Changes as First-Class Citizens: A Research Perspective on Modern Software Tooling

QUINTEN DAVID SOETENS, Dept. of Mathematics and Computer Science, University of Antwerp
ROMAIN ROBBES, Computer Science Department (DCC), University of Chile
SERGE DEMEYER, Dept. of Mathematics and Computer Science, University of Antwerp

Software must evolve to keep up with an ever-changing context, the real world. We discuss an emergent trend in software evolution research revolving around the central notion that drives evolution: Change. By reifying change, and by modelling it as a first-class entity, researchers can now analyse the complex phenomenon known as software evolution with an unprecedented degree of accuracy. We present a Systematic Mapping Study of 86 articles to give an overview on the state of the art in this area of research and present a roadmap with open issues and future directions.

CCS Concepts: • **Software and its engineering** → **Software configuration management and version control systems**; **Software development techniques**; *Software maintenance tools*; Software libraries and repositories;

Additional Key Words and Phrases: Systematic mapping study, fine-grained changes, change recording, change distilling, atomic change operations, fine-grained edit operations

ACM Reference Format:
Quinten David Soetens, Romain Robbes, and Serge Demeyer. 2017. Changes as first-class citizens: A research perspective on modern software tooling. ACM Comput. Surv. 50, 2, Article 18 (April 2017), 38 pages.
DOI: http://dx.doi.org/10.1145/3038926

1. INTRODUCTION

Software is vital to our society and, consequently, the software engineering community is investigating means to produce reliable software. For a long time, reliable software was seen as software “without bugs.” Today, however, reliable software has come to mean “easy to adapt” because of the constant pressure to change. With the use of continuous integration, for instance, it is possible to shorten the release cycle from a couple of months to a few weeks, which implies that bugs get fixed faster as well [Khomh et al. 2015]. Continuous delivery takes the concept of rapid release cycles to the extreme: Updates get pushed to the customer multiple times per day. Facebook, for example, pushes new updates to production twice a day; within Amazon, it happens on average every 11.6s [Jenkins 2011].

This work is partially supported by FONDECYT (Chile) Project No. 1151195 and partially sponsored by the Institute for the Promotion of Innovation through Science and Technology in Flanders (Belgium) through a project entitled “Change-centric Quality Assurance” (CHAQ) with number 120028.
Authors’ addresses: Q. D. Soetens (current address), OM Partners nv, Koralenhoeve 23, 2160 Wommelgem (Antwerpen), Belgium; email: qsoetens@ompartners.com; S. Demeyer, Dept. of Mathematics and Computer Science, University of Antwerp, Middelheimlaan 1, 2020 Antwerpen, Belgium; email: serge.demeyer@uantwerpen.be; R. Robbes (current address), Free University of Bozen-Bolzano, Faculty of Computer Science, Piazza Domenicani 3, 39100 Bolzano (BZ), Italy; email: romain.robbes@gmail.com.

DOI: http://dx.doi.org/10.1145/3038926

Indeed, software evolution is inevitable [Lehman and Belady 1985]: In the software industry, the cost of maintaining and changing software greatly outweighs the initial cost of development. As a consequence, organisations producing software must seek for a delicate balance between two opposing forces: striving for *reliability* as well as for *agility*. In the former, organisations optimise for perfection; in the latter, they optimise for development speed.

While the tension between *reliability* and *agility* is an essential ingredient of modern software development, part of the balancing act can be alleviated with proper tooling. Since the mid-2000s, there has been a trend towards devising software engineering tools in which the act of changing software is represented as a first-class entity. As such, a change becomes tangible in the form of an object that can be directly analysed or manipulated. This idea can be traced back to the mid 1990s to operation-based versioning systems [Lippe and van Oosterom 1992], which is now experiencing a revival. Moreover, with the adoption of distributed version control systems like git and hosting services like GitHub and BitBucket, the granularity of tracking changes has fundamentally shifted. The yearly Eclipse Community Survey Report, for instance, shows a growing trend in the adoption of subversion (SVN) and git, with git rapidly surpassing SVN [Skerrett 2013]. As such, these *change-based* approaches have spread beyond mere versioning and have proven useful in a variety of contexts, including, but not limited to, collaboration and awareness, recommendation systems, change impact analysis, regression testing, reverse engineering, and many more.

In this article, we summarise the research contributions that are based on a first-class representation of changes to software systems and highlight future applications. We report on a Systematic Mapping Study that we performed in order to provide an overview on the state of the art in reified source-code changes. A Systematic Mapping Study is designed to provide a wide overview of an area of research in order to establish if there exists evidence on a particular topic and provide an indication of the quantity of said evidence [Budgen et al. 2008; Kitchenham and Charters 2007; Petersen et al. 2015, 2008]. It can also identify areas where more primary research is needed or areas that are ripe for a more in-depth literature review. As our goal is to present an overview on the state of the art in change reification as well as to present a roadmap for future research directions, a Systematic Mapping Study is an ideal approach.

An overall research question (RQ) to summarise our goal can thus be formulated:

RQ: *To what purposes has change reification been used, what is the evidence for the success of approaches employing it, and what directions of further research are considered important?*

The rest of this article is structured as follows. In Section 2 we give a short introduction into the basics of change reification. Section 3 explains in detail the process of a Systematic Mapping Study and explains how we selected our primary studies. We continue with the classification of the articles and answer each of the research questions (posed in Section 3) in Sections 4, 5, 6, and 7. We wrap up the article with the conclusions and our vision for the future in software development in Section 8.

2. THE BASICS OF CHANGE REIFICATION

The key concept behind change reification approaches is that changes are represented as first-class entities, objects that are directly manipulable by programs and humans. As such, we define a first-class change as a tangible object that represents an action that changes a software system. They describe meaningful operations on the data they act on, which is computer programs. This is in contrast with other models of the evolution of software systems, in which a change is the implicit difference between two versions. For instance, the change representation found in most versioning systems is still the text-based δ between the versions [Robbes and Lanza 2005]. The entities affected by first-class changes instead are actual program entities: packages, classes, methods, and even statements.

A further characteristic shared by a number of change reification approaches is that these changes are recorded through the Integrated Development Environment (IDE), *while developers program*. This retains the precision of the change sequence that originated from the developers themselves, instead of having to reconstruct it from the limited data in source-code management systems. The gathered change information can be leveraged by analyses, which benefit from the accuracy of the recorded information. The kinds of changes, and types of analyses supported, depends on the model chosen.

There are three ways of obtaining changes while software engineers are working: change-oriented programming, change recording, and change recovery.

Change-Oriented Programming (or ChOP) is a programming style that centralises change [Ebraert et al. 2007a]. To create a piece of software in ChOP, a programmer does not need to write large chunks of source code but rather uses the IDE to *explicitly apply changes* to the source code. This approach is already quite common for very specific cases in most mainstream IDEs. For instance, Eclipse provides the possibility of adding new classes or creating getter and setter methods by the click of a button and interacting with the IDE through dialogs. Additionally, many tools support the automatic execution of refactorings, which can be considered composite changes consisting of several atomic changes. In the end, it is the IDE (or the refactoring tool) that produces the source code and inserts this code in the specified location. In pure ChOP, the entire system is built by having the IDE execute changes. This approach, however, is unrealistic, as programmers cannot be expected to respond to a dialog for every fine-grained change that they want to perform.

Change Recording (or Change Logging) silently records the activities of the programmers while they are working, and infers the changes that they produce [Robbes and Lanza 2007a]. For instance, if the programmer changes a method, a differencing algorithm will infer which statements were added, changed, and removed. Additionally, the change recorder can maintain dependencies between the changes. This approach can also be mixed with the first one by recording the activity of the code wizards and refactoring engines.

Change Recovery (or Change Distilling) recreates changes after they were performed by analysing two subsequent versions of a system [Fluri et al. 2007b]. This approach finds the shortest sequence of changes to transform version “a” into version “b.”

The difference between these three approaches to change reification is in how close the inferred changes lie to the actual changes a developer performed. Change recovery tries to reconstruct changes from the limited information in source-code repositories, whereas change recording obtains changes that show what actually happened. Pure ChOP would be the best approach, since the changes obtained are exactly those changes the developer performed.

Taxonomy of Changes. Lehnert et al. have reviewed different types of changes and their effect on change impact analysis and regression testing [Lehnert et al. 2012]. Based on their review, they proposed a taxonomy of changes (see Figure 1). They identify four classification criteria: the *Abstraction Level*, the *Composition Type*, the *Type of Operation*, and the *Scope of Change*.

—In the abstraction level, they distinguish between *Generic* and *Concrete* changes. A generic change needs to be instantiated to become a concrete one; for instance, a

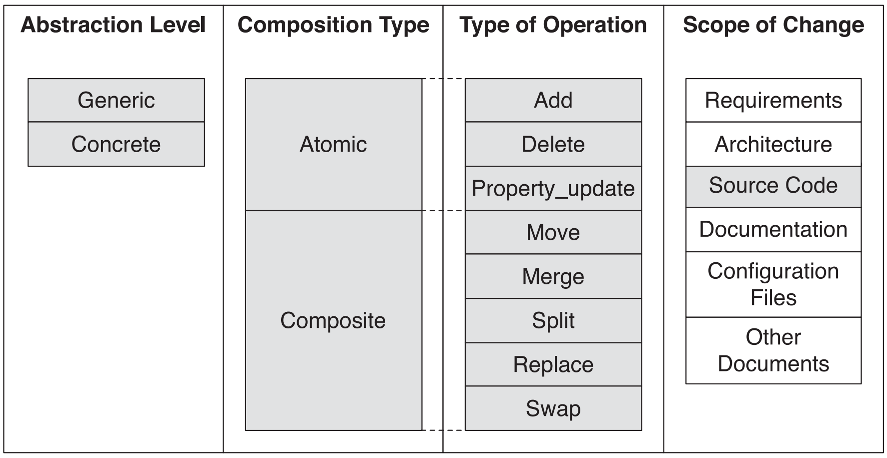

Fig. 1. Taxonomy of Changes (taken from Lehnert et al. 2012).

refactoring can be defined in a generic way, which allows it to be used in several concrete instances.

—A change can be either atomic or composite. A composite change can be decomposed into smaller changes that together represent a meaningful operation. Atomic changes are then the basic changes that cannot be further decomposed.

—The third criterion (type of operation) directly reflects the composition type and says that there are three atomic operations (*Add*, *Delete*, and *Property_update*) and the others (*Move*, *Merge*, *Split*, *Replace* or *Swap*) are examples of composite changes.

—The scope of change is used to associate a change to the type of artefact the change can be applied to like source code, architectural models, or even text documents.

In this article, we limit ourselves to atomic and composite changes (both generic and concrete) that act on source code (see the grayed out parts in Figure 1). We did find that some articles straddled the boundaries, such as articles operating on models in the Unified Modelling Language (UML) extracted from source code. We kept these articles on the rationale that the original material is the actual source code, not the UML models.

Scope of Change—Models. Note that there is a large body of research on change reification in modelling languages as well. It is beyond the scope of this mapping study to incorporate this body of work. However, we provide some recent examples to demonstrate the breadth of the field. Just like with source code, change-based approaches for models have proven useful in a variety of contexts. Taentzer et al., for instance, demonstrated that by representing model structures and their changes over time, it is possible to construct a richer version control system for models [Taentzer et al. 2014]. Johann et al. showed that explicit changes allow for incremental model transformations, which avoids the costly matching operations on those parts of the model that cannot be affected [Johann and Egyed 2004]. Blanc et al.argued that detection and resolution of structural inconsistencies becomes much easier when models are represented by sequences of elementary construction operations [Blanc et al. 2008]. Bennaceur et al. discuss challenges when developing mechanisms to support runtime software adaptation [Bennaceur et al. 2014]. Dávid et al. integrated incremental model queries, complex event processing, and reactive transformation techniques to manipulate models that are only available as a stream of data and hence cannot be fully materialised [Dávid et al. 2016].

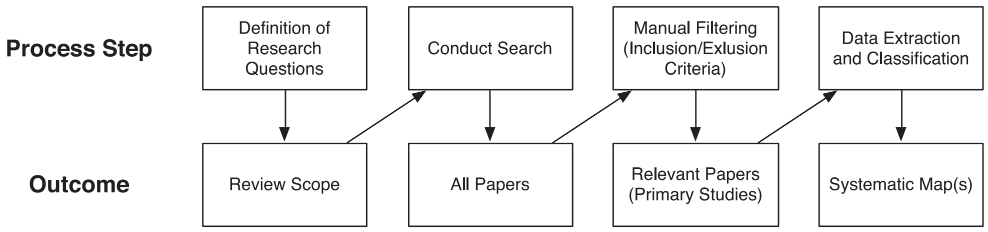
Process Step
Definition of Research Questions
Conduct Search
Manual Filtering (Inclusion/Exlusion Criteria)
Data Extraction and Classification
Outcome
Review Scope
All Papers
Relevant Papers (Primary Studies)
Systematic Map(s)

Fig. 2. Systematic Mapping Study process.

3. SYSTEMATIC MAPPING STUDY SETUP

In this section, we present the process that we followed while performing our systematic mapping study. We begin by quoting some definitions for Systematic Mapping Studies:

> *Systematic Mapping Studies (also known as Scoping Studies) are designed to provide a wide overview of a research area, to establish if research evidence exists on a topic and provide an indication of the quantity of the evidence. The results of a mapping study can identify areas suitable for conducting Systematic Literature Reviews and also areas where a primary study is more appropriate.* [Kitchenham and Charters 2007]

> *A systematic mapping study provides a structure of the type of research reports and results that have been published by categorizing them. It often gives a visual summary, the map, of its results. It requires less effort* [than a systematic literature review] *while providing a more coarse-grained overview.* [Petersen et al. 2008]

We interpret this as follows. A Systematic Literature Review is an in-depth study of the complete corpus of research performed in a particular field, whereas a Systematic Mapping Study is a more shallow study meant to provide an overview of a particular field.

