Similarity management of 'cloned and owned' variants

Thomas Schmorleiz and Ralf Lämmel
Software Languages Team
University of Koblenz-Landau, Germany
http://softlang.wikidot.com/

ABSTRACT

The 'clone and own' approach to software product lines assumes that variants are created by cloning and evolve more or less independently afterwards. In this paper, we describe a process to manage similarity of such 'cloned and owned' variants along the timeline. The process uses annotations for recording developer intentions and it leverages automatic change propagation. We describe a case study where we manage similarity for clowned-and-owned Haskell-based variants of a simple human-resources management system.

CCS Concepts

•**Software and its engineering** → *Feature interaction; Abstraction, modeling and modularity; Software maintenance tools;*

Keywords

Variability; Similarity analysis; Similarity management; Software product lines; Clone detection; Annotation; Change propagation

1. INTRODUCTION

The discipline of *clone management* studies detection, avoidance, and removal of clones as well as compensating activities. For instance, such compensation helps in the context of the 'clone-and-own' approach to managing *software variants* [15, 16, 5, 1]; the approach is widely established in industry for reasons such as developer independence [16, 4]. In this paper, we address the clone-and-own approach by a combination of variability and clone management which we refer to as *similarity management*. We treat the variants as a cloning genealogy [7, 17]. Clone detection is performed at a granularity level of 'fragments' by which we mean named abstractions such as methods or functions. Similarity measures for source-code files, folders, and variants enable exploration of similarity at different levels and, ultimately, similarity improvement. The developer must decide whether a found similarity is to be maintained or improved. In the first case, we speak of a maintenance invariant; in the second case of a maintenance task.

SAC 2016,April 04-08, 2016, Pisa, Italy

http://dx.doi.org/10.1145/2851613.2851785

Contributions of the paper. a) We introduce similarity measures and classifiers to capture similarity and evolution thereof across clowned-and-owned variants. b) We present a process for similarity management with activities for analyzing, annotating, maintaining, and improving similarities. c) We present a first case study on similarity management. The case study relies on designated tool support. All tools and artifacts underlying the case study of this paper are on the paper's website[1].

Research question. How to measure similarity of clowned-and-owned variants and effects of similarity improvement in a useful manner? One can define similarity at the fragment-level in some standard way using algorithms such as longest common subsequence or Levenshtein distance. However, we seek a deeper understanding of similarity including the development of fragment sharing on the time line. We answer the question in §6 by researching metrics such as the number of distinct fragments across all variants and the number of variants a fragment is shared in.

Road-map of the paper. §2 sets up the process of similarity management. §3 describes the analysis of similarities on the timeline. §4 describes annotations to be used for maintenance invariants and tasks. §5 describes tool support for developers who manage similarities. §6 describes the design of the case study, its results, and threats to validity. §7 discusses related work. §8 concludes the paper.

2. SIMILARITY MANAGEMENT

We set up the overall process for similarity management—from a developer perspective; §3–§5 fill in technical details.

Analysis. An automated analysis is applied to the commit history of given variants, thereby determining how fragment-level similarities evolved over time. For instance, a similarity may be identified as having 'diverged from equality'. The same analysis is applied incrementally whenever the developer makes local changes or pulls new versions.

Review. Subject to interactive tool support, the developer reviews similarity evolutions sorted by decreasing similarity

[1] http://softlang.uni-koblenz.de/simman

measures, i.e., equalities show up at the top. Similarities that are seen as adequate may be tagged to be taken as invariants such as 'Maintain Equality'. Similarities that have evolved unreasonably may be acted upon as follows.

Improvement. Option 1: The developer suggests *automated change propagation* to make two fragments the same again. As a side effect, the resulting fragment equality is tagged as an invariant that is to be maintained. Option 2: The developer *manually changes* the involved fragments to increase similarity, thereby, again, implicitly tagging the result as an invariant. Option 3: The developer *defers the change, but tags* the similarity evolution at hand with a maintenance task for a due improvement such as 'Increase Similarity'.

