# Untangling Source Code Changes Using Program Slicing

Muylaert, Ward; De Roover, Coen

*Published in:*
Proceedings of the 16th edition of the BElgian-NEtherlands software eVOLution symposium

*Publication date:*
2018

*License:*
CC0

*Document Version:*
Accepted author manuscript

# Untangling Source Code Changes Using Program Slicing

Ward Muylaert
ward.muylaert@vub.be
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium

Coen De Roover
coen.de.roover@vub.be
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium

***Abstract*—Version control systems (VCS) are widely used to manage the history of code bases. These histories in turn provide opportunities for research. Researchers expect the commits in these version control systems to be atomic. That is, each commit performs one task. This is however not always the case. To remedy this, we propose a commit untangling technique using program slicing. In particular, we posit that all related changes are part of the same program slice. To do so, we perform program slicing on *changes*. Preliminary results using intra-procedural slicing have proven to be encouraging. We are currently working on expanding our work to be inter-procedural.**

## I. Introduction

Version control systems (VCS) are widely used to manage the history of code bases. Prominent examples include Git, SVN, or Mercurial. A developer may "save" their changes into units called commits. Best practice suggests each commit should only contain changes related to one task. Such commits are called atomic commits [2], [14]. In this manner, the version control system can be used to keep track of how the program under development evolves. The version control system can also be applied to, for example, revert individual changes or port changes to other versions of the code base. On the research side, version control systems provide a trove of software evolution information open to analysis.

However, developers do not necessarily follow the best practice of creating only atomic commits [9]. For example, a small bug may be quickly fixed while working on another feature and placed in the same commit. Floss refactoring is another problem: refactoring in order to implement a new feature. These situations make for larger commits in which many unrelated changes are tangled together. Such commits are called tangled commits.

Tangled commits occur on a regular basis. A study by Herzig et al. found that up to $15\%$ of Java bug fixes contain tangled changes [3]. Tao et al. found that between $17\%$ and $29\%$ of investigated revisions were tangled [14]. Nguyen et al. found that $11\%$ to $39\%$ of all the fixing commits used for mining archives were tangled [10].

Tangled commits lead to problems on many fronts. We provide four examples. A developer will have problems reverting particular changes if they are a part of a bigger commit. A developer may also have problems integrating particular changes from another developer if the desired change is part of many different changes in a tangled commit. A code reviewer will have a harder time understanding larger commits of unrelated changes [13]. This in turn will lead to lower quality feedback [1]. A researcher interested in historical analysis, finally, will need to decide on the "one" function of a commit even though many unrelated changes may be present in the commit.

In light of these difficulties, we propose an automated commit untangling technique. Our technique employs program slicing around changes. Program slicing is a technique to answer questions about the influence of program statements on other program statements [15], [11]. We extend this idea. We posit that all related changes are part of the same program slice. Thus, a commit may be untangled by means of the created slices. A preliminary implementation of our technique performs intra-procedural slicing. Despite this limitation, early results are encouraging. We are currently in the process of expanding the implementation to work on an inter-procedural level.

## II. Architecture

Our technique consists of four major parts. First, the commit is distilled into fine-grained changes to the program's abstract syntax tree (AST). Second, a program dependence graph of the program is created. Third, a slice of the program dependence graph is produced for every fine-grained change. Finally, changes are grouped by means of the slices and the commit is partitioned accordingly. We have implemented our technique to work on Java programs. The rest of this section provides further detail into each of the four steps.

For the first step, we make use of ChangeNodes [12]. ChangeNodes works on the AST of a Java program. Given two versions of a program, ChangeNodes provides a list of Insert, Update, Move, and Delete operations. In our case the two versions are the version before and the version after the commit under analysis. Applying the obtained list of changes to the AST of the earlier version results in the AST of the later version. ChangeNodes thus provides fine-grained changes describing the commit.