In our case, we will apply this technique in the area of change reification. We followed the updated guidelines for conducting and reporting systematic mapping studies [Petersen et al. 2015]. We start by conducting a search for relevant articles and filter them according to pre-defined inclusion and exclusion criteria. We then proceed to classify the articles and extract data in order to answer our research questions. In Section 3.1, we describe in detail the process that we used as well as motivate some of the decisions we made. We end by giving an overview of some threats to the validity of our process and outcome and how we alleviate these threats in Section 3.2.

3.1. Process

The essential steps in our process are depicted in Figure 2. These are as follows: the definition of our research questions; conducting the search for possible relevant articles; manually filtering these candidates using inclusion/exclusion criteria; and, finally, extracting the data we need to classify the primary studies.

Definition of Research Questions. We start by establishing the research questions that we pose when performing this Systematic Mapping Study.

RQ1: *Where and when have studies that use reified source-code changes been published?* This is a standard research question in a Systematic Mapping Study and serves to assess whether there is gaining popularity in the topic.

RQ2: *What are the commonalities and differences in the change models adopted by the primary studies? Can we extract a unified model for changes?* Since the change model forms the basis for each approach, it is important to investigate the different representations and their limitations.

RQ3: *What are the applications in which reified source-code changes are used in the primary studies?* With this research question, we want to provide an overview of areas where analysis of reified source-code changes has already proven to be useful. By looking at the number of studies reported in a particular application, we can identify areas where more primary studies are needed or areas in which a more in-depth literature review is needed.

RQ4: *What is the future work indicated by the primary studies?* Contrary to **RQ3**, here we want to summarise the vision that researchers in this domain put forward and in which as-yet unexplored areas they propose change reification might prove beneficial. Additionally, we would like to see if there is any consensus regarding future work among the studies analysed.

All of these research questions will, each in part, solve our overall research question.

RQ: *To what purposes has change reification been used, what is the evidence for the success of approaches employing it, and what directions of further research are considered important?*

Conducting the Search. Our search strategy is composed of two steps: First, we query the ACM Digital Library[1] and the IEEE Xplore Digital Library,[2] and, second, we use the possibly relevant articles from those queries as a starting point for forward snowballing (i.e., recursively check the references of the relevant articles for other possibly relevant articles) [Wohlin 2014]. In the first step, we chose to only query the ACM Digital Library and IEEE Xplore, ignoring other publishers, like Springer, Elsevier, or Wiley, and other query engines, like Google Scholar, Sciencedirect, or Scopus.

We find that the ACM Digital Library and IEEE Xplore provide a sufficient set of articles to serve as a starting point for the snowballing process. Moreover, the ACM Digital Library also contains articles from other publishers (like IEEE or Springer). The snowballing process allowed us to include more articles from other publishers.

This is sufficient for our purposes as a mapping study, where we present an overview of the research topics in this domain and a roadmap for future work. Yet a more in-depth literature review should also query the other publishers and enhance it with at least one query engine like Google Scholar.

We performed the first query on the IEEE Xplore Digital Library using only Metadata (i.e., not searching the full text); this includes, among others, the titles and abstracts of the articles. Note that in the IEEE Xplore Digital Library we can perform queries using the wildcard "`*`" to search for similar words. For instance, "`logg*`" matches both "`logged`" and "`logging`." What we searched for are articles about software engineering or software evolution where fine-grained (atomic) changes (edit operations) are either distilled (recovered) or logged (recorded). This is then translated into the following query:

```
(change OR "edit operation") AND (distil* OR record* OR logg* OR
recover* OR "fine grained" OR atomic) AND (software)
```

The IEEE Xplore Digital Library returned 1,839 results when running this query.

Performing the same query on the ACM Digital Library returned over 90,000 results. This is because, for this database, we have to manually indicate the metadata that need

[1]http://dl.acm.org.
[2]http://ieeexplore.ieee.org.

Table I. Inclusion and Exclusion Criteria Used While Filtering for Relevant Articles

Inclusion Criteria	
IC1	Materials that are articles (including journal articles, conference articles, workshop articles, tool demonstration articles, or short articles at conferences) that use an explicit (first-class) representation for source-code changes.
Exclusion Criteria	
EC1	Materials that are from a different domain (i.e., not about software engineering).
EC2	Materials that are books, Ph.D. theses, doctoral symposium articles, technical reports, abstracts for keynotes, presentations, posters, white papers, or blog posts.
EC3	Materials that are shorter than four pages.
EC4	Materials that use explicit representation for changes to things other than source code (e.g., changes in structured text documents).
EC5	Materials that use implicit representation for changes (i.e., the changes themselves are not modelled but rather different versions of the data is used)
EC6	Where studies were published multiple times (e.g., first as a conference article and then as a journal article) only the most recent publication was included.

to be queried, otherwise it will also search the full text. As such, we needed to transform the query into the following form:

```
(Abstract:change OR Abstract:"edit operation" OR Title:change OR
Title:"edit operation") AND (Abstract:distilled OR Abstract:recorded
OR Abstract:logged OR Abstract:recovered OR Abstract:"fine grained" OR
Abstract:atomic OR Title:distilled OR Title:recorded OR Title:logged OR
Title:recovered OR Title:"fine grained" OR Title:atomic) AND software
```

The ACM Digital Library returned 1,003 results when running this query.

After conducting these queries, we did a preliminary filtering based on titles and abstracts to find a smaller set of possibly relevant articles (mostly based on exclusion criterium **EC1**). This resulted in a set of 110 articles. With this smaller set of articles, we proceeded to use forward snowball sampling (i.e., we looked at the reference list in each article to find possibly relevant work that is not yet in our set) to increase our set of possibly relevant articles. This was done repeatedly until no more articles could be added, giving us a total of 169 articles.

During the process, we repeatedly validated the working set of articles against a reference set of articles we expected to be included based on our prior work in the field. This reference set consisted of 37 articles drawn from workshops like Mining Software Repositories (MSR), conferences like International Conference on Software Engineering (ICSE), and journals like *IEEE Transactions on Software Engineering* (TSE). As a result, after the snowballing step, we added three articles that we were aware of but that were not found by the search queries: De Roover et al. [2014], Ebraert et al. [2011], and Soetens et al. [2013b]. These papers' reference lists were also checked for any missing works.

Manual Filtering. Table I shows the inclusion criterium and each of the exclusion criteria that we used. Both the first and second authors of this article then individually classified the remaining articles into "*Relevant*," "*Out of Scope*," "*Not Relevant*" and "*Unsure*" based on the remaining exclusion criteria (**EC2**, **EC3**, **EC4**, and **EC5**). Exclusion criteria **EC1** and **EC5** are cause to label an article as *Not Relevant*. Exclusion criteria **EC2**, **EC3**, **EC4**, and **EC6** are cause to label an article *Out of Scope*. When it is not 100% certain into which category a article should be classified, we label it as *Unsure*. An *Unsure* article was automatically scheduled for discussion between the authors to determine its final status.

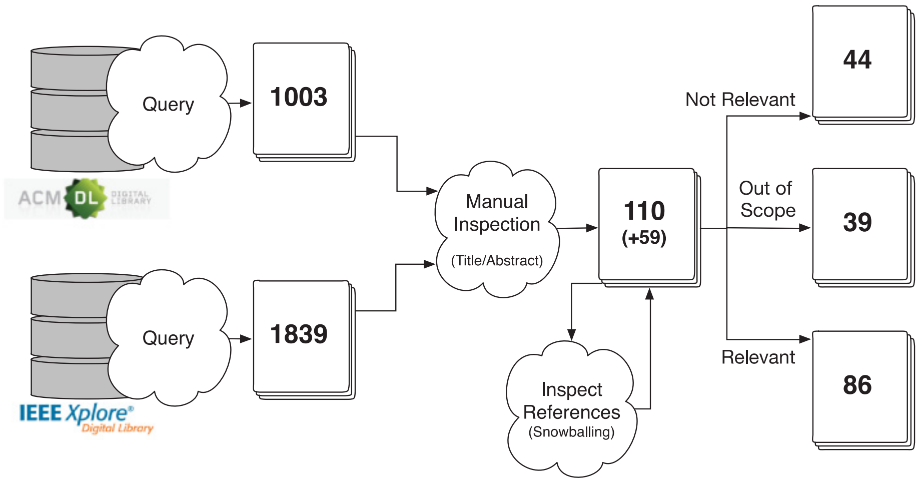

Fig. 3. Article selection process.

For 110 of 169 (65%) articles, we had the same classification: 75 *Relevant*, 19 *Not Relevant*, 13 *Out of Scope*, and 3 *Unsure*. For 27 articles (16%), we agreed on the fact that the article did not belong in the category *Relevant*; however, there was a disagreement on which of the other three categories the article belonged to. When one was *Unsure*, it was discussed; in the other cases, the article was not included in the final set of primary articles.

For the remaining 32 articles, we discussed the reasons for excluding or including the article and came to a consensus by approving 11 more articles for inclusion in the set articles. The whole search process is summarised in Figure 3 and ultimately resulted in a set of primary studies counting 86 relevant articles.

Data Extraction. For each of the research questions, we extracted the required information from each of the articles. For **RQ1**, we extracted the year in which it was published, the venue at which it was published, and also what kind of publication it was (a journal article, conference article, workshop article, tool demonstration article, or short article). For **RQ2**, we were interested in learning about the change model that was employed. For this, we only looked at those articles that either introduce a new change model or that modify/extend a previous change model. Other articles that merely use a previously introduced change model are not considered for this research question. For **RQ3**, we looked at the applications for which the changes are used. To this end, we analysed the Abstract, Introduction, and Conclusions for each article to identify keywords for the applications that the article presents. We then proceeded with a card-sorting strategy to classify the identified keywords into categories. For **RQ4**, we extracted all sections and paragraphs of future work from each of the articles by performing a search in the articles for the words "future" (as in "future work"), "plan" (as in " we plan to ..."), or "will" (as in "we will ..."). Again from each of these future work paragraphs, we extracted keywords to get an idea of popular or interesting lines of future work.

3.2. Threats to Validity

We now identify factors that may jeopardise the validity of our results and the actions we took to alleviate the risk. Consistent with the updated guidelines for reporting systematic mapping studies, we organise the identified threats into five categories as follows [Petersen et al. 2015]:

Descriptive Validity – To what extent are the observations described accurately and objectively? For each research question, it is known beforehand what data need to be extracted from the article. For **RQ1**, this is straightforward metadata from the article: *publication venue*, *publication type*, and *year of publication*. For the other research questions, the process is more intricate: We derived keywords on the applications and future works from particular sections or paragraphs in the article that we then classified. As such, we rely on the clarity of the articles themselves. For instance, to identify future work, we mined the article for the words "future" (as in "future work"), "plan" (as in "we plan to"), or "will" (as in "we will"). However, if they present their future work in a different manner (e.g., "As a follow-up study, we intend to"), we would not find it. Using this strategy, we were able to identify future work in all but 13 articles; in these cases, we did a more thorough scan of the article to make sure there was indeed no future work explicitly mentioned.

Theoretical Validity – Do we capture what we intend to capture? When looking at the process steps, there are factors that influence the theoretical validity in each of the last three steps:

Conducting Search. First, the set of primary studies that we used for our analysis might not be complete. There are several factors that influence this, including the manual filtering step, which will be discussed next. Yet even before the manual filtering, some relevant articles could be missing after conducting the search in the first place. In the first place, the query itself could influence the results. To minimise this threat, we had a certain number of studies that we were aware of at the beginning, which we used as a reference set of articles in order to assess the performance of the queries. Second, we only queried two digital libraries (ACM DL and IEEE Xplore), and hence important articles that have been published elsewhere (Springer, Elsevier, etc.) could be missing from the base set of articles. To reduce this threat, we extended our initially filtered set of articles through snowball sampling; that is, we searched the references of the sampled articles for studies that were missing from our set. We could have further extended the search with other types of sampling: backwards snowball sampling (looking for articles that have a reference to one of the articles in our set) or checking the publication lists on the webpages of key authors. Of our initial set, and after conducting snowball sampling, only three articles were not encountered, which gives us confidence on the coverage of our study; we added these three manually for the sake of completeness.

An additional issue concerns recent articles. Since all articles take time to be cited, the odds of adding recent articles through snowball sampling are slim at best; as such, we think we likely missed several recent articles. We can observe this with the statistics from 2015, which exhibit a sharp drop in the number of articles published. Note that the drop is also attributable to the fact that we conducted the search halfway through 2015 and also to the additional delay that conference and workshop articles take to appear in digital libraries.

Manual Filtering. During manual filtering, there is a threat of researcher bias appearing. In order to lessen this threat, the first two authors separately performed the manual filtering by classifying the articles as "*Relevant,*" "*Out of Scope*," "*Not Relevant*," and "*Unsure.*" Articles where there was disagreement were scheduled for discussion in order to come to a consensus.

Data Extraction. In this phase, researcher bias is also a threat. To alleviate this threat, the first author did the data extraction and classification, which was later reviewed by the second and third authors.