Reestablishment. Subject to interactive tool support, the developer is notified of invariants that have been broken by local or pulled changes, thereby encouraging the developer to reestablish these invariants by means of improvement, as discussed before, or to possibly abandon them.

Committal. The developer commits the local version also including the annotations for invariants or pending maintenance tasks. Thereby, developers can manage similarity collaboratively.

3. SIMILARITY ANALYSIS

The automated similarity analysis processes the commits of the given variants. The following concepts are involved.

Variants. Without loss of generality, we assume that variants of a given software system are organized as folders in a single repository. By tracking creation, removal, or moves of folders or files, we identify creation, removal, and renaming of variants.

Fragment snapshots. We extract all source-code fragments from all commits for all variants. The term fragment is used here to proxy for 'significant named source code unit' such as the methods of a class in Java or the top-level functions or type declarations of a module in Haskell. We speak of a fragment *snapshot* when a fragment at a specific commit point is meant. A fragment within a file is identified by a classifier/name pair and possibly an index for disambiguation. The complete locator for a fragment snapshot is a URL consisting of (names of) a variant, a folder, a file, and the aforementioned classifier/name pair.

Fragment tracking. A fragment is tracked as a series of fragment snapshots for a consecutive series of commits, where each snapshot represents the fragment at the respective commit point. Without loss of generality, we assume that snapshots at consecutive commit points are linked for tracking, when they have the same URL while handling renaming at variant, file, and fragment level as well as content changes of a fragment, and combinations thereof. The details of our tracking heuristic are available online.

Similarity snapshots. Similarity snapshots are relations of two fragment snapshots of the same version. To *measure* the similarity of fragment snapshots we adopt Cordy et al.'s idea [14] to pretty-print the token sequence of each fragment into many lines and then apply a text-based similarity measure. In this manner, we ignore dissimilarities that are based on formating and whitespace. As a text-based similarity measure we use 'diff ratio'. Let M be the length of the longest common substring of the two input sequences. The diff ratio is $2 * M/T$ with T as the total length of both input sequences. The value of diff ratio lies between 0 and 1. A diff ratio of 1 indicates that the two fragment snapshots are equal; they are perfect clones of type 1.

Similarity evolutions. We track similarities over commits, i.e., similarity evolutions, by extracting all similarity snapshots between all fragment snapshots of the fragment tracks for a given similarity snapshot. Fig. 1 illustrates the concepts explained thus far.

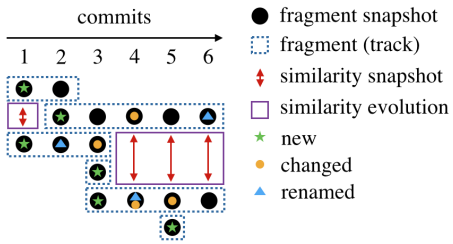

Figure 1: Illustration of fragment snapshots, tracks, and similarities along a series of commits $1, \ldots, 6$.

We identify these categories of similarity evolutions: *Always Equal:* The similarity value is 1 for all similarity snapshots. *Diverge from Equal:* The similarity value for some similarity snapshot is 1 but it is below 1 in the latest version. *Always Non-equal:* The similarity value is below 1 for all similarity snapshots. We could also apply more refined categories to distinguish whether similarity has increased or decreased overall. *Converge to Equal:* The similarity value for some similarity snapshot is below 1 but it is 1 in the latest version.

4. SIMILARITY ANNOTATIONS

We will now introduce annotations ('tags') for maintenance invariants or tasks, as a developer can apply them to similarity evolutions.

Maintenance invariants. These are invariants in so far as they constrain code changes of variants and their violation would trigger maintenance tasks; see below. We identified these invariants:

Maintain Equality: Past the commit point of adding the invariant, the two involved fragments may be changed, but they must remain equal to each other. When invariant violation is detected, then the maintenance task 'Restore Equality' (see below) is triggered. This may be an automated task—especially if only one of the fragments changed and thus, the direction of propagation is obvious.