For the second step, we make use of TinyPDG [6], [7], [5]. TinyPDG creates a program dependence graph (PDG) of a Java program. TinyPDG does this intra-procedurally. Our

main motivation for choosing TinyPDG is that it makes use of Eclipse libraries to create the underlying AST. ChangeNodes also employs the Eclipse libraries for this purpose. This makes it more straightforward to link these two parts together.

In the third step, our technique performs forward and backward slicing on the program dependence graph. This is done for every distilled fine-grained change obtained in step one. Thus for every change $c_i$ a slice $S(c_i)$ is obtained. We implemented the intra-procedural slicing algorithm as described by Horwitz et al. [8] on top of the PDG created by TinyPDG. The algorithm by Horwitz et al. performs backward slicing. We adjusted the algorithm so that it also performs forward slicing. When we refer to slicing, we consider the combination of forward and backward slicing.

Finally, our technique considers the following relation $R$ between changes. Changes $c_i$ and $c_j$ are related (notation: $c_i R c_j$) if and only if $c_i \in S(c_j) \vee c_j \in S(c_i)$. By definition of how slicing works, this relation is reflexive. The relation is also clearly symmetric due to its symmetric definition. The relation is however not necessarily transitive. We cannot state that if $c_i R c_j$ and $c_j R c_k$, then $c_i R c_k$. Consider for this the following simplified situation. $c_j$ is part of the root node of a PDG with two children. $c_i$ is part of one of the child nodes. $c_k$ is part of the other child node. Slicing in this situation results in $S(c_i) = \{c_i, c_j\}$, $S(c_j) = \{c_i, c_j, c_k\}$, and $S(c_k) = \{c_j, c_k\}$. Then $c_i R c_j$ and $c_j R c_k$, but $\neg(c_i R c_k)$. The relation $R$ is thus not an equivalence relation. Instead, we partition the set of changes into subsets by means of the following steps.

1) If a change is not in relation with any change in any of the existing subsets, create a new subset with that change in it.
2) If a change is in a relation with (an) element(s) of exactly one existing subset, place the change in that subset.
3) If a change is in a relation with two (or more) elements of different subsets, join the subsets together and add the change to it.

The last step fakes transitivity and effectively "widens" the relation $R$: more changes are considered related than they would be by our original definition of $R$. In terms of these subsets, we rephrase our hypothesis as: A commit is atomic if and only if our technique does not split up the commit into different subsets of changes.

## III. Evaluation

To evaluate our hypothesis, we make use of a dataset of five Java programs as used by Herzig et al. in [3], [4]. The programs in question are: ArgoUML, GWT, Jaxen, JRuby, and Xstream. These projects were chosen for meeting certain quality criteria. For each of the projects, Herzig et al. manually identified atomic commits using commit and issue information. Using the atomic commits, Herzig et al. also created artificial tangled commits for each of the projects.

We used this dataset to perform a preliminary evaluation of our hypothesis and technique. This preliminary evaluation has one obvious limitation. Due to the current implementation of our technique being intra-procedural, only atomic and tangled commits affecting just one method could be analysed. This limits the number of commits that can be analysed, the large majority of commits cover more than one method. In the case of the Jaxen project, no commits are left to be analysed. This limitation will be solved by our (current and) future work in which we are expanding the implementation to work inter-procedurally. The setup of our evaluation will remain the same for the inter-procedural evaluation.

We apply our technique to every atomic and tangled commit affecting just one method. We consider three separate outcomes for the analysis of a commit, regardless of it being atomic or tangled.

1) The commit is correctly identified as atomic or tangled. Our technique and the dataset agree on what kind of commit it is.
2) The commit is not correctly identified. Our technique identified the commit as atomic/tangled while the dataset calls the commit tangled/atomic respectively.
3) No result. This happened when the memory overhead made the analysis crash. We have not yet played around with this overhead in order to solve or analyse it.

In the case of atomic commits, the results are promising. For each of the projects except JRuby, the ratio of correctly analysed atomic commits is larger than $90\%$. For JRuby nearly $20\%$ of commits could not be analysed. Incorrectly identified commits happened largely through statements like `throw` or `catch` not being supported by TinyPDG.