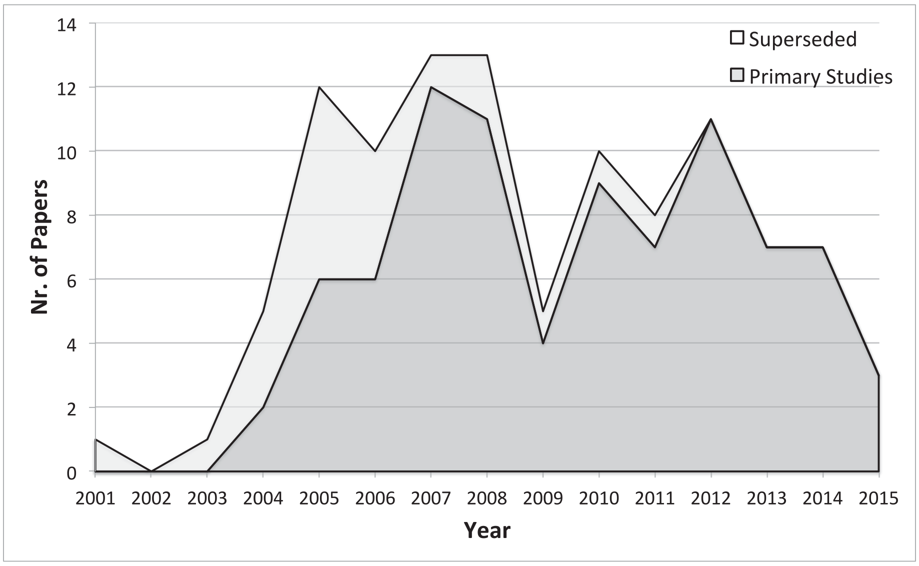

Fig. 4. Number of articles per year.

Generalizability – To what extent is it possible to generalise the findings? In this article, we focus on reified changes in the scope of source code. However, the findings with regards to the change models used, as well as the future work identified, could also be applicable to changes in other scopes (e.g., architecture models, structured documents, etc.).

Interpretive Validity – Are the conclusions drawn reasonable given the data? A threat to the interpretation of the data is researcher bias. Indeed, all three of the authors have authored or co-authored articles in the set of primary studies. The experience with the topic, however, can help in extracting and interpreting the data. Indeed, the guidelines by Petersen et al. suggest using an expert's opinion to evaluate the search and the data extraction [Petersen et al. 2015].

Repeatability – Is the research process reported in sufficient detail? The repeatability of our mapping study is achieved by accurately describing the followed process in Section 3.1, as well as by elaborating on possible threats to the validity and the actions taken to reduce them in this very section.

4. TOPIC POPULARITY

In this first section on the results of our mapping study, we provide an answer for **RQ1** (*Where and when have studies that use reified source-code changes been published?*) in order to provide some insight into the popularity and relevance of this topic within the software engineering research community. Note that for this analysis, we not only looked at the final set of primary studies but also included the articles that were excluded by criteria **EC6** (i.e., articles that are superseded by one of the articles in the final set of primary studies). This is done to get a more precise trend of the topics popularity over time.

4.1. Frequency of Publication

In Figure 4, we show the number of articles, including articles that are superseded, that we found for each year. The oldest articles date back to 2001 [Ryder and Tip

Table II. Number of Articles and Authors of Several Literature Studies

Study	Articles	Authors
This study	86	111
This study (excluding superseded works)	65	111
Kagdi et al. [2007]	79	142
Ståhl and Bosch [2014]	49	110
Fabry et al. [2015]	36	109

2001] and 2003 [Xing and Stroulia 2003], although both of these were superseded. The first articles from our primary set are from 2004 [Ebraert et al. 2004; Ren et al. 2004]. Ebraert et al. [2004] first introduced the idea of recording development activities in the IDE, whereas Ren et al. [2004] was the first to introduce a way of calculating the difference (in terms of changes) between two versions of a system. After that, the interest in this topic steadily increased to 13 articles in 2007 and 2008 with an average of 8 articles per year from that point onwards. Note that, for 2015, there are only 3 articles; this can be explained in two ways: First, as the data extraction for this article was performed in June 2015, there was still half a year in which articles could be published. Moreover, it often takes months for a published conference or workshop article to actually appear in digital libraries, further reducing the amount of discoverable articles as of June 2015. Second, the most recent articles are most likely not yet cited in any other articles, and therefore they cannot be included in our set during the snowballing step of our search process; they have to be immediately found by the search query or otherwise be missing from this study entirely.

We also counted the number of authors of the articles, as an indicator of the size of the community: In our primary studies, we found 111 unique authors. This number shows that there is a significantly large community of people interested in this topic. Furthermore, in order to provide a frame of reference, we also provide the same metrics (number of articles and number of authors) for three literature studies in adjacent topics, as shown in Table II.

The survey by Kagdi et al. [2007] is the closest work to ours, as it is a survey of the early field of MSR—as we will see later, a significant portion of the work concerning first-class changes happens in the context of MSR. The survey, published in the *Journal of Software Maintenance and Evolution* in 2007, identified 79 articles as relevant, with 142 authors. We also include two recent studies. The survey by Ståhl and Bosch [2014] concerns the topic of continuous integration and the industry practices there. It was published in the *Journal of Systems and Software* in 2014 and features 49 articles from 110 authors. Finally, the survey by Fabry et al. [2015] was recently published (in 2015) in *ACM Computing Surveys* and concerns the topic of Domain-Specific Aspect Languages. This work surveyed 36 articles with 109 authors.

We can see that these four works are in the same range in terms of both number of studies and number of unique authors, with the Kagdi study being relatively larger in terms of authors. We note that our study tends to include more articles, for a similar number of authors, and assume this was due to the presence of superseded studies. Removing those yields the same amount of unique authors but brings the number of articles down to 65, putting this study further inline with the others.

Finally, we wish to mention that the Kagdi study mentions the need for MSR approaches on finer-grained entities as one of the open issues of the field at the time. Kagdi et al. state that "[They] feel that a finer-grained understanding of the source-code changes is needed to address these types of questions." In that view, we can see the work on change reification addresses that problem in a direct fashion. Kagdi et al. even

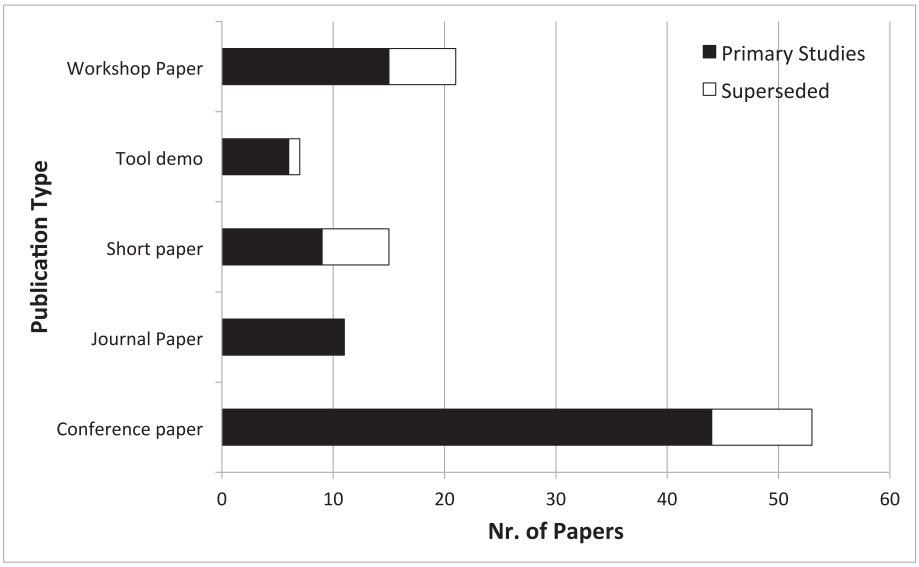

Fig. 5. Number of articles per publication types.

mention one of our primary studies—Fluri's work [Fluri et al. 2005]—as a work in that direction.

We can conclude from these numbers that this is a topic that has piqued and sustained the interest of the software engineering research community.

4.2. Venues of Publication

In this study, we only considered peer-reviewed venues (including journals as well as peer-reviewed conferences and workshops). The articles published at workshops or conferences also include tool demonstration articles and short articles (maximum of four pages). Figure 5 shows the distribution of the articles (including superseded articles) according to its type of publication. We can see that the most popular venues for publishing in this topic have been conferences with few workshop articles and even fewer journal articles. This is a common trend for articles in software engineering or, more generally, in computer science [Meyer et al. 2009]. When looking at the actual venue where the articles in our primary set have been published (see Figure 6), we can clearly identify the top most popular venues. The two most popular venues (with 14 articles each) are the ICSE and the International Conference on Software Maintenance and Evolution (ICSME) (including all the articles published under the conference's previous name: International Conference on Software Maintenance). The next most popular venue is the International Conference on Software Analysis, Evolution, and Reengineering (SANER), but we have to note that this venue includes all articles published at both the European Conference on Software Maintenance and Evolution and the Working Conference on Reverse Engineering, as these two conferences merged as SANER. The next conference, with 5 articles, is the Working Conference on MSR with another 3 articles at MSR from when it was still a workshop (making 8 in total).

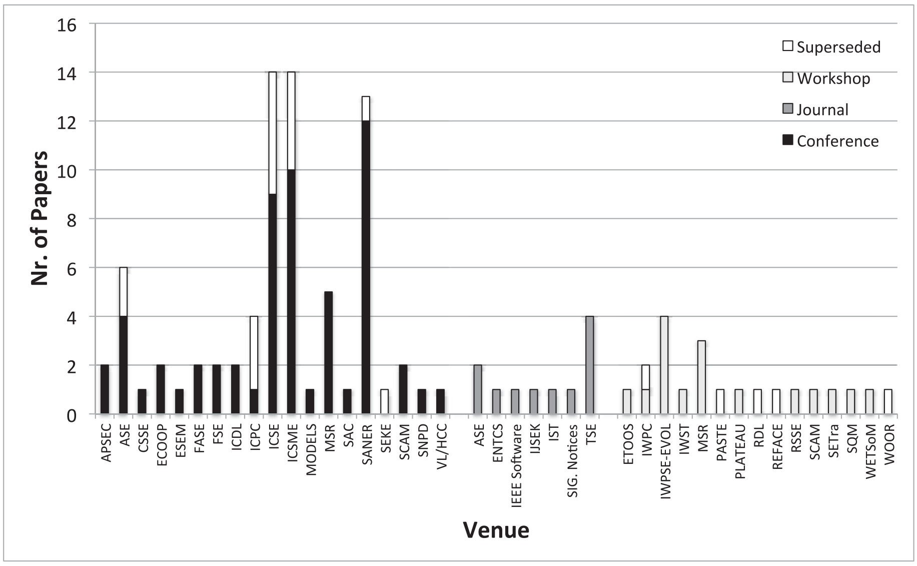

Fig. 6. Number of articles per venue.

We note that an important proportion of the works were published in strong, highly regarded software engineering venues, such as ICSE, TSE or the international conference on Automated Software Engineering (ASE). This gives evidence that work on change reification has been perceived as relevant, high-quality work by the reviewers for these venues.

As can be expected, most of the articles regarding change reification have appeared at maintenance-/evolution-oriented conferences, namely ICSME, SANER, and, to a lesser extent, MSR. However, the theme also appeared in the top[3] *software engineering venues (ICSE, ASE, TSE).*

5. CHANGE MODELS USED

In this section, we deal with **RQ2** (*What are the commonalities and differences in the change models adopted by the primary studies? Can we extract a unified model for changes?*). Each approach that uses reified changes has, at its core, a representation for changes. It is therefore important to look at the different representations and investigate their strong points and limitations.

We classified the tools or approaches presented in the primary studies according to two discrete dimensions: the granularity of the changes (either fine-grained changes or composite changes) and the way of obtaining these changes (either through recording them from the IDE or by recovering them from the difference between two versions). The composite changes in practise boil down to the recording and recovering of refactorings. As such, we identified four categories of studies using change reification: Recording of Fine-Grained Changes, Recording of Refactorings, Change Distilling, and Refactoring Reconstruction. The tools and approaches presented in the primary studies

[3]According to the CORE rankings; http://portal.core.edu.au/conf-ranks/.

Table III. Classification of Tools and Approaches

	Atomic Changes (Fine Grained)	Composite Changes (Refactorings)
Recording Changes (Logging)	ChangeBoxes ChEOPS ChEOPSJ CodingTracker Epicea Fluorite GumTree MolhadoRef OperationRecorder (&Replayer) Spyware Syde	CatchUp! ChangeBoxes CodingSpectator MolhadoRef Spyware
Recovering Changes (Distilling)	Celadon ChangeDistiller ChEOPSJ Chianti Diff/TS FaultTracer LSDiff UMLDiff	ChEOPSJ Diff/TS LibReDe RefactoringCrawler RefFinder UMLDiff

are then classified into these categories (see Table III). Note that a tool can be in more than one category. For instance, MolhadoRef [Dig et al. 2008] is capable of recording both source-code changes and refactoring operations. ChEOPSJ [Soetens and Demeyer 2012] is another example, as it is capable of both recovering changes from a Software Configuration Management (SCM) system, as well as record them from the IDE, and then use the recorded changes to reconstruct refactoring operations.

In the next sections, we will have a closer look at the models used for recording or recovering fine-grained changes (in Section 5.1) and the refactorings that are supported by tools that record or recover those (in Section 5.2).

5.1. Atomic Changes

We first looked at all articles that detail the model they use for recording or recovering fine-grained changes. For each article that defines such a model, we looked at several things:

—What are the basic change operations defined?
—What kind of source-code entities do the changes act on?
—Which programming languages are supported?
—How do they deal with modifying or updating an existing source-code entity?