Maintain Similarity: Past the commit point of adding the invariant, the similarity of the two involved fragments must remain equal. When invariant violation is detected, then the maintenance task 'Restore Similarity' (see below) is triggered. This is a manual task—the developer either confirms the new similarity as a consistent change or performs a 'co-change' to recover consistency.

Ignore Similarity: The two involved fragments are thereby described as being superficially similar. The developer will no longer be reminded of the similarity, when reviewing similarities. 'Ignore Similarity' is a trivial invariant that cannot be violated. All other invariants can be violated.

Maintenance tasks. These are tasks in that they require code changes of variants—to be performed either by automated change propagation or manually by the developer. We identified these tasks:

Restore Equality: The task is applicable if the involved fragments were once equal, but they are unequal in the latest version. The task may be performed in an automated manner, as discussed in §5.

Establish Equality: The developer needs to manually change the involved fragments so that they become equal again—they may never have been equal. Once the fragments are found to be equal, then the task is considered completed and the invariant 'Maintain Equality' is attached.

Remove Equality: The developer needs to manually change the two involved fragments so that they become unequal eventually. Once similarity analysis finds that the fragments are unequal, then the task is considered completed and the invariant 'Maintain Similarity' is attached. The task would be needed in the rare case that a developer finds two variants to use equal fragments while they should not, if they correctly implemented variability. This may happen if variants are committed after plain cloning, before adapting clones.

Restore Similarity: The task is applicable when the involved fragments were once more similar than in the latest version. The task can be accomplished by the developer by manually changing either or both fragments until the diff ratio reaches the previously highest value. Due to the sensitive nature of diff ratios, the developer may also manually confirm any specific diff ratio to mean that similarity was restored.

Increase Similarity: The developer needs to manually change the two involved fragments so that they become more similar eventually.

5. TOOL SUPPORT

Similarity management relies on tool support for the developer (in addition to just similarity analysis).

Reviewing similarities. Similarities need to be reviewed by the developer in a systematic manner. Two orthogonal views on similarities may be helpful in exploring similarities and editing annotations.

Figure 2: Commit-centric view on similarities.

Fig. 2 illustrates a commit-centric view on similarities. A list of all commits with at least one similarity evolution is provided. The commits are sorted by timestamp. The developer can explore a commit by expanding its variants, files, and fragments. At each level, the number of emerging similarity evolutions are shown. Once the developer has selected a fragment, all similar fragments at the same commit point are listed. Once similar fragments were selected, the similarity can be annotated; see below. There is also a variant-centric view.

Editing annotations. The developer can review specific fragment-level similarities, as shown in Fig. 3. An annotation is added or an existing annotation is changed. The developer may also enter some intent (i.e., 'comment'). Also, a direction (left to right or vice versa) may be specified to override a fragment in the interest of restoring or establishing equality.

Automated metadata update. The data of the similarity analysis and the annotations for maintenance invariants and tasks need to be continuously updated, as local edits are performed or new versions are pulled. The developer can request such an update and it is integrated into the workflow for accessing the version control system. The following rules apply in the given order:

Increment similarities: The similarity analysis is incrementally applied to new and revised source files.

Recommend invariant for unannotated similarities:
▷ 'Maintain Equality' — if diff ratio $= 1$ at HEAD.
▷ 'Maintain Similarity' — otherwise.
Subject to interactive tool support for a 'TODO list', these recommendations can be reviewed, confirmed, and altered by the developer.

Possibly turn invariant into task:
▷ 'Maintain Equality' $\mapsto$ 'Restore Equality'
if diff ratio < 1 at HEAD.
▷ 'Maintain Similarity' $\mapsto$ 'Restore Similarity'
if diff ratio has changed at HEAD.