Our technique falters on certain types of tangled commits. Except for the GWT project, more tangled commits were incorrectly identified as atomic than they were correctly identified as tangled. However, the large majority of these incorrect identifications are due to formatting changes or changes to comments. Our technique does not take formatting nor comments into account, only code. As such, a commit handling both one code related task and one formatting task would be classified as tangled in the dataset, but identified as atomic by our technique.

## IV. Future Work

We are currently in the process of making the implementation of our technique inter-procedural. To do so, we need to make two major changes.

First, our program dependence graph needs to consider the entire program. Rather than a *procedure* dependence graph, we require a *system* dependence graph (SDG) or *class* dependence graph (ClDG). To achieve this, we need to extend TinyPDG to work inter-procedurally. The main hurdle here is creating the summary edges via the algorithm as described in [8]. These are necessary to avoid the calling context problem as described there.

Second, our slicer needs to be adjusted to work on the inter-procedural system dependence graph. These adjustments are also described by [8].

Once our implementation works inter-procedurally, we will perform the evaluation described in section III again.

## V. Conclusion

We wanted to untangle commits performing more than one task. For this, we considered the hypothesis that related changes belong to one and the same program slice. We created a first implementation to test this hypothesis. This implementation works intra-procedurally. Despite this limitation, initial results are promising. Incorrect identifications happen largely either due to parts of the program our implementation does not handle (e.g., `throw` statements) or due to formatting and comment changes which our technique does not consider. We are currently working on expanding the implementation of our technique to work inter-procedurally. Once this is done, we will redo the evaluation with the complete dataset.

## Acknowledgment

Ward Muylaert is an SB PhD fellow at FWO, project number 1S64317N.

## References

[1] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *International Conference on Software Engineering (ICSE)*, 2013.

[2] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse, "Untangling fine-grained code changes," in *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015.

[3] K. Herzig, S. Just, and A. Zeller, "The impact of tangled code changes on defect prediction models," *Empirical Software Engineering*, vol. 21, no. 2, pp. 303–336, Apr. 2015.

[4] K. Herzig and A. Zeller, "Untangling changes," 2011.

[5] Y. Higo. Tinypdg: A library for building intraprocedural pdgs for java programs. [Online]. Available: https://github.com/YoshikiHigo/TinyPDG

[6] Y. Higo and S. Kusumoto, "Enhancing quality of code clone detection with program dependency graph," in *Working Conference on Reverse Engineering*, 2009.

[7] ——, "Code clone detection on specialized pdgs with heuristics," in *European Conference on Software Maintenance and Reengineering*, 2011.

[8] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, pp. 26–60, Jan. 1990.

[9] M. Konopka and P. Navrat, "Untangling development tasks with software developer's activity," in *2015 IEEE/ACM 2nd International Workshop on Context for Software Development*, May 2015, pp. 13–14.

[10] H. A. Nguyen, A. T. Nguyen, T. N. Nguyen, Electrical, Computer, and E. Department, "Filtering noise in mixed-purpose fixing commits to improve defect prediction and localization," in *International Symposium on Software Reliability Engineering (ISSRE)*, 2013.

[11] J. Silva, "A vocabulary of program slicing-based techniques," *ACM Computing Surveys*, vol. 44, no. 3, Jun. 2012.

[12] R. Stevens and C. De Roover, "Extracting executable transformations from distilled code changes," in *International Conference on Software Analysis, Evolution and Reengineering*, 2017.

[13] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, "How do software engineers understand code changes? - an exploratory study in industry," in *International Symposium on the Foundations of Software Engineering (FSE)*, 2012.

[14] Y. Tao and S. Kim, "Partitioning composite code changes to facilitate code review," in *International Conference on Mining Software Repositories (MSR)*, 2015.

[15] M. Weiser, "Program slicing," in *International Conference on Software Engineering (ICSE)*, 1981, pp. 439–449.