We present an overview of some basic details in Tables IV and V that answer some of these questions for all of the approaches in the left column (Atomic Changes) of Table III.

Representation of Source Code. Table IV summarises which model is used to represent source-code entities for all tools investigated.

One third (6 of 18) of these approaches use the Abstract Syntax Tree (AST) as a way to represent source-code changes. Changes are then modelled as operations that act on the AST nodes [Negara et al. 2012; Falleri et al. 2014; Robbes and Lanza 2008c; Hattori and Lanza 2010; Fluri et al. 2007b; Hashimoto and Mori 2008].

Another third (6 of 18) of these approaches use an ad hoc representation for source code, meaning that they do not use a formal model for representing source-code entities; rather, they only represent a certain number of (high level) source-code entities. For instance, FaulTracer only supports changes on fields and methods [Zhang et al.

Table IV. Tools Supporting Atomic Source Code Changes

Approach	Source-Code Representation		
	Language Supported	Model Used	Entities Supported
ChangeBoxes	Smalltalk	Ad hoc	Class, Field & Method
ChEOPS	Smalltalk	FAMIX	any FAMIX Entity
ChEOPSJ	Java	FAMIX	any FAMIX Entity
CodingTracker	Java	AST	any AST node
Epicea	Smalltalk	Ad hoc	Package, Class, Field & Method
Fluorite	Java	Source Code Text	
GumTree	Java	AST	any AST node
MolhadoRef	Java	Ad hoc	Package, Class, Field & Method
OperationRecorder	Java	Source Code Text	
Spyware	Smalltalk	AST	any AST node
Syde	Java	AST	any AST node
Celadon	AspectJ	Ad hoc	Aspects
ChangeDistiller	Java	AST	any AST node
Chianti	Java	Ad hoc	Class, Field, & Method
Diff/TS	Python, C, C++, Java	AST	any AST node
FaultTracer	Java	Ad hoc (Chianti-Like)	Method, Field
LSDiff	Java	Tyruba Logic Facts	Package, Class, Field & Method
UMLDiff	Java	UML	Package, Class, Field & Method

Table V. Tools Supporting Atomic Source Code Changes

Approach	Type of Operations		
	Add & Delete	Property Update	Composite Changes
ChangeBoxes	✓		Rename
ChEOPS	✓		Modify = Delete + Add
ChEOPSJ	✓		
CodingTracker	✓	✓	
Epicea	✓	✓	
Fluorite	✓		
GumTree	✓	✓	Move
MolhadoRef	✓	code edits	
OperationRecorder	✓		
Spyware	✓	✓	
Syde	✓	✓	
Celadon	✓		
ChangeDistiller	✓	✓	Move
Chianti	✓		
Diff/TS	✓	✓(Relabel)	Move
FaultTracer	✓		
LSDiff	✓		
UMLDiff	✓		

2011], whereas ChangeBoxes and Chianti also support changes on classes [Denker et al. 2007; Ren et al. 2004]. Epicea and MolhadoRef additionally support changes on packages [Dias et al. 2013; Dig et al. 2008]. Similarly, Logical Structural Diff (LSDiff) and UMLDiff model changes on packages, classes, fields, and methods, yet for UMLDiff these are modelled using the UML meta-model and in LSDiff they are stored as facts in a factbase [Xing and Stroulia 2007b; Kim and Notkin 2009]. Celadon is tool that extends the change model of Chianti to also support the aspect-oriented programming language AspectJ [Zhang et al. 2008b]. To that extent, they support changes on entities like aspects, advice, or pointcuts.

ChEOPS and its Java brother, ChEOPSJ, used the FAMOOS Information Exchange Model (FAMIX) to represent source-code entities, as this is not a language-specific model and can thus be used for any object-oriented programming language [Ebraert et al. 2007a; Soetens and Demeyer 2012].

It is feasible that all of the applications for which the above tools have been used can be translated to a tool that models the source code using the more fine-grained AST representation—with maybe the exception of Celadon, which works on aspects.

There are two more tools that do not model changes that act on the AST: Fluorite [Yoon and Myers 2011] and OperationRecorder [Omori and Maruyama 2009]. In fact, these tools record changes at an even finer levels of granularity. They record textual changes made to the source code itself. From these textual changes, they can later infer AST changes [Kitsu et al. 2013].

Type of Operations. All approaches agree on using add (or insert, create, or introduce) and delete (or remove) as basic change operations. Spyware has two types of additions and two types of deletions to take into account whether the location of the source-code entity is relevant (i.e., just add or add at location x) [Robbes and Lanza 2008c].

Dealing with modifications or updates of existing source-code entities is more tricky. Some approaches have only modelled the source code down to the level of methods, in which case changes in the method body are either ignored [Denker et al. 2007; Xing and Stroulia 2007b] or considered as a single "method body change" [Ren et al. 2004; Kim and Notkin 2009]. In ChEOPS, a method body change is considered a composite change, consisting of a remove method and an add method [Ebraert et al. 2007a].

Most other approaches agree on representing modifications as a *"property update"* [Negara et al. 2012; Dias et al. 2013; Falleri et al. 2014; Hattori and Lanza 2010; Fluri et al. 2007b]. The first approach to do so was Robbes and Lanza's Spyware [Robbes and Lanza 2008c]. To achieve this, a property update needs to contain (at least) the three following values: (i) the identifier of the property being updated, (ii) the old value of the property, and (iii) the new value of the property.

Additionally, some approaches also explicitly model the dependencies between changes (i.e., a change can only be applied if the changes it depends on have been applied first) [Ren et al. 2004; Ebraert et al. 2007a; Soetens and Demeyer 2012]. As an example of this, one can only add a method to a class if the class itself was added before. However, one could argue that this dependency information is already represented in the source-code model itself (e.g., a method `belongsTo` a class).

Unified Model of Fine-Grained Changes. Overall, from all these approaches, we can derive a common model for first-class changes: We define a *Subject* as an object representing a source-code entity (or a relationship between source-code entities). This representation can be done in several ways: Either we directly use the AST-nodes or we use a more-abstract and higher-level source-code representation, like FAMIX or UML.

We can then define three "atomic" change operations that act on these subjects:

—*Add* is an object representing the creation and addition of a *Subject*.
—*Delete* is an object representing the removal and deletion of a *Subject*.
—*Update* is an object that represents a change in one of the properties of a *Subject*.

One can then devise means of modelling composite changes (like refactorings) as a combination of any of these three atomic changes. In order to support the applications investigated with Chianti, ChEOPS, and ChEOPSJ, our common model also supports

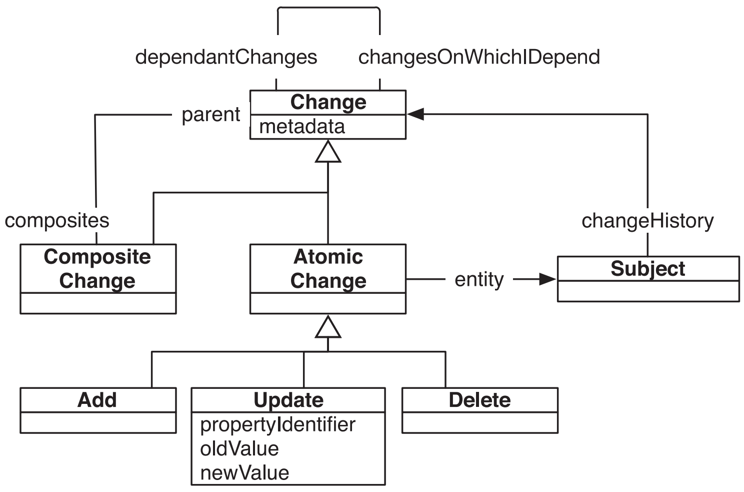

Fig. 7. Core model for first-class changes.

dependencies between changes. As such, our common model for first-class source-code changes is shown in Figure 7.

A change can be either a Composite Change *or an* Atomic Change*. An* Atomic Change *acts directly on a source-code entity (a* Subject*) and can be one of three different changes (*Add*,* Delete*, or* Update*). A* Composite Change *consists of many different other changes, which in turn can be* Composite *or* Atomic*.*

5.2. Composite Changes

In this section, we summarise the tools and approaches that deal with the recording or recovering of composite changes. As it stands, most approaches only deal with refactorings as composite changes. Therefore, in this section, we will be discussing the recovering and reconstruction of refactorings rather than composite changes.

Several approaches that we mentioned in Section 5.1 have built-in support for smaller refactorings like MOVE or RENAME [Fluri et al. 2007b; Dig et al. 2008; Robbes and Lanza 2008c; Hattori and Lanza 2010; Negara et al. 2012; Falleri et al. 2014]. This means that they are represented in their own first-class change operation. A RENAME, for instance, can be considered as either a composition of a delete change and an add change or as an update in which the source-code identifier is a property being updated (though it depends on the source-code model used and on which the entity is being updated). In the FAMIX model, for instance, the method identifier is a property of a method object, but if the source code is represented using the Java AST from the Eclipse JDT, any identifier is in fact its own AST node (`SimpleName`), so in that case the update will not act on the `MethodDeclaration` but on the `SimpleName`. A MOVE can also be considered as a composition of a delete and an add.

As we can see in Table III, there are also several approaches specifically targeted at the recording or reconstruction of refactoring operations. We start by listing those approaches that record refactoring operations that are performed in an IDE.

CatchUp! was the first tool that proposed recording refactoring operations straight from the IDE [Henkel and Diwan 2005]. This tool prototype was implemented as an Eclipse plugin. The basic idea was to allow library developers more flexibility in *software libraries' evolving application programming interfaces (API's)*, and client developers would more easily port their application to a newer version of the library by simply replaying the recorded refactorings.

This idea was then later adopted by the Eclipse team and has found its way into the main release and is now standardly available in the Eclipse IDE [Dig et al. 2007, 2008]. Eclipse allows a developer to create a "Refactoring Script" that keeps track of the recorded refactorings that can later be replayed on dependent projects. This refactoring script contains refactoring descriptor objects formatted in the Extensible Markup Language (XML).

CodingSpectator uses these refactoring descriptors and enhances it with other recorded data (timestamp of refactoring, any problems reported, the time spent using the wizards, etc.) [Vakilian et al. 2012].

MolhadoRef only mentions the theoretical possibility of its refactoring recording capabilities [Dig et al. 2008]. More specifically this approach is also implemented as an Eclipse plugin and as such also has access to the log of refactorings performed. One of the things they mention is that even if such a log of refactorings were not present, one could reconstruct refactorings using one of the other approaches.

In order to recover refactorings, we summarise five approaches as follows.

Both LibReDe and RefactoringCrawler are very similar approaches, in that they both apply a signature-based analysis [Weißgerber and Diehl 2006b; Dig et al. 2006]. LibReDe starts by looking for refactoring candidates based on added, changed, or removed entities using a signature-based analysis [Weißgerber and Diehl 2006b]. It then uses a similarity metric as a means to filter out bad candidates. In contrast, RefactoringCrawler uses a combination of syntactical and semantical analysis [Dig et al. 2006]. The difference lies in the order in which the signature-based analysis and the similarity measure is done. Further, RefactoringCrawler uses shingles, a form of hashing, as a similarity metric. RefactoringCrawler first identifies refactoring candidates using the Shingles similarity metric and then uses the signature-based analysis to filter out the bad candidates.

RefFinder, an Eclipse plugin by Kim et al., is to date the most comprehensive refactoring reconstruction tool as it supports 63 different types of refactorings [Kim et al. 2010]. They use the technique proposed by Prete et al., which is stronger than all previous techniques because they not only detect primitive refactorings (which all previous techniques do to some extent) but also "complex refactorings" (i.e., refactorings that are combinations of primitive refactorings) [Prete et al. 2010]. To do this they rely on a fact base with a strong query engine (Tyruba logic). They describe the structural constraints before and after applying a refactoring in terms of template logic queries. RefFinder takes two versions of a system as input from the Eclipse workspace and recovers changes as logic facts about the systems' syntactic structure using LSDiff. These are then stored in a factbase, which can be queried to identify program differences that match the constraints of each refactoring type under focus.

Xing and Stroulia have developed an algorithm that compares two subsequent versions of a system at the design level [Xing and Stroulia 2007b]. Their UMLDiff algorithm is capable of detecting some basic structural changes in the system, such as the addition, removal, renaming, or moving of UML entities. More complex structural changes can be found using a suit of queries that try to find a composition of elementary changes.

ChEOPSJ is a tool that records fine-grained change operations and dependencies between these changes from the Eclipse IDE [Soetens et al. 2012]. Changes are as such represented as a graph with changes as nodes and dependencies between changes as edges. It is then capable of mining the change graph (representing the evolution of a software system) for fixed patterns of changes representing refactorings. Soetens et al. demonstrated that their use of recorded changes improves on approaches that use recovered changes, since in this way they do not suffer from the fact that refactorings become obfuscated by other changes in the same commit [Soetens et al. 2013b].

Diff/TS has a similar approach, in that it first distills fine-grained changes between two versions of a system, using a tree-differencing algorithm [Hashimoto and Mori 2008]. These fine-grained changes can subsequently be mined for source-code change patterns, including some simple refactorings like LOCAL VARIABLE EXTRACTION or LOCAL VARIABLE RENAME [Hashimoto et al. 2015]. Interestingly, this is the only approach that also mentions the reconstruction of other composite changes by mining the change history for bug fixing patterns and by mining the Linux kernel source code for the "BKL (Big Kernel Lock) pushdown."