Possibly turn task into invariant:
▷ 'Restore Equality' $\mapsto$ 'Maintain Equality'
▷ 'Establish Equality' $\mapsto$ 'Maintain Equality'
▷ 'Increase Similarity' $\mapsto$ 'Maintain Equality'
if diff ratio $= 1$ at HEAD.
▷ 'Increase Similarity' $\mapsto$ 'Maintain Similarity'
if the diff ratio has increased at HEAD.
▷ 'Restore Similarity' $\mapsto$ 'Maintain Similarity'
if the previous diff ratio is back at HEAD.

Possibly turn task into invariant recommendation:
▷ 'Restore Similarity' $\mapsto$ 'Maintain Similarity'
if the diff ratio has increased at HEAD.
▷ 'Remove Equality' $\mapsto$ 'Maintain Similarity'
if diff ratio < 1 at HEAD.

Automated change propagation. Whenever a 'Maintain Equality' invariant is violated, it is turned into a 'Restore Equality' task, as described above. Automated change propagation can be attempted afterwards. Given a task for similarity evolution with two fragments, if only one fragment was changed, then the change is propagated to the other

Figure 3: Two highly similar fragments (of test code) from even within a single variant are shown. The developer is about to attach the 'Maintain Similarity' invariant. This will also affect clones in other variants.

fragment. 'Three-way merge' is attempted, when both fragments have changed. If the automated attempt succeeds, then the task is considered completed and the invariant 'Maintain Equality' is attached. If the attempt fails, then the task remains pending until the developer restores equality manually, just like in the case of the task 'Establish Equality'. Regression testing is also used in this process.

Inferred annotations. Many annotations can be inferred (automatically) from existing annotations or membership of fragments in clone groups. There are inference rules in place to reduce the number of annotations that a developer needs to provide explicitly. For instance, if a and b are already annotated to establish equality with c, then the eventual invariant on the pair a and b to maintain equality is implied. The details of our inference heuristic are available online.

6. CASE STUDY

Repository of the study. The *101companies* project aggregates knowledge about software languages, technologies, and concepts on a wiki[2]; the project can be regarded as a software chrestomathy [8], that is, a collection of small software systems ('contributions') useful for teaching programming and software engineering. Each 'contribution' implements parts of a common feature model to demonstrate specific languages, technologies, and concepts. The different contributions are, in fact, variants where variance is present because (a) different features are implemented and (b) shared features are implemented in different ways. This paper focuses on the *101haskell* [9] subset of *101companies*'s contributions. This subset demonstrates programming concepts in and technologies for the Haskell language. The project is hosted as a repository on GitHub[3] and variants are organized by means of folders. The *101haskell* project hosts 36 variants. The Haskell code at the HEAD totals 4294 lines (NCLOC); the variants also contain code in other languages such as CSS, HTML, Makefile, SQL, and XML.

Similarity metrics. We use the following metrics to measure similarity and improvement thereof to answer the research question of §1.

Equality classes: We measure *similarity improvement* (more specifically gained equalities) by measuring the total number of non-singleton equality classes of fragments and the maximum, median, and average size of them.

Fragments: We measure the total number of fragments. To measure *similarity improvement*, we measure the number of distinct fragments (thus, counting each equality class just once), the number of shared fragments (thus, counting only members of a non-singleton equality class), the number of unshared fragments, and the median and average number of variants a fragment is shared in.

Similarities: We measure the median and average *similarity* of fragments at HEAD where the similarity value passes a threshold that is tuned to identify all interesting similarities; see below.

Similarity evolutions: We measure the total number of similarity evolutions per evolution category. For instance, an increasing number of 'Converge to Equal' would clearly indicate similarity improvement.

Annotations: We measure the total number of annotations per annotation category. For instance, a decreasing number of 'Restore Equality' in combination with an increasing number of 'Maintain Equality' would clearly indicate similarity improvement. We also measure the number of inferred annotations, also giving rise to a ratio of inferred annotations: the more annotations are inferred, the better the usability (scalability) of our apprroach.

