information on the different tools we used, such as Dumpconf, or CPPSTATS. None of those tool can "identify changes" on their own. They merely transform the artefacts that we will then compare. Those tools are well documented on their own. Furthermore, we use them to transform the various artefacts into the different change models that are completely presented in this work. In the future, there is no guarantee that DumpConf or CPPSTATS will remain available, but with the change model one can still reproduce our work.

R3.C13: Page 9: "The inclusion of a Makefile in the build process may be subject to feature selection.": In what sense? How is this done? Via a conditional include or recursive make?

We adjusted the description to read as follows: "via conditional inclusion, or more complex mechanism relying on variables and file path reconstruction."

R3.C14: Page 11: "pointing to the artefact in which the change took place": At line level?

We adjusted the description to add details on this specific point. The text now reads as follows: *"This relationship is established at a file level. The details of the changes within that artefacts are contained in the associated Edit entity."*

R3.C15: Page 13: "The FeatureDesc entity captures the information presented in Table 3.": So, if at least one attribute changes, an entirely new FeatureDesc is created?

Yes. Any chance to a feature may lead to the creation of a new TimeLine entity - if none existed. Because TimeLine entities are meant to regroup all changes (no matter how small) pertaining to the same features, we believe this makes sense. To clarify this point, we added the following sentence: *"For any feature change occurring at a variability model level, the change will be represented by a "FeatureEdit" entity, and at least one "FeatureDesc" entity in case of addition or removal, and at most two in the case of the modification of an existing feature."*

R3.C16: Table 4: What is meant by "variable"?

In this table, we mentions that a mapping is performed between "something" and an asset. In the general case, that "something" can be a feature or any intermediate variable. This happens when we deal with aliases for instance.

However, in our approach and experiment, we actually do not keep track of the assets mapped to internal Makefile variables. We record them, extract possible changes but once we are done, if the asset is mapped to a Makefile variable, we discard it. We only keep changes affecting features. Therefore the only valid value for the attribute is "FEATURE".

It was awkward to keep this information in the table. We removed this notion from the change attribute description. However we keep the overall description of the MappingEdit type, so that the description matches the content of the dataset.

R3.C17: Fig. 5: "the sourcefile containing the behavior of feature APDS9300,": Does this one have conditional code inside?

As shown in diagram for that commit, we see that the file containing the implementation of our feature APDS9300 is associated with two SourceEdit (labeled ADDED). So, by default, the correct answer to the question is: I do not know whether it contained any #ifdef block in the past, but after the change, there are at least more two of them. We updated the description of the Figure to make this clearer (see our answer R3.C4).

R3.C18: Page 18: "creating if necessary a new TimeLine entity": When does this happen? When a new feature is added?

The condition under which feature TimeLine entity are created are listed in Section 4.5.

R3.C19: Page 18: "the change affecting the feature": What does "change" refer to? The commit?

In this instance, "change" refer to the transformation operated on the feature by a developer. It will be contained within a commit, but we cannot say that the commit it the change.

R3.C20: Page 18: “the model we rebuild is always partial”: In the sense that parts of feature that were not touched are not included, or what does partial refer to here?

“Partial” refers to “partial with respect to the complete variability model of the Linux kernel”. We rebuild a portion of that variability model from the artefacts (Kconfig files here) that are changed by a commit.

R3.C21: Page 20: “compilation flags either start by the follwing strings “-D”, “-L”, “-m”, or “-W”, “-I”, “-f”.”: What if these flags are stored inside a make variable, possibly dynamically assigned/determined?

In such cases, we miss the information. We cannot determine that those are compilation flags, the variable containing the flag should be assimilated to a file/compilation flag/folders. It is very likely that a change involving a variable name will simply be ignored by FEVER (unless we mistake the variable name for a file, a folder or something else).

We addressed this point as part of our answer to R2.C3.

R3.C22: Page 20: “ mapped to the alias tree_test.o referring to two compilation units tree_main.o and tree.o. ”: Where is the mapping between tree_test.o and tree_main.o/tree.o?