Most approaches to date only consider refactorings as composite changes. The approach of recording refactorings was even so successful that it was incorporated in the standard release of the Eclipse IDE. Of the approaches that reconstruct refactorings RefFinder is to date the most comprehensive tool, yet this approach could still be improved were it to use recorded changes instead of recovered ones.

6. OVERVIEW OF APPLICATIONS

In order to answer **RQ3** (*What are the applications in which reified source-code changes are used in the primary studies?*), we list and briefly explain the applications for which the primary studies used change reification. Figure 8 shows an overview, in which, for each identified application, the number of articles dealing with that application is counted. We used a card-sorting strategy to classify the applications into one of three categories: *Software Development*, *Reverse Engineering*, and *Software Evolution*. Each of these categories has achieved a similar number (±30) of studies. Note that an article can sometimes mention more than one application for which they use reified changes.

6.1. Applications in Software Development

Applications in software development often boil down to recommendations systems to aid developers in their development tasks. Even in the 1980s the *Programmer's Apprentice project* strived to provide each developer with an "apprentice" to assist them with the mundane tasks of software development [Rich and Waters 1990]. The apprentice would be an intelligent computer program that actively participates in the software engineering process and acts as a recommendation system to the programmer. Since then, many recommendation systems for software developers have been created [Zeller 2007; Robillard et al. 2010]. Most of the existing approaches base their suggestions on data obtained from source code, software repositories, bug reports, or mailing lists. Recorded changes offer unique opportunities here, as the data that are recorded are much more fine grained than the one recovered from source-code repositories. In particular, the change data can be exploited as soon as the developer processes it, instead of waiting for a developer to commit to a software repository.

In the category of software development, we found that, in our primary studies, the most popular application that uses reified changes is *change impact analysis* with a particular emphasis on *predicting or suggesting future changes* and *test selection*.

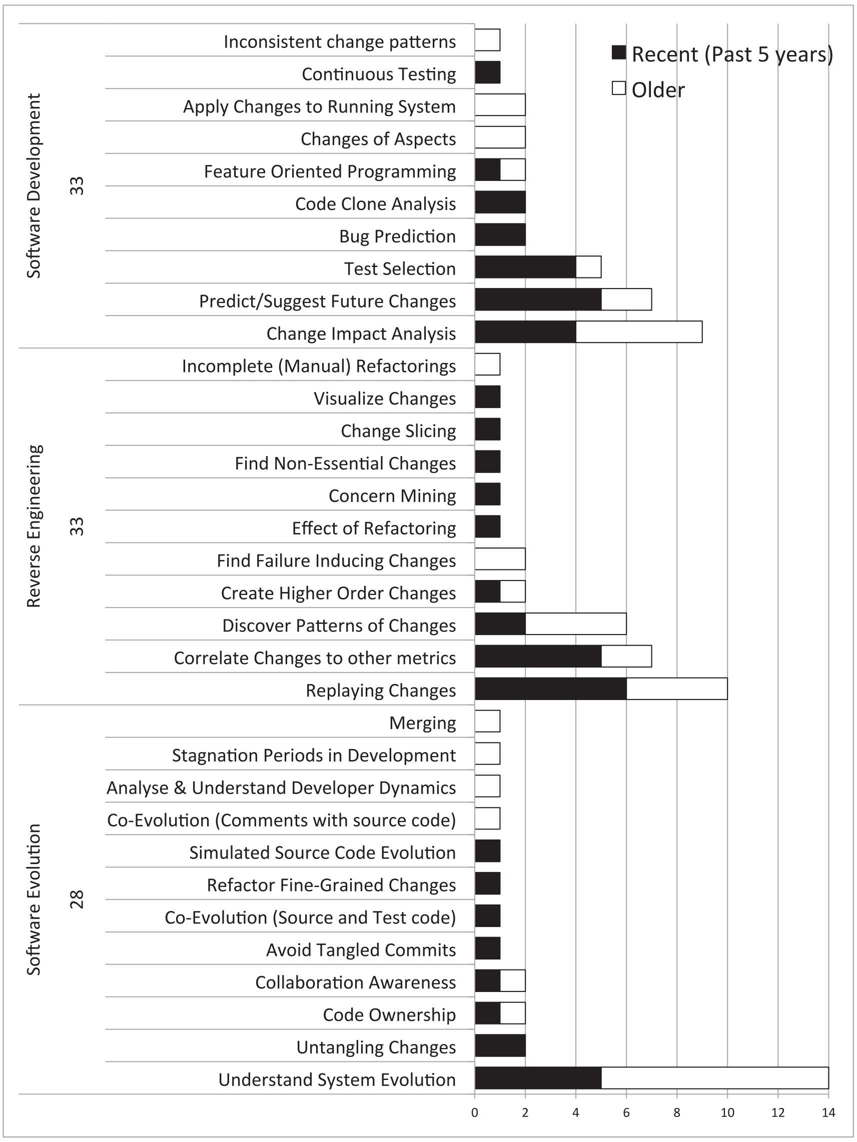

Fig. 8. Applications mentioned in the primary studies.

Change Impact Analysis. One of the most obvious applications of modelling changes as explicit entities is change impact analysis—that is, to identify the potential consequences of a change or to estimate what needs to be changed for a particular task. Files that have in the past changed together are likely to be related. Thus, when in the future we change one file, we can expect that the other must also be changed. This is what is called change coupling or logical coupling. Fluri et al. and Robbes et al. have investigated the notion of change coupling in terms of fine-grained source-code changes [Fluri et al. 2005; Robbes et al. 2008]. In doing so, they demonstrated that a significant amount of change couplings are in fact not caused by source-code changes [Fluri et al. 2005]. The study by Robbes et al. found that fine-grained change coupling was a good early predictor of later coarse-grained change coupling, showing that fine-grained change data is a more accurate metric.

The next two applications use change impact analysis in their foundation. Predicting changes, on the one hand, uses historic information to estimate what needs to be changed in the future. Test selection, on the other hand, tries to find sets of tests that are impacted by selected changes.

Predicting or Suggesting Future Changes. Code completion is one of the most used recommenders in practice [Murphy et al. 2006], yet it can be largely improved. The change data recorded by Spyware has been used to design and evaluate a variety of code completion algorithms [Robbes and Lanza 2010] that go beyond the default alphabetical sorting that is used by the state-of-the-practice IDEs. Algorithms based on recent change information were found to be more than 5 times as accurate as the default algorithm in Pharo, an open-source Smalltalk IDE. Later, a tool that implemented one of the algorithms was released for Pharo and was approved by the developer community [Robbes and Lanza 2010]; it was subsequently adopted as the default completion tool in Pharo and is now in daily use.

In the same vein as code completion, several change prediction algorithms, aimed at easing navigation towards the code entity that a developer wishes to change, were evaluated [Robbes et al. 2010]. Surprisingly, of the extensive array of algorithms tested, the simplest one (recommending recent changes) was found to be among the most effective. Likewise, a simple recommender tool for the Pharo IDE was developed.

Fluri et al. developed a tool called ChangeCommander, which suggests changes when a developer introduces a new method invocation into the system based on patterns of changes obtained by analysing the history stored in a software repository [Fluri et al. 2008b].

Finally, Romano and Pinzger, as well as Giger et al., used correlations of changes to other metrics as a way to rank source-code files according to their proneness to be changed [Romano and Pinzger 2011; Giger et al. 2012b].

Test Selection. Chianti is another approach to change prediction in the context of impact analysis. It works at the granularity of two versions, using change recovery and static analysis to pinpoint the changes that caused a behavioural change in a test suite (inferred via dynamic analysis of the versions) in the potentially large sequence of edits between the versions [Ren et al. 2004]. Chesley et al. used the data obtained from Chianti as a means to identify *failure-inducing changes* [Chesley et al. 2005]. When after a long editing session a test unexpectedly fails, it is often difficult to identify which change(s) are the cause of the failing test. The approach by Chesley et al. allows a programmer to select those changes that affect the failing test and apply these changes to the original (unmodified) system. This allows a programmer to focus only on those changes that affect the failing test. In this way, a programmer can iteratively select, apply, and undo individual (or sets of) changes in order to find the failure-inducing change(s). Stoerzer et al. have extended this approach by classifying the changes a programmer can select as *Red* (high likelihood of being failure-inducing changes), *Yellow* (possibly problematic changes), or *Green* (changes correlated with successful tests) [Stoerzer et al. 2006].

In most software systems, the tests grow to become a considerable part of the source code: Tests can in some cases make up to 50% of a system's source code. Running all tests whenever a software developer adapts a small part of the code can introduce an unacceptable overhead. By analysing the dependencies between the change objects of the program code and those of the test code, one can discover what the relevant tests are for each change. These tests can be suggested to the developer so he or she does not have to run all the tests [Soetens and Demeyer 2012; Soetens et al. 2013a].

Other applications. It has long been believed that files or source-code modules that are changed often are more likely to contain bugs. Giger et al. have used fine-grained source-code changes and correlated metrics thereof with the number of bugs in source-code files. In doing so, they were capable of training a prediction model to indicate bug-prone parts of a system [Giger et al. 2011, 2012a]. Giger et al. showed that their approach of using fine-grained source-code changes outperforms previous approaches (e.g., Nagappan and Ball [2005]) that leverage code churn.

Ebraert proposed to model features as sets of changes that have to be applied to a software system in order to add the corresponding functionality to the system. Concretely, this approach enables bottom-up feature-oriented programming and consists of three phases. First, the change operations are captured as first-class entities. Second, these operations are classified into separate sets that each implement one functionality. Finally, those modules are recomposed in order to generate software variations that provide different functionalities. The fine-grained information of the change objects can also contribute to the verification of the validity of compositions and in revealing composition conflicts [Ebraert 2008; Ebraert et al. 2011].

Most systems undergo changes even after the actual development phase, when they are in production. While the addition of new features can usually wait until the next release of a system, changes like bug fixes may need to be integrated into the running system as soon as possible. However, the modification of a running system is barely possible. Müller and Villegas state that in order to achieve this, one needs to outfit highly dynamic software systems with self-adaptation mechanisms. Thus the system itself is capable of monitoring selected requirements and environment conditions to assess whether it needs to change [Müller and Villegas 2014].

Denker et al. proposed Changeboxes as a general-purpose solution for encapsulating change as first-class entities in a running software system [Denker et al. 2007]. ChangeBoxes supports first-class changes and allows one to apply them to a running system to support runtime evolution. Further, ChangeBoxes supports change scoping, that is, applying changes only in part of a system, allowing several versions of the system to coexist at the same time. This allows one, for instance, to gradually deploy changes in a running application so they affect initially only selected users and, if successful, to deploy them more broadly to reach all users.

Zhang et al. have developed Celadon, a tool that infers fine-grained changes of a system that is built using aspect-oriented programming [Zhang et al. 2008b]. They then used this tool for the purpose of change impact analysis in AspectJ code. Qian et al. used this same tool to mine for frequently occurring change patterns in AspectJ code [Qian et al. 2008].

6.2. Applications in Reverse Engineering

Reverse engineering is the process of analysing a software system and extracting a higher level view of its design. The goal is to improve understanding of how a system works and how it is structured. One way of achieving this is by studying how a system evolved or by analysing patterns or metrics of a system. Thus, in reverse engineering, the top three applications are to *replay changes*, *correlate changes to other metrics*, and *discover patterns of changes*.

Replaying Changes. This can be interpreted in two ways: (1) recording changes made to a particular software system and then replaying them in order to better understand the evolution of that system or (2) recording changes in order to replay them on other (related) systems. Replaying changes on a single system to better understand the system's evolution was done by Hattori et al. [2010, 2013], as well as by Robbes et al. [2010], Robbes and Lanza [2010], and Omori and Maruyama [2009, 2011].

Maruyama et al. argued that replaying all changes in chronological order is not always necessary [Maruyama et al. 2012]. When a developer wants to investigate how a particular entity in the system evolved, he or she often need to skip several unrelated changes. To this end, Maruyama et al. introduced the concept of change slicing. A change slice contains all changes related to a particular source-code entity. Hence, when investigating the evolution of that entity, a developer only has to replay the changes in the slice, thus reducing program comprehension effort.

The second interpretation of this applications (re-executing on other systems) has to date only been done in the context of API evolution. Dig et al., as well as Henkel and Diwan, developed ways to record refactoring operations applied on a system in order to replay these refactorings on other dependant systems [Dig et al. 2008; Henkel and Diwan 2005].

Correlate Changes to Other Metrics. As mentioned before, Giger et al. have correlated changes to the bug proneness of source code and used this to create prediction models that can be used to predict which source-code modules will more likely contain bugs [Giger et al. 2011, 2012a]. Moreover, they also correlated changes to traditional object-oriented metrics and to measures from social network analysis [Giger et al. 2012b], thus creating prediction models of which source-code files are more prone to be changed.

Romano et al. did a similar study, using anti-patterns. Given the notion that classes affected by anti-patterns are more likely to be changed, they provided deeper insight into which anti-patterns lead to which types of changes [Romano et al. 2012].

Stoerzer et al. correlated changes to test failures, thus creating models that classify changes into the likelihood that they are failure inducing [Stoerzer et al. 2006].

Weißgerber and Diehl correlated reconstructed refactorings to bug reports, thus analysing whether refactorings are more likely to introduce failures than non-refactoring changes [Weissgerber and Diehl 2006a].