Variants: We measure similarity improvement by the median and average uniqueness of all variants, that is, the number of fragments not shared in any other variants.

Step-wise process. The case study involves these steps:

Annotation 1: We annotate all similarity evolutions leading to an equality or where the fragments diverged intentionally or unintentionally. That is we annotate all similarity evolutions of categories 'Always Equal', 'Converge to Equal', 'Diverge from Equal'. We first annotate all equalities by annotating equality classes, thereby also maximizing the effectiveness of inference of annotations.

Restore equalities automatically: We trigger change propagation, thereby automatically restoring equalities of fragments that diverged unintentionally. It turned out that all the tasks could be executed automatically.

Annotation 2: We annotate all similarity evolutions where the underlying fragments have never been equalities. That is, we annotate all similarity evolutions of category 'Always Non-equal' and therefore annotate all remaining evolutions. We applied a threshold of 0.8 for the diff ratio, as we noticed that interesting similarities stopped to show up at this value.

Establish equalities manually: We manually edited fragments.

[2]http://101companies.org/wiki
[3]http://github.com/101companies/101haskell

That is, we look at such tasks in the TODO list and complete them. Then, we performed automated metadata update.

Increase similarities manually: We manually increase the similarity of fragment pairs until a satisfactory similarity value is reached. That is, we work on 'Increase Similarity' tasks. Again, we performed automated metadata update.

Results of the case study. Table 1 addresses the research question for the case study. Initially, the median vs. average similarity of fragments were 98.85% vs. 94.18% for the threshold. The process led to an improvement: 100.00% vs. 96.80%. The number of distinct fragments was decreased by 5.85% from 632 to 595, while the number of shared fragments could be increased by 6.96% from 388 to 415. Initially, fragments where shared in 1 variant median and 1.46 variants in average. The average was improved to 1.55 variants. The distinctness of fragments is also reflected by the equality classes. Initially, 95 non-singleton equality classes were present where the maximum size was 13. The number of equality classes was reduced to 85 where the maximum sized class now contains 20 fragments. The uniqueness of variants in terms of non-shared fragments was also changed by improving similarities. Here, the median uniqueness of variants could be decreased by 5.13% from 52.47% to 47.34%, while the average uniqueness was decreased by 3.34% from 53.34% to 50.00%. Improvements can also be observed by looking at the metrics for similarity evolution categories. Initially, 'Always Equal' and 'Always Similar' were the most common categories of similarity evolutions with 38.1% and 35.0%. Only 8.6% could be categorized as 'Converge to Equal' and 18.2% as 'Diverge from Equal'. However by restoring and establishing equalities finally, 32.5% of all evolutions were in category 'Converge to Equal'. Overall, 324 similarity evolutions were annotated with 'Restore Equality'. That is, we identified 324 divergences as unintentional. The annotations could all be executed automatically using change propagation. We did not annotate any equality with 'Remove Equality', that is, all present equalities were intentional and should not become non-equalities. With a similarity threshold of 0.8, 421 similarity evolutions were considered irrelevant for maintenance and therefore annotated with 'Ignore Similarity'. The high number shows that the threshold appears to be low enough to find the interesting similarities. Out of all final annotations, 70.0% could be inferred.

Threats to validity. We picked *101haskell* because we knew of the involved sharing and we suspected unintended dissimilarities including divergence from equality, which simplifies validation of all measurements. However, other repositories may clearly differ in terms of the degree of sharing, the numbers for different similarity evolutions, etc.

We do not provide guidance with regard to the choice of syntactic levels for fragment extraction. For instance, one could consider local functions, as opposed to just top-level functions. Also, fragment tracking may fail to find snapshot links. In the case study, we manually inspected terminated tracks; they turned out to be terminated for a 'good reason' such as in the case of a function being removed.