We updated the example with a sample taking directly, as is, from the Linux kernel (as opposed to a simplified one used for the moment). They are very similar in nature. The new example is the following:

```
obj-$(CONFIG_BLK_DEV_SWIM) += swim_mod.o
swim_mod-y := swim.o swim_asm.o
```

We adjusted the paragraph describing the aliasing as follows: *An example of an alias is presented in the following listing, where feature CONFIG_BLK_DEV_SWIM is mapped to the alias “swim_mod.o” referring to two compilation units “swim.o” and “swim_asm.o”. The association between “swim_mod” and the two compilation units is done the last line of the listing. We identify such aliases based on the naming convention : name of the object file appended by “-y”. Note that there are no concrete artefact corresponding to “swim_mod” by itself in the Linux source tree.*

R3.C23: Page 22: “CPPSTATS expends conditions”: Not sure what “expends” refers to here.

We rephrased the sentence to avoid the word “expends” deemed too generic for the purpose at hand. The sentence now reads as follows: *It should be noted that CPPSTATS provide the condition of each block by taking into account nesting. In practice, if a block with condition B is nested inside a block with condition A, CPPSTATS will report two blocks, one with condition A and one with condition “A&B”.*

R3.C24: Page 22: " FEVER uses a combination of the condition of the block combined with its content (the actual code) as a unique identifier. ": What if the content changes, does this invalidate finding a mapping with future versions of the block?

In this instance, we would like to point out that we will be looking for THE next version of the block - as it is after the changes proposed in a commit. If a code block is changed multiple times in multiple commits, we do not attempt to relate those changes together directly. The fact that those changes are related can still be observed through TimeLines, but not between blocks.

If the content of a block changes between two revision of the same file, then those changes are captured as "added/removed/modified blocks". And the changes inside the blocks will be qualified as "added/removed/edited/preserved".

It is easier from an EMF Compare perspective to conside that, if the content of the block is modified, the block is new (or entirely removed), then we use information from the Source Code change model to figure out if the block pre-existed / still exists (through line numbers, conditions and so on).

R3.C25: Page 23: " by invoking it for every Kconfig file and Makefilepresent in the system.": This also ignores the "include" statements, so to what extent does this impact the initial feature list?

The effect of doing so on this initial feature list is the "removal" of duplicated feature names. In the Linux kernel, several features are declared multiple times, and we only keep the first occurrence of the feature. That's information we lose. This being said, at this point of the process we are interested in finding what features exist in the systems - so this is a minimal loss.

Removing the "include" statement affects our ability to analyze the feature hierarchy. For this operation, it does not matter. Removing this "include" does not affect our results for that operation.

R3.C26: Page 23: " then run through all commits, starting from the leaves in a breadth-first manner ": What are the leafs? The first commit (there is only one)?

There a multiple "first" commits per release. We find interesting commits by using "Git" mechanisms. We search for all commits "accessible from the end commits" which are "not accessible from the first one". The "leaves" are then all the branches which were not merged into the kernel at the time of the release. We have one ending commit (the last of the release), but multiple starting commits.

We added the following information (with the footnote). *Note that there may be more than one initial commit in a set: we have to consider branches as well. In our experiment we usually have one initial commit of the release itself, and the different branches that have not yet been merged.*[1]

[1] the list of commit is obtained using the following Git log command, asking for all commit reachable from the last considered commit and not accessible from the first commit of the release - i.e.: git log v3.7...v3.8

R3.C27: Page 23: “ we do not attempt to map such files to features. ”: How much does this limit applications of FEVER?

This sentence refer to the mapping between features and files, when considering header files (.h), more specifically in the /include folder in the kernel.

The question is: in the Linux kernel, are those file mapped effectively to any features ? Since those can be included in (any) source file through a single #include statement, are those really conditioned by features ? Or are those included in the final product through #include statements ?

Given that in some cases the header files, used mostly as interfaces, do contain actual implementation, it is reasonable to think that they *could* be mapped to a feature. On the other hand, the mapping is never explicit in a makefile, and an header file may be used by several implementation (so for multiple features) - which one(s) should the header be mapped to? We took a rather conservative approach. When we can find a mapping in the Makefile, we use it. If we cannot, then the inclusion process is performed by some other means than feature selection. The possible impact of this choice is to minimize the number of changes occurring in the implementation for a given feature. Given our conclusions, this would only strengthen our observations.

R3.C28: Page 24: “ how the changes to theheuristics impacted the accuracy of the FEVER approach ”: What were the old heuristics?

Page 24: “all new change attributes, ”: Which ones?

Page 24: “ Section 3 described the FEVER approach with its improvement ”: No, it only discussed the resulting approach, no clear statement of what is new.