Discover Patterns of Changes. Given the fine-grained notion of reified changes, several researchers set out to find as-yet-unknown patterns of changes [Fluri et al. 2008a, 2008b; Hashimoto et al. 2015; Negara et al. 2014; Qian et al. 2008; Xing and Stroulia 2005a]. Fluri et al. and Negara et al. succeeded in identifying several patterns [Fluri et al. 2008a; Negara et al. 2014]. Fluri et al. discovered that such patterns often capture the semantics of a particular development activity. For instance, the patterns Swap Control Flow Order or Merging Control Flow affect the control flow inside a method, whereas Remove Superfluous Parameter affects the API [Fluri et al. 2008a]. Negara et al. identified 10 frequently occurring patterns of changes, which they then presented to 420 participants in a survey [Negara et al. 2014]. Eight of their patterns were deemed relevant by more than half of the participants. Some of the patterns they found include Add Precondition Check for a Parameter, Wrap Code With Timer, or Copy Field Initializer.

Other applications. Reifying changes that were recorded from the programmer's activity in the IDE allows one to manipulate them further. Robbes and Lanza defined a semi-automatic, tool-supported process to convert a concrete sequence of recorded changes to a generic higher-order change [Robbes and Lanza 2008a]. The higher-order change accepts parameters (e.g., classes, methods, source-code selections) and acts as a change generator whose output is determined by these very parameters. Executing the generated changes on the system's AST effectively applies the higher-order change to the system. A further benefit is the possibility of including metadata that "tags" the changes as having been issued from the higher-order change. This allows a full integration of the higher-order changes in the evolution, allowing them to be tracked over time and possibly reworked when the higher-order change evolves.

Hayashi et al. proposed a method of rewriting (or restructuring) the recorded edit history. An added benefit is that this allows us, for instance, to compose large refactorings from primitive refactorings or to define bad smells in the change sequence. Such bad smells could then automatically be detected before committing to the repository [Hayashi et al. 2012].

Kawrykow and Robillard argue that techniques that recover changes from software repositories might be influenced by "non-essential changes," such as local variable refactorings or textual differences induced as part of a rename refactoring [Kawrykow and Robillard 2011]. They developed a way of identifying such non-essential changes in a revision of a software system, thus gaining insights into the distribution of non-essential changes in a system's evolution.

6.3. Applications in Software Evolution

In xoftware evolution, most work has been done in order to better *understand system evolution*, which can also be considered part of reverse engineering. Apart from that, the software evolution applications are centred around improved versioning systems, as well as increased awareness of multiple developers working on the same system.

Understand System Evolution. At a lower level, reviewing the sequence of changes recorded during the implementation of a functionality can assist the program comprehension effort. Since the actual sequence of changes is recorded, one can review the changes in order. Thus replaying changes (as mentioned in Section 6.1) is an important tool to better understand how a system evolved.

Hattori et al. take the comprehension of development sessions to a collaborative perspective with Syde by allowing developers to explore and watch past development sessions of any team member [Hattori et al. 2010]. Hence, a developer can also be assisted in understanding the past work of others. The possibility to replay past changes of any team member has proven to be efficient and effective to answer common questions raised by developers related to the evolution of a system.

The approach by Omori and Maruyama also supports operation replay and allows the definition of fine-grained visualisation of the changes in order to identify specific patterns of activity, such as programmer inactivity, commenting out source code, debugging activities, and so on [Omori and Maruyama 2008, 2011].

Versioning. A system's reified changes represent its entire evolution. This allows for more accurate software evolution analysis. Traditionally, such analysis is based on the limited amount of information contained in a software repository (such as SVN, git or the Concurrent Versions System (CVS). Such repositories are line-based tools that suffer from two essential problems: (1) The textual edits contain no semantic or even syntactic meaning, whereas changes explicitly operate on AST entities, and (2) they do not contain every intermediate version of a system but only snapshots taken when a developer committed source code into the repository [Robbes and Lanza 2005]. This leads to another problem when several developers are working together on the same project. Merge conflicts can occur when two developers commit changes to the same part of a system.

Change reification and the relations between changes can provide better conflict detection and better support for resolving these conflicts. Indeed, one of the first approaches to achieve this was taken by Lippe and van Oosterom, who introduced the concept of Operation-based Merging [Lippe and van Oosterom 1992]. Their technique also lies at the core of MolhadoRef, the refactoring-aware source-code management system by Dig et al. [2008]. The issue addressed by this work is that refactorings (such as consistently renaming an entity) result in large changes in the code base: One renaming refactoring changes all the files that reference the renamed entity. This can break the source code or cause conflicts if someone changes one of these files concurrently. MolhadoRef treats refactorings as change operations that are recorded and replayed when a merge is necessary, effectively removing conflicts due to refactorings. Furthermore, modelling recording refactorings as logical operations instead of large physical changes helps one to better understand a system's evolution.

Another problem with source-code repositories is that when developers commit their changes, they often have changed several things. While in the process of fixing a bug, a developer might have performed a few refactorings and fixed a typographical error they found in one of the comments. When all of these changes are committed to the repository in a single commit, we speak of tangled commits. Dias et al. and Hayashi et al. have proposed techniques to avoid this problem by untangling a set of changes before committing to the repository and then committing the changes for the different activities in their own separate commit [Dias et al. 2015; Hayashi and Saeki 2010; Hayashi et al. 2012]. Hayashi et al. take it one step further and give the reins back to the developer, who can then "refactor" their change history; by merging or splitting changes and reordering them, a developer is given control over which changes are part of which task [Hayashi et al. 2015].

Collaboration and Awareness. Awareness is defined as the understanding of the changes of others that can impact one's work. Recent work has investigated how to break the workspace isolation introduced by SCM systems by providing developers with real-time notifications of the activity of the team (e.g., who is changing which parts of the system) [Sarma et al. 2008]. Most awareness approaches treat the system as a collection of files; instead, Syde uses change reification to record every change made by every developer within a team and provides them with real-time information [Hattori and Lanza 2009b, 2010]. With this information, Syde enriches the Eclipse IDE with visual cues that show which parts of the system are under change or have recently changed and by whom.

One kind of information that is particularly useful for developers is to know when they are performing concurrent modifications and thus when and where potential merge conflicts emerge. Hattori and Lanza have proposed an operation-based conflict detection algorithm that notifies developers in real time when merge conflicts might be emerging, thus preempting the conflict detection of SCM systems [Hattori and Lanza 2010].

When multiple developers are working on the same system, there is also the notion of code ownership, that is, which developer is most responsible for which part of the code. When changes are recorded from the IDE, we can generate a much more fine-grained image of code ownership [Hattori and Lanza 2009b, 2010].

Other applications. Fluri et al. have investigated the co-evolution between source code and comments [Fluri et al. 2007a]. They found that most of the comment changes are done at the same time as the associated source-code change. However, they rarely co-evolve, newly introduced code is barely commented, and, when it is, it is mostly high level to document the intent of a class or method declaration.

Marsavina et al. looked into the fine-grained co-evolution of production code and test code [Marsavina et al. 2014]. Thus they were able to identify patterns of co-evolution and thus provide a better understanding of how test code evolves.

We find that many applications that use reified changes have been studied. Most notably, recorded changes of a system could be replayed to increase the understanding of how the system evolved. Moreover, a given set of changes can be mined for frequent patterns that can in turn be used to predict or suggest future changes. Other areas where reified changes have proved useful include regression test selection, improved version control approaches, and awareness of other developers in multi-developer projects. Nevertheless, it should be pointed out that in the latter areas reified changes are not a necessary precondition but rather an added value. Indeed, the field of mining software repositories (cf. the article by Kagdi et al. [2007]) also studies these application areas, often without having an explicit representation of the changes themselves. It remains to be seen whether the added value of an explicit change model outweighs the cost.

7. ROADMAP FOR FUTURE RESEARCH

In order to answer **RQ4** (*What is the future work indicated by the primary studies?*), we extracted the future work from each primary study and identified keywords indicating ways of improving the approaches and new applications. Figure 9 shows all the future work keywords split into actual applications (top part) or improvements to the approaches employing change reification (bottom part). In this figure, we have also differentiated between recent future work (i.e., called for in the past 5 years) and older future work. Interestingly enough, we find that most of the older future work is still called for in more recent articles. In the remainder of this section, we list and briefly explain the most frequent future work keywords as extracted from the primary studies. We start by explaining the possible applications (in Section 7.1) and follow this up by an overview of improvements of both the techniques employed as well as the ways of validating the techniques (in Section 7.2).

7.1. Future Applications of Reified Changes

Detect Patterns of Changes. The most frequently called-for application in future work is to *detect patterns of changes*. In the past 5 years, this has been suggested by Hashimoto and Mori [2010], Hayashi et al. [2012], and Soetens et al. [2013b]. In Section 6.2, we also mentioned that this is one of the most investigated applications. Most recently, Hashimoto et al. and Negara et al. have done research on this topic [Negara et al. 2014; Hashimoto et al. 2015]. Even though Negara et al. were capable of identifying new patterns of changes, there could still emerge new unknown patterns when replicating the experiment on other (commercial) cases. It is therefore worthwile to continue this line of research.

Replaying Changes. The next-most-mentioned line of future work is *replaying changes*. Again, this is also one of the most frequently investigated applications. However, only one interpretation of this application has to date been investigated in depth. The other interpretation—replaying changes in a different context—has been called for by Hashimoto and Mori [2010], Soetens and Demeyer [2012], and De Roover et al. [2014]. Hashimoto et al. suggest to replay generic patches for linux drivers, whereas Soetens et al. as well as De Roover et al. both suggest to use changes in order to replay bug fixes on other related systems. For instance, a bug fix could be fixed on one branch of a system, yet the bug might still exist in parallel branches, where replaying the changes of the bug fix could be a useful operation.

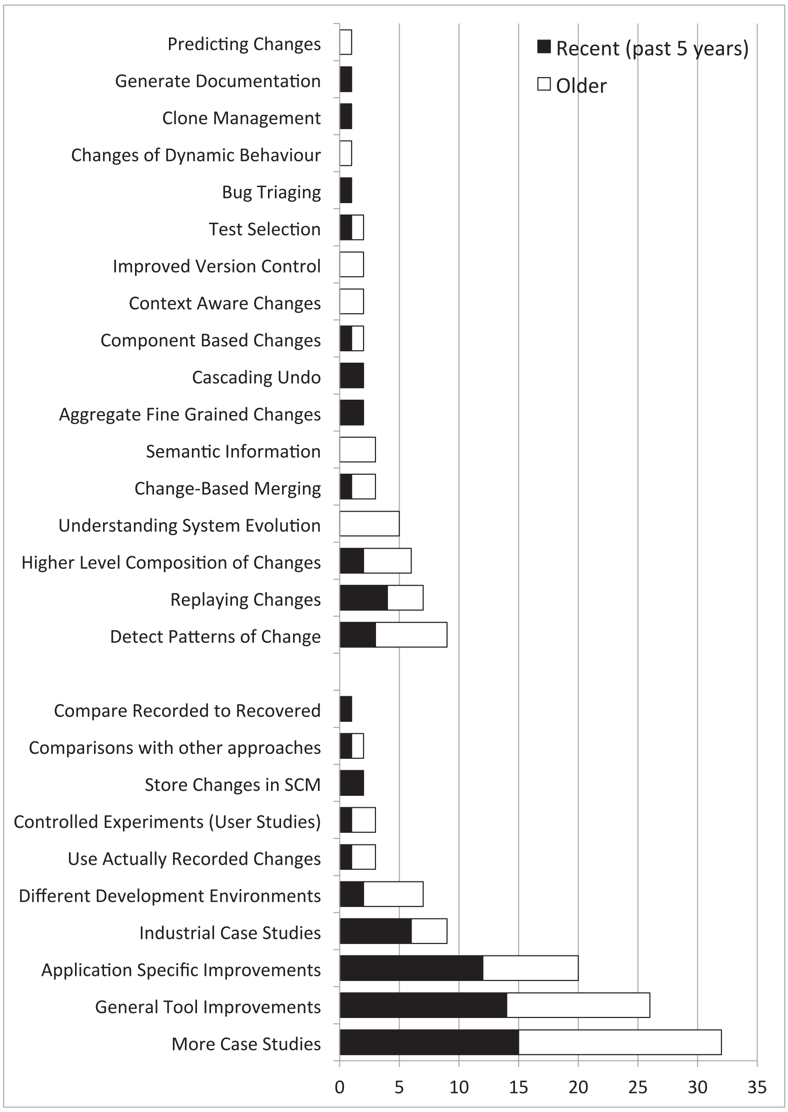

Fig. 9. Future work mentioned in the primary studies.

Change Composition. *Higher-order composition of changes* is an interesting line of future work that has seen little research in the past. It has most recently been called for by Falleri et al. [2014] and Hayashi et al. [2015]. We also see that this is closely related to the previous two lines of future work, where a possible use case could be as follows: Identify a frequently occurring pattern of changes, compose this pattern into a single higher-order change, and, finally, replay the abstract higher-order change pattern in different contexts. The higher-order composition of changes also includes defining refactorings as compositions of smaller changes, which has, to some extent, been done by Prete et al. [2010], Soetens et al. [2013b], and Hashimoto et al. [2015].

Understand System Evolution. This is by far the most investigated application in the primary studies. Articles that called for future work in this line are all relatively old [Dig et al. 2006; Kim et al. 2007; Omori and Maruyama 2009; Robbes et al. 2007; Xing and Stroulia 2006c], yet some of the future work they mentioned is still relevant today. Omori et al. suggest that to efficiently understand software evolution one needs to analyse a combination of fine-grained and coarse-grained change histories. However, given the popularity of this application it is a prime target for a more in-depth literature review and comparisons between different techniques.