The assignment of annotations was carried by the first author; the second author was consulted for confirming a small number of difficult cases. The overall process was straightforward to carry out for the case study, given the size of the code base and the authors' familiarity with *101haskell*. Such a process could be time-consuming and it could require communication of different stakeholders such as testers, architects, and developers.

7. RELATED WORK

Variability. This work realizes a concrete, albeit limited instance of the vision for a *virtual platform* for flexible product line engineering [1] even though no explicit feature model is used here. Our work provides support in a situation where inconsistencies (e.g., divergence from perfect clones) were introduced already. Fischer et al. have also proposed an approach that enhances basic cloning with systematic reuse of variants [5].

Change propagation. Hemel et al. extracted the variability in a set of Linux variants [6]. Their work discusses that changes are often not propagated from the main Linux kernel to its variants. Nadi et al. study anomalies in the variance of linux by asking about their causes, the time it takes for them to get fixed, and how they are fixed [12]. The term 'late propagation' refers to a pattern of commits where clones first diverge to converge only later again. Mondal et al. have studied the late propagation of near-miss clones [11], while Barbour et al. have defined types of late propagation and discuss why such propagation is indeed harmful [2]. Our approach helps with late propagation in that the underlying analysis reliably determines cases of divergence to act on. Our approach can also be used in a continuous manner to avoid late propagation.

Developer annotations. Nguyen et al. utilize annotations to indicate the intents of cloning in their work on JSync [13]. In our work, annotations are used to express maintenance tasks. Developer annotations are used more broadly in software development. For instance, Brühlmann et al. use annotations in a generic approach to enrich reverse engineering with human knowledge [3]. Lucas et al. [10] describe reuse contracts for object-oriented software systems. These contracts are essentially annotations for change management in that they can be used both to signal changes to reusing parties and to assess the impact of changes.

8. CONCLUSION

The paper shows how the similarities in software variants can be managed by means of clone detection techniques and annotations proxying for maintenance invariants and tasks. In the case study, we recovered full consistency of all variants and provided developers with systematic, explorable information regarding the similarity of the variants. Overall, the reported research provides an important data point for combining clone and variability management in that 'clone-and-own' is refined into a proper product-line engineering approach.

Acknowledgment. The authors would like to thank their coauthors in a related paper [1] and specifically Michal Antkiewicz, Krzysztof Czarnecki, and Andrzej Wasowski for fruitful discussions on the reported research.

	INIT	A1	ARE	2A	MEE	MIS	Δ
Equality classes							
total (non-trivial)	95		93		85	85	−10
max size (non-trivial)	13		20		20	20	+7
median size (non-trivial)	3		4		4	4	+1
average size (non-trivial)	4.08		5.02		4.89	4.89	+0.84
Fragments							
total	925		925		925	925	±0
distinct	632		617		595	595	−37
shared	388		401		415	415	+27
unshared	537		524		510	510	−27
variants sharing (median)	1		1		1	1	±0
variants sharing (average)	1.46		1.51		1.55	1.55	+0.09
Similarities							
median	0.98850		1.0		1.0	1.0	+1.163%
average	0.94186		0.96495		0.96795	0.96802	+2.777%
Similarity evolutions							
Always Equal	777		777		780	780	+3
Converge to Equal	176		582		712	712	+536
Diverge from Equal	371		47		47	47	−324
Always Similar	714		714		650	650	−64
Annotations							
Maintain Equality (auto)	0 (0)	953 (858)	1359 (1264)	1359 (1264)	1492 (1387)	1492 (1387)	+1492
Restore Equality (auto)	0 (0)	324 (212)	0 (0)	0 (0)	0 (0)	0 (0)	±0
Establish Equality (auto)	0 (0)	0 (0)	0 (0)	130 (67)	0 (0)	0 (0)	±0
Remove Equality	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	±0
Increase Similarity (auto)	0 (0)	3 (0)	3 (0)	4 (0)	4 (0)	0 (0)	±0
Maintain Similarity (auto)	0 (0)	44 (20)	44 (20)	206 (123)	272 (145)	276 (145)	+276
Ignore Similarity	0 (0)	0 (0)	0 (0)	421 (0)	421 (0)	421 (0)	+421
Variants							
uniqueness (median)	52.47%		49.00%		47.34%	47.34%	−5.13%
uniqueness (average)	53.34%		51.39%		50.00%	50.00%	−3.34%