Page 24:“ Fromthe initial version of this work, we improved the following aspects of the approach:”: This listing should have been put earlier in the approach section (or even introduction).

Page 24:“ heuristics for code reference identification - heuristics for code changes within modified code blocks - heuristics for asset-feature mapping identification (compilation flag, defaultlist, and artefact extensions management) ”: These did not exist at all in the previous FEVER, or they have been improved?

We added details on how the heuristics changed between the two versions of the work in the description of the approach - see our answers to R1.C1. We provide additional information on the new change attributes by adding the list in parenthesis as follows *(artefact changes, “data” artefact types))*. With those adjustments the changes to the FEVER approach will be clearer.

R3.C29: Page 24: “RQ2: To what extent does the improved FEVER data match changes performed by developers?”: Isn’t this RQ answered automatically when RQ1 is (or vice versa)?

We argue that RQ1 is about “how much better we are?”, and RQ2 is “how good is it?”. We agree that those questions are related. However, they address different aspect of the work and therefore should be separated.

R3.C30: Page 25: “we considered that this change could not have been mapped by FEVER, and FEVER should notreport any feature-related mapping change. ”: What if the mapping is elsewhere, not in the path to root? Would this be treated as a false positive?

In such cases, the mapping between files and features are : not in the Makefile contained in the file's folder, and not in any of the Makefile in the direct hierarchy. Indeed, this does not mean that the file is not mapped to any features. From a theoritical point of view, there should always be at least one combination of features leading to the inclusion of a specific file in the kernel - otherwise that artefact is useless and should be removed (excluding documentation, tooling scripts and so on). We considered here that we cannot capture those changes, and this is part of the limitation of the approach, not error per se.

We updated the related section as follows to make our protocol clear on this point: *During the evaluation, if an artefact is not assigned to a feature in FEVER and we cannot manually find which feature it should be assigned to following the methodology presented above, we consider that the FEVER output is correct.*

R3.C31: Page 25: “ the file ”./mm/Makefile” is not constrained by anyfeature in the root “./Makefile” of the kernel source tree. ”: What does “constrained” mean?

We adjusted this sentence as follows: *We see that the inclusion of the file “./mm/Makefile” is not conditioned by any feature in the root “./Makefile” of the kernel source tree.*

R3.C32: Page 27: “ and “code change: added code” are related ”: There is not really an increase for this one.

While being in a new paragraph, it was our intention to talk here about the sample size, rather than the accuracy of the tool. We updated this paragraph to make this clearer.

R3.C33: Page 27: “We adjusted the algorithm to identify files, enforcing strict file extension (.S), hence the file was ignored during thesecond evaluation.”: Why are .S_shipped files no longer included? Because they correspond to firmware files?

The .S_shipped files are no longer included because they correspond to specific version of existing .S files. So their management is somewhat different than for other files. It didn't seem correct to associated them with features, when the actual mapping within Makefile are done to .S files rather than .S_shipped files. Should one want to follow the evolution of features with respect to those files, we would recommand observing how their non-shipped counterparts are associated with features.

R3.C34: Table 7: What explains the difference in number of TimeLine objects? Is this due to the larger set of links/features found by this paper's FEVER version?

No, the increase in the number of TimeLine objects has to do with the conditions under which we create such entities. From our initial work, we extended the notion of "TimeLine" - we now create TimeLine objects for features whose influence change over time. The conditions under which we create TimeLines are presented in the section 4.5 (Consolidation and TimeLines). This is further presented in section 5, when we state that we changed "the timeline model to include *influence updates* on feature changes".

We adjusted our comments on the TimeLine evaluation by adding the following sentence: *The conditions under which we create **TimeLines** are presented in Section 4.5. In that list, the points two and three, on feature relationship changes were previously not recorded. However, the 743 **TimeLines** initially recorded are a subset of the 11,225 **TimeLines** observed during the replication.*

> R3.C35: Page 28: " did not allow for commits not affecting any feature to be included in the evaluation, which, in our opinion created a bias in the evaluation": Not sure what this bias refers to?

This bias is related to the construction of the sample we used for our MSR evaluation of the FEVER accuracy. The first set was built by taking commits affecting certain artefacts. That excluded commits not affecting anything. So we still needed to make sure that those commits were correctly handled i.e. they indeed affected no files, and that FEVER didn't create wrong Edit entities. In that sense, we excluded from our sample cases that could lead to potential errors - hence the bias mentioned in that sentence. Our updated sampling approach compensate for this case - and is generally more comprehensive.