Others. Although Dig et al. have developed an intelligent merging system [Dig et al. 2008], it is interesting to see if the explicit representation of fine-grained changes allows for more intelligent merging algorithms. *Change-based merging* should be able to take a system's syntactical structure into account, rather than the traditional text-based merge algorithms deployed by state-of-the-art SCM systems [Denker et al. 2007; Dig et al. 2008; Hashimoto and Mori 2010].

Hattori et al. and Kitsu et al. argue that the sheer number of changes recorded from development sessions impedes the program comprehension efforts needed [Hattori et al. 2013; Kitsu et al. 2013]. Thus they suggest that methods of aggregating changes into more manageable numbers are in order if one wants to provide a general overview of the change history of a system. Kitsu et al. have already done this to some extent, and yet their approach can still be improved by more advanced aggregation algorithms [Kitsu et al. 2013].

In contrast to a linear history model, the change model allows the user to undo a change other than the last one. The dependencies between the changes can be used to achieve a *cascading undo*, in which all change operations that depend on the undone change are also undone in a transitive way [Hayashi and Saeki 2010; Soetens and Demeyer 2012].

Ebraert et al. as well as Romano and Pinzger suggest to apply this idea of change reification on component-based systems [Ebraert et al. 2007a; Romano and Pinzger 2011]. Thus the meta-model of the *Subject*s should model component entities like components, provided services, required services, ports to those services, and connectors to connect the ports.

7.2. Improved Use and Validation of Reified Changes

Replication. The most called-for track of future work is to perform more case studies (in total 32 articles) and some articles even call specifically for industrial cases (in total 9 articles). Indeed, replication should be an important part of any reengineering research, and replication experiments should be valued accordingly. Unfortunately, that is rarely the case: To quote Tonella et al. when analysing the current state of the art in reverse engineering research:

> *Replication is not supported, industrial cases are rare, [. . .]. In order to help the discipline mature, we think that more systematic empirical evaluation is needed.* [Tonella et al. 2007]

Yet there is a downside to replicating approaches that record changes in an industrial context, as this requires a change recording tool to be deployed at a company that needs to run and collect changes for a fixed amount of time. In contrast to this, change recovering tools can (almost) instantly work with the readily available data in SCM systems.

General Tool Improvements. Apart from replication, many articles suggest future work to improve their tooling infrastructure, meaning that in order for the tool or application to become viable for use in industry, it needs to be made more efficient or more performant. Giger et al. suggest to store changes in a database in order to improve performance [Giger et al. 2012a]. Hattori and Lanza suggest that the collected changes should be cleaned for "noise"—changes collected about a trial-and-error programming style that are of no interest for the application [Hattori and Lanza 2009b].

Application Specific Improvements. Most articles mention some ideas of how to improve on their specific application. For instance, Fluri et al. suggest that in order to better understand how source code and comments co-evolve, one could improve the way comments are identified by taking into account the scope of the comment or by filtering out source code that is commented out [Fluri et al. 2007a]. Another example of application-specific improvements is by Omori and Maruyama as well as Falleri et al., who want to take into account changes that happen across multiple files [Omori and Maruyama 2009, 2011; Falleri et al. 2014].

Support for Different Development Environments. A specific kind of tool/application improvement that is called for in many articles is to port the tool to different IDE's and/or programming languages. For instance, Giger et al. mention that their results might be biased towards the Eclipse IDE and that it might be worthwhile to investigate if their conclusions hold when implementing their approach in other development environments [Giger et al. 2011, 2012b]. Yoon and Myers also report that it might be interesting to look at issues that cross IDE's and programming languages [Yoon and Myers 2011]. Other approaches were at the time of their writing only implemented in a Smalltalk environment [Ebraert et al. 2007a; Robbes and Lanza 2008c]. Robbes et al. proposed to port Spyware to more popular programming environments like Eclipse in order to isolate language- and environment-independent concepts [Robbes et al. 2007]. By now Ebraert's tool ChEOPS and Robbes' Spyware have been translated to Eclipse variants in the forms of Soetens' ChEOPSJ and Hattori's Syde [Ebraert et al. 2007a; Robbes and Lanza 2008c; Hattori and Lanza 2010; Soetens and Demeyer 2012]. Nevertheless, this highlights one of the downsides of change reification approaches: They are highly programming language dependent and, in the case of change recording approaches, IDE dependent; this necessitates extra effort to broaden the applicability of this approach.

Others. As an extension to the previously mentioned *industrial case studies*, some authors suggest to deploy change recording tools in a real development team [Hattori and Lanza 2009b; Robbes 2007; Soetens et al. 2013b]. As such, we can perform studies on changes recorded in the wild. Developers from industry can also be used user studies to evaluate the useability of existing tools and applications [Soetens et al. 2010; Stoerzer et al. 2006; Xing and Stroulia 2007a].

Soetens et al. and Yoon et al. suggest that fine-grained changes should be stored alongside the code in the software repositories. This would allow several developers to work simultaneously on the same change model [Soetens et al. 2013b; Yoon et al. 2013]. Yet one could argue that the change model itself represents a software system as well as its entire evolution, and therefore it should be sufficient to merely store the change model (without separately storing the source code).

Future Work in Literature Reviews. Apart from the future work extracted from the primary studies, by doing this mapping study, we have also identified future work with regards to literature reviews. There is a need for more in-depth and more complete literature reviews and comparisons in the following applications and approaches:

—Approaches for recording of fine-grained changes.
—Approaches for recovering of fine-grained changes.
—Approaches for recording of refactorings.
—Approaches for recovering of refactorings.
—Understanding system evolution.

—Detecting "unknown" patterns of changes.
—Replaying changes on (i) a single system (i.e., replay a systems evolution) or on (ii) multiple systems (i.e., replay abstract changes in different contexts).

Some of these systematic literature reviews have already been done, albeit in Japanese, by Omori et al. and Choi et al. The first have published an article in Japanese in which they present a survey on techniques that record changes from an IDE [Omori et al. 2015]. Their study was limited to fine-grained code changes in the context of software evolution. The other, by Choi et al., presented a survey on refactoring reconstruction techniques based on change history analysis [Choi et al. 2015].

Lehnert et al. did a study to compare different change-based approaches and devised a taxonomy of changes, which we summarised in Section 2 [Lehnert et al. 2012].

Kim and Notkin did a study in 2006 comparing different ways of matching program elements across versions in order to track software entities that have been changes [Kim and Notkin 2006]. As we have shown in this article, new techniques have emerged to reconstruct changes from a source-code repository; therefore, this comparison is due for an update.

Mens and Tourwé made an extensive overview of the existing research (in 2004) in the field of software refactoring. They compared and discussed the works based on the types of refactoring activities and the types of software artefacts being refactored, specific techniques and formalisms to support these activities, as well as issues that need to be taken into account when building refactoring tools [Mens and Tourwé 2004].

There is still room for research in the area of change reification. Some interesting highlights on our roadmap include the following: finding even more frequent patterns of changes, reperforming changes in different contexts, composing changes into a higher-order one, replicating existing studies (preferably in an industrial setting), and more systematic literature reviews in each of the subdomains we identified.

8. CONCLUSIONS

In this article, we performed a systematic mapping study in order to answer the following research question:

RQ: *To what purposes has change reification been used, what is the evidence for the success of approaches employing it, and what directions of further research are considered important?*

We have shown that the number of articles published on change reification has seen a steady increase, followed by a period of stability. Furthermore, more than 100 authors have been involved in these works. This shows that the software engineering research community found the topic worthy of sustained interest. Additionally, articles on change reification have been published at strong conference venues (e.g., ICSE, ASE) and in highly ranked journals (e.g., TSE). As such, we found convincing evidence for the success of change reification.

We provided an overview of applications for which change reification has been used in the past. Most prominently, there have been approaches for replaying changes, predicting future changes, test selection, correlating changes to other metrics, change impact analysis, discovering patterns of changes, and better understanding of software evolution. There are also a variety of other approaches that have been developed, showing that change reification is versatile and has a broad range of applications.

Finally, we have also provided an overview of the vision that researchers in this domain put forth by listing the most popular future applications, which includes detecting more patterns of changes, replaying changes on other systems, and higher-order composition of changes. Moreover, we also find that future work is also called for to improve the existing applications as well as their validation by replicating past studies and comparing different approaches as well as new experiments using actual software developers. Other applications envisioned in future work further show the broad potential of change reification.

In recent years, there has been a resurgence followed by sustained interest for change reification. Dig et al. have even received the most influential article award at ICSME 2015 [Dig and Johnson 2005]. We believe this trend is far from over. We envision a future where IDEs will use first-class changes as a basic mechanism in their architecture, broadcasting changes between IDEs, and using change data for highly accurate recommendations to increase the modularity of highly complex systems, to dynamically evolve them, and to definitively bridge the gap between programming and modelling. Some of these applications already exist as prototypes (see, for instance, the Change-Oriented Programming Environment[4]), and others are already in daily use.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

REFERENCES

Raihan Al-Ekram, Archana Adma, and Olga Baysal. 2005. DIFFX: an algorithm to detect changes in multi-version XML documents. In *Proc. of the Conf. of the Centre for Advanced Studies on Collaborative Research (CASCON'05)*. IBM Press, 1–11.

Lars Bendix and Fabio Vitali. 1999. VTML for fine-grained change tracking in editing structured documents. In *Proc. of the 9th Int. Symposium on System Configuration Management (SCM-9)*. Springer-Verlag, London, UK, 139–156.

Amel Bennaceur, Robert France, Giordano Tamburrelli, Thomas Vogel, Pieter J. Mosterman, Walter Cazzola, Fabio M. Costa, Alfonso Pierantonio, Matthias Tichy, Mehmet Akşit, Pär Emmanuelson, Huang Gang, Nikolaos Georgantas, and David Redlich. 2014. *Mechanisms for Leveraging Models at Runtime in Self-adaptive Software*. Springer International Publishing, Cham, 19–46. DOI: http://dx.doi.org/10.1007/978-3-319-08915-7_2

Xavier Blanc, Isabelle Mounier, Alix Mougenot, and Tom Mens. 2008. Detecting model inconsistency through operation-based model construction. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*. ACM, New York, NY, 511–520. DOI: http://dx.doi.org/10.1145/1368088.1368158

David Budgen, Mark Turner, Pearl Brereton, and Barbara Kitchenham. 2008. Using mapping studies in software engineering. In *Proc. of the 20th Annual Meeting of the Pschology of Programming Interest Group (PPIG'08)*. Lancaster University, 195–204.

Ophelia C. Chesley, Xiaoxia Ren, and Barbara G. Ryder. 2005. Crisp: A debugging tool for java programs. In *Proc. of the 21st IEEE Int. Conf. on Soft. Maintenance (ICSM'05)*. 401–410.

Eunjong Choi, Kenji Fujiwara, Norihiro Yoshida, and Shinpei Hayashi. 2015. A survey of refactoring detection techniques based on change history analysis (in Japanese). *Comput. Soft.* 32, 1 (Feb 2015), 47–59.

István Dávid, István Ráth, and Dániel Varró. 2016. Foundations for streaming model transformations by complex event processing. *Softw. Syst. Model.* (2016), 1–28. DOI: http://dx.doi.org/10.1007/s10270-016-0533-1

Coen De Roover, Christophe Scholliers, Viviane Jonckers, Javier Pérez, Alessandro Murgia, and Serge Demeyer. 2014. The implementation of the CHA-Q meta-model: A comprehensive, change-centric soft. representation. In *Proc. of the 8th Int. Workshop on Soft. Quality. ECEASST* 65 (2014).

Marcus Denker, Marcus Denker, Oscar Nierstrasz, Lukas Renggli, and Pascal" Zumkehr. 2007. Encapsulating and exploiting change with changeboxes. In *Proc. of the 2007 Int. Conf. on Dynamic Languages (ICDL'07)*. 25–49.

[4]http://cope.eecs.oregonstate.edu.

Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. 2015. Untangling fine-grained code changes. In *Proc. of the 22nd IEEE Int. Conf. on Soft. Analysis, Evolution and Reengineering (SANER'15)*. 341–350.

Martin Dias, Damien Cassou, and Stéphane Ducasse. 2013. Representing code history with development environment events. In *Proc. of the 5th Int. Workshop on Smalltalk Technologies (IWST'13)*. http://arxiv.org/abs/1309.4334

Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. 2006. Automated detection of refactorings in evolving components. In *Proc. of the 20th European Conf. on Object-Oriented Programming (ECOOP'06)*. Springer-Verlag, Berlin, 404–428. http://dx.doi.org/10.1007/11785477_24

Danny Dig, Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. 2005. Automatic detection of refactorings for libraries and frameworks. In *Proc. of the 6th ECOOP Workshop on Object-Oriented Reengineering (WOOR'05)*.

Danny Dig and Ralph Johnson. 2005. The role of refactorings in API evolution. In *Proc. of the 21st IEEE Int. Conf. on Soft. Maintenance (ICSM'05)*. 389–398.

Danny Dig, Kashif Manzoor, Ralph Johnson, and Tien N. Nguyen. 2007. Refactoring-aware configuration management for object-oriented programs. In *Proc. of the 29th Int. Conf. on Soft. Eng. (ICSE'07)*. 427–436.

Danny Dig, Kashif Manzoor, Ralph Johnson, and Tien N. Nguyen. 2008. Effective soft. merging in the presence of object-oriented refactorings. *IEEE Trans. Softw. Eng.* 34, 3 (May 2008), 321–335.