Table 1: Metrics of the stepwise case study ("auto" refers to the number of annotations that could be inferred; INIT refers to initial results; A1, ARE, A2, MEE, and MIS refer to results after each step of §6 respectively).

9. REFERENCES

[1] M. Antkiewicz, W. Ji, T. Berger, K. Czarnecki, T. Schmorleiz, R. Lämmel, S. Stanciulescu, A. Wasowski, and I. Schaefer. Flexible product line engineering with a virtual platform. In *Proc. of ICSE 2014*, pages 532–535. ACM, 2014.

[2] L. Barbour, F. Khomh, and Y. Zou. Late propagation in software clones. In *Proc. of ICSM 2011*, pages 273–282. IEEE, 2011.

[3] A. Brühlmann, T. Gîrba, O. Greevy, and O. Nierstrasz. Enriching reverse engineering with annotations. In *Model Driven Engineering Languages and Systems*, pages 660–674. Springer, 2008.

[4] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki. An exploratory study of cloning in industrial software product lines. In *Proc. of CSMR 2013*, pages 25–34. IEEE, 2013.

[5] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. Enhancing clone-and-own with systematic reuse for developing software variants. In *Proc. of ICSME 2014*, pages 391–400. IEEE, 2014.

[6] A. Hemel and R. Koschke. Reverse engineering variability in source code using clone detection: A case study for linux variants of consumer electronic devices. In *Proc. of WCRE 2012*, pages 357–366. IEEE, 2012.

[7] M. Kim and D. Notkin. Using a clone genealogy extractor for understanding and supporting evolution of code clones. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–5, 2005.

[8] R. Lämmel. Software chrestomathies. *Science of Computer Programming*, 2013.

[9] R. Lämmel, T. Schmorleiz, and A. Varanovich. The 101haskell chrestomathy: A whole bunch of learnable lambdas. In *Proc. of IFL 2013*, pages 25:25–25:36. ACM, 2014.

[10] C. Lucas, P. Steyaert, and K. Mens. Managing Software Evolution through Reuse Contracts. In *Proc. of CSMR 1997*, pages 165–170. IEEE Computer Society, 1997.

[11] M. Mondal, C. K. Roy, and K. A. Schneider. Late propagation in near-miss clones: An empirical study. *Electronic Communications of the EASST*, 63, 2014.

[12] S. Nadi, C. Dietrich, R. Tartler, R. C. Holt, and D. Lohmann. Linux variability anomalies: What causes them and how do they get fixed? In *Proc. of MSR 2013*, pages 111–120. IEEE, 2013.

[13] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Clone management for evolving software. *Software Engineering, IEEE Transactions on*, 38(5):1008–1026, 2012.

[14] C. K. Roy and J. R. Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proc. of ICPC 2008*, pages 172–181. IEEE, 2008.

[15] J. Rubin and M. Chechik. A framework for managing cloned product variants. In *Proc. of ICSE 2013*, pages 1233–1236. IEEE, 2013.

[16] J. Rubin, K. Czarnecki, and M. Chechik. Managing cloned variants: A framework and experience. In *Proc. of SPLC 2013*, pages 101–110. ACM, 2013.

[17] S. Xie, F. Khomh, and Y. Zou. An empirical study of the fault-proneness of clone mutation and clone migration. In *Proc. of MSR 2013*, pages 149–158. IEEE Press, 2013.