> R3.C36: Page 29: "51 commits not affecting any artefact, ": Not even source code? Not even the source! There a number of those.

Indeed. Not even source code. Such commits are tags and merges. In merges, there are no new changes in the artefacts - just previously existing changes get merged. This point goes with your remarks on Page 29 regarding the counting of the "+" and "-" signs in diff. That's were we check that the merge were indeed "pure merges of existing changes ", and are not introducing new changes (which we would then miss).

> R3.C37: Page 29: "255": Why 255 and not, e.g., 250? What is special about that number?

This number comes from the first evaluation we did, consider 50 changes per category of change (combination of affected spaces, from none to all). This should have led us to 250 changes, however during the evaluation, we accidentally included one more commit in each category. It made little sense to remove it just for the sake of having a round number, so we kept it.

R3.C38: Page 29: “Using this view of the patch, we searched for content added or removed from all parents. Practically, this amounts of searching for lines in the “diff” where the number of “+” or “-” symbols atthe beginning of modified lines of text equals the number of parents.9”:

I did not understand this approach and why it is necessary. Is it necessary to enable a study like the exploratory analysis of co-evolution later on?

Page 29: “any of such change is accounted for as a false negative for the relevant change attribute during the evaluation.”: See previous comment.

This is important to avoid missing changes within merge. The underlying question is “what happens when a developer makes changes to perform a merge operation?” So, checking the number of “+” and “-” is a way of checking if all code changes within a merge are indeed the result of a merge - rather than new changes. Failing to take this into account, we risk missing information, i.e. false negative.

R3.C39: Page 29: “local variables”: I.e., constants?

We agree that the term “local variable” is not the best suited for this sentence. We concretely refer to such statements : “#define CONFIG_MY_PERSONAL_VAR”; which may be followed by a value, and maybe further referenced/adjusted using pre-processor statements. We propose to use the term “symbol” for this and we rephrase the sentence as follows: *During this evaluation, we found two cases where developers defined symbols using the #define C directive whose name matched feature naming convention (CONFIG_XX pre-fix)*

R3.C40: Page 33: “Because TimeLine entities are regrouping changes across spaces and commit”: This seems to be the first place where it becomes clear that TimeLine objects connect feature change data across commits. Maybe “TimeLine” was not the best naming for such a concept?

We adjusted the definition of “TimeLine”, to make it clearer and appear sooner in the presentation of our work. The idea behin this entity is that, starting for it, we can extract all changes over time (a release) affecting a feature, and effectively seeing its “evolution time line”. But we agree, maybe it wasn’t the best name, but we consider its sufficient.

This was further discussed in our answer to R3.C2.

R3.C41: Page 36: “odering” What does this mean?

It is a typo. It should have read “ordering”. In this case, it refers to ordering of first and last name within recorded user names (in my case: “Nicolas Dintzner”, or “Dintzner Nicolas”). The typo is now fixed.

R3.C42: Page 36: “incorporation” What does this mean?

We adjusted the sentence to give the complete meaning, namely:“irrelevant incorporation in the name”. This refers to user name containing additional information on the company or project they work on (i.e. a user name such as “Nicolas_Dintzner_TUDELFT”).

R3.C43: Page 36: “Regarding the co-evolution of artefacts with respect to feature evolution, we can see that most features evolve only through their implementation. ”: ... after introduction.

We adjusted the sentence as suggested to make this clearer.

R3.C44: Page 41: “ For this evaluation, the manual review of commits was performed twice - for the entire dataset, leaving a small timegap (between 2 days and a week) between the two evaluation rounds. ”: Is this between the replication and the analysis of the new data set?

Both the replication and the new dataset were checked twice. To be totally clear, we first did the replication for the first time, then the new dataset, then second round on both, in this order.

5 Acknowledgements and information

This research was supported by the Dutch national program COMMIT and carried out as part of the Allegio project under the responsibility of the Embedded Systems Innovation group of TNO.
The authors have no conflicts of interest to declare.
Please address all correspondence to:
Nicolas Dintzner, Tel : 00 31 6 231 57 647, Email: N.J.R.Dintzner@tudelft.nl
Martin Pinzger, Email: Martin.Pinzger@aaut.at
Arie van Deursen, Email: Arie.vanDerusen@tudelft.nl

We look forward to hearing from you at your earliest convenience.