Nicolas Dintzner, Arie Van Deursen, and Martin Pinzger. 2013. Extracting feature model changes from the linux kernel using FMDiff. In *Proc. of the 8th Int. Workshop on Variability Modelling of Soft.-Intensive Systems (VaMoS'14)*. ACM, New York, NY, Article 22, 8 pages. http://doi.acm.org/10.1145/2556624.2556631

Hannes Dohrn and Dirk Riehle. 2014. Fine-grained change detection in structured text documents. In *Proc. of the 2014 ACM Symposium on Document Eng. (DocEng'14)*. ACM, New York, NY, 87–96. http://doi.acm.org/10.1145/2644866.2644880

Peter Ebraert. 2008. First-class change objects for feature-oriented programming. In *Proc. of the 15th Working Conf. on Reverse Eng. (WCRE'08)*. 319–322.

Peter Ebraert, Theo D'Hondt, and Tom Mens. 2004. Enabling dynamic soft. evolution through automatic refactoring. In *Proc. of the Int. Workshop on Soft. Evolution Transformations (SETra'04)*, Ying Zhou and James R. Cordy (Eds.). 3–6.

Peter Ebraert, Quinten David Soetens, and Dirk Janssens. 2011. Change-based foda diagrams: bridging the gap between feature-oriented design and implementation. In *Proc. of the 2011 ACM Symposium on Applied Computing (SAC'11)*. ACM, New York, NY, 1345–1352. http://doi.acm.org/10.1145/1982185.1982478

Peter Ebraert, Jorge Vallejos, Pascal Costanza, Ellen Van Paesschen, and Theo D'Hondt. 2007a. Change-oriented soft. eng.. In *Proc. of the 2007 Int. Conf. on Dynamic Languages: In Conjunction with the 15th Int. Smalltalk Joint Conf. 2007 (ICDL'07)*. ACM, New York, NY, 3–24. http://doi.acm.org/10.1145/1352678.1352680

Peter Ebraert, Ellen Van Paesschen, and Theo D'Hondt. 2007b. *Change-Oriented Round-Trip Eng.* Technical Report. Vrije Universiteit Brussel. Retrieved from ftp://prog.vub.ac.be/tech_report/2007/vub-prog-tr-07-04.pdf.

Johan Fabry, Tom Dinkelaker, Jacques Noyé, and Éric Tanter. 2015. A taxonomy of domain-specific aspect languages. *ACM Comput. Surv.* 47, 3 (2015), 3:1–3:44. DOI:http://dx.doi.org/10.1145/2685028

Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Montperrus. 2014. Fine-grained and accurate source code differencing. In *Proc. of the 29th IEEE/ACM Int. Conf. on Automated Soft. Eng. (ASE'14)*. ACM, New York, NY, 313–324. http://doi.acm.org/10.1145/2642937.2642982

Beat Fluri and Harald C. Gall. 2006. Classifying change types for qualifying change couplings. In *Proc. of the 14th IEEE Int. Conf. on Program Comprehension (ICPC'06)*. 35–45.

Beat Fluri, Harald C. Gall, and Martin Pinzger. 2005. Fine-grained analysis of change couplings. In *Proc. of the 5th IEEE Int. Workshop on Source Code Analysis and Manipulation (SCAM'05)*. IEEE Computer Society, Washington, DC, 66–74. http://dx.doi.org/10.1109/SCAM.2005.14

Beat Fluri, Emanuel Giger, and Harald C. Gall. 2008a. Discovering patterns of change types. In *Proc. of the 23rd IEEE/ACM Int. Conf. on Automated Soft. Eng. (ASE'08)*. 463–466.

Beat Fluri, Michael Wursch, and Harald C. Gall. 2007a. Do code and comments co-evolve? On the relation between source code and comment changes. In *Proc. of the 14th Working Conf. on Reverse Eng. (WCRE'07)*. 70–79.

Beat Fluri, Michael Wursch, Martin Pinzger, and Harald C. Gall. 2007b. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.* 33, 11 (Nov 2007), 725–743.

Beat Fluri, Jonas Zuberbühler, and Harald C. Gall. 2008b. Recommending method invocation context changes. In *Proc. of the 2008 Int. Workshop on Recommendation Systems for Soft. Eng. (RSSE'08)*. ACM, New York, NY, 1–5. http://doi.acm.org/10.1145/1454247.1454249

Harald C. Gall, Beat Fluri, and Martin Pinzger. 2009. Change analysis with evolizer and ChangeDistiller. *IEEE Soft.* 26, 1 (Jan 2009), 26–33.

Emanuel Giger, Marco D'Ambros, Martin Pinzger, and Harald C. Gall. 2012a. Method-level bug prediction. In *Proc. of the ACM-IEEE Int. Symposium on Empirical Soft. Eng. and Measurement (ESEM'12)*. 171–180.

Emanuel Giger, Martin Pinzger, and Harald C. Gall. 2011. Comparing fine-grained source code changes and code churn for bug prediction. In *Proc. of the 8th IEEE Working Conf. on Mining Soft. Repositories (MSR'11)*. ACM, New York, NY, 83–92. http://doi.acm.org/10.1145/1985441.1985456

Emanuel Giger, Martin Pinzger, and Harald C. Gall. 2012b. Can we predict types of code changes? An empirical analysis. In *Proc. of the 9th IEEE Working Conf. on Mining Soft. Repositories (MSR'12)*. IEEE Press, Piscataway, NJ, 217–226.

Carsten Görg and Peter Weißgerber. 2005a. Detecting and visualizing refactorings from software archives. In *Proc. of the 13th Int. Workshop on Program Comprehension (IWPC'05)*. 205–214.

Carsten Görg and Peter Weißgerber. 2005b. Error detection by refactoring reconstruction. In *Proc. of the 2nd Int. Workshop on Mining Soft. Repositories. SIGSOFT Softw. Eng. Notes* 30, 4 (May 2005), 1–5. http://doi.acm.org/10.1145/1082983.1083148

Masatomo Hashimoto and Akira Mori. 2008. Diff/TS: A tool for fine-grained structural change analysis. In *Proc. of the 15th Working Conf. on Reverse Eng. (WCRE'08)*. 279–288.

Masatomo Hashimoto and Akira Mori. 2010. A method for analyzing code homology in genealogy of evolving soft. In *Fundamental Approaches to Soft. Eng.*, David S. Rosenblum and Gabriele Taentzer (Eds.). Lecture Notes in Computer Science, Vol. 6013. Springer, Berlin, 91–106. http://dx.doi.org/10.1007/978-3-642-12029-9_7

Masatomo Hashimoto and Akira Mori. 2012. Enhancing history-based concern mining with fine-grained change analysis. In *Proc. of the 16th European Conf. on Soft. Maintenance and Reengineering (CSMR'12)*. 75–84.

Masatomo Hashimoto, Akira Mori, and Tomonori Izumida. 2015. A comprehensive and scalable method for analyzing fine-grained source code change patterns. In *Proc. of the 22nd IEEE Int. Conf. on Soft. Analysis, Evolution and Reengineering (SANER'15)*. 351–360.

Lile Hattori. 2010. Enhancing collaboration of multi-developer projects with synchronous changes. In *Proc. of the 32nd Int. Conf. on Soft. Eng. (ICSE'10)*, Vol. 2. 377–380.

Lile Hattori, Marco D'Ambros, Michele Lanza, and Mircea Lungu. 2011. Soft. evolution comprehension: Replay to the rescue. In *Proc. of the 19th Int. Conf. on Program Comprehension (ICPC'11)*. IEEE, 161–170.

Lile Hattori, Marco D'Ambros, Michele Lanza, and Mircea Lungu. 2013. Answering soft. evolution questions. *Inf. Softw. Technol.* 55, 4 (April 2013), 755–775. http://dx.doi.org/10.1016/j.infsof.2012.09.001

Lile Hattori and Michele Lanza. 2009a. An environment for synchronous software development. In *Companion of the 31th Int. Conf. on Soft. Eng. (ICSE Companion'09)*. 223–226.

Lile Hattori and Michele Lanza. 2009b. Mining the history of synchronous changes to refine code ownership. In *Proc. of the 6th IEEE Int. Working Conf. on Mining Soft. Repositories (MSR'09)*. 141–150.

Lile Hattori and Michele Lanza. 2010. Syde: A tool for collaborative software development. In *Proc. of the 32nd ACM/IEEE Int. Conf. on Soft. Eng. (ICSE'10)*, Vol. 2. 235–238.

Lile Hattori, Mircea Lungu, and Michele Lanza. 2010. Replaying past changes in multi-developer projects. In *Proc. of the Joint ERCIM Workshop on Soft. Evolution (EVOL) and Int. Workshop on Principles of Soft. Evolution (IWPSE) (IWPSE-EVOL'10)*. ACM, New York, NY, 13–22. http://doi.acm.org/10.1145/1862372.1862379

Shinpei Hayashi, Daiki Hoshino, Jumpei Matsuda, Motoshi Saeki, Takayuki Omori, and Katsuhisa Maruyama. 2015. Historef: A tool for edit history refactoring. In *Proc. of the 22nd IEEE Int. Conf. on Soft. Analysis, Evolution and Reengineering (SANER'15)*. 469–473.

Shinpei Hayashi, Takayuki Omori, Teruyoshi Zenmyo, Katsuhisa Maruyama, and Motoshi Saeki. 2012. Refactoring edit history of source code. In *Proc. of the 28th IEEE Int. Conf. on Soft. Maintenance (ICSM'12)*. 617–620.

Shinpei Hayashi and Motoshi Saeki. 2010. Recording finer-grained soft. evolution with IDE: An annotation-based approach. In *Proc. of the Joint ERCIM Workshop on Soft. Evolution (EVOL) and Int. Workshop on Principles of Soft. Evolution (IWPSE) (IWPSE-EVOL'10)*. ACM, New York, NY, 8–12. http://doi.acm.org/10.1145/1862372.1862378

Johannes Henkel and Amer Diwan. 2005. CatchUp! Capturing and replaying refactorings to support API evolution. In *Proc. of the 27th Int. Conf. on Soft. Eng. (ICSE'05)*. 274–283.

Jon Jenkins. 2011. Velocity Culture. (2011). Keynote Address at the Velocity 2011 Conference.

Sven Johann and Alexander Egyed. 2004. Instant and incremental transformation of models. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE'04)*. IEEE Computer Society, Washington, DC, 362–365. DOI:http://dx.doi.org/10.1109/ASE.2004.43

Huzefa H. Kagdi, Michael L. Collard, and Jonathan I. Maletic. 2007. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *J. Softw. Maint.* 19, 2 (2007), 77–131. DOI:http://dx.doi.org/10.1002/smr.344

David Kawrykow and Martin P. Robillard. 2011. Non-essential changes in version histories. In *Proc. of the 33rd Int. Conf. on Soft. Eng. (ICSE'11)*. ACM, New York, NY, 351–360. http://doi.acm.org/10.1145/1985793.1985842

Asad Masood Khattak, Khalid Latif, Manhyung Han, Sungyoung Lee, Young-Koo Lee, and Hyoung-Il Kim. 2009. Change tracer: Tracking changes in web ontologies. In *Proc. of the 21st Int. Conf. on Tools with Artificial Intelligence (ICTAI'09)*. 449–456.

Foutse Khomh, Bram Adams, Tejinder Dhaliwal, and Ying Zou. 2015. Understanding the impact of rapid releases on software quality. *Emp. Softw. Eng.* 20, 2 (2015), 336–373. DOI:http://dx.doi.org/10.1007/s10664-014-9308-x

Miryung Kim, Matthew Gee, Alex Loh, and Napol Rachatasumrit. 2010. Ref-finder: A refactoring reconstruction tool based on logic query templates. In *Proc. of the 18th ACM SIGSOFT Int. Symposium on Foundations of Soft. Eng. (FSE'10)*. ACM, New York, NY 371–372. http://doi.acm.org/10.1145/1882291.1882353

Miryung Kim and David Notkin. 2006. Program element matching for multi-version program analyses. In *Proc. of the 3rd Int. Workshop on Mining Soft. Repositories (MSR'06)*. ACM, New York, NY, 58–64. http://doi.acm.org/10.1145/1137983.1137999

Miryung Kim and David Notkin. 2009. Discovering and representing systematic code changes. In *Proc. of the 31st Int. Conf. on Soft. Eng. (ICSE'09)*. IEEE Computer Society, Washington, DC, 309–319. http://dx.doi.org/10.1109/ICSE.2009.5070531

Miryung Kim, David Notkin, and Dan Grossman. 2007. Automatic inference of structural changes for matching across program versions. In *Proc. of the 29th Int. Conf. on Soft. Eng. (ICSE'07)*. IEEE Computer Society, Washington, DC, 333–343. http://dx.doi.org/10.1109/ICSE.2007.20

Barbara Kitchenham and Stuart Charters. 2007. Guidelines for performing systematic literature reviews in soft. eng. (2007).

Eijiro Kitsu, Takayuki Omori, and Katsuhisa Maruyama. 2013. Detecting program changes from edit history of source code. In *Proc. of the 20th Asia-Pacific Soft. Eng. Conf. (APSEC'13)*, Vol. 1. 299–306.

Meir M. Lehman and Laszlo A. Belady. 1985. *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA.

Steffen Lehnert, Qurat-ul-ann Farooq, and Matthias Riebisch. 2012. A taxonomy of change types and its application in soft. evolution. In *Proc. of the 19th Int. Conf. and Workshops on Eng. of Computer Based Systems (ECBS'12)*. 98–107.

You Liang and Lu Yansheng. 2012. The atomic change set of java programming language. In *Proc. of the 7th Int. Conf. on Computer Science Education (ICCSE'12)*. 1090–1092.

Zhongpeng Lin. 2013. Understanding and simulating software evolution. In *Proc. of the 35th Int. Conf. on Soft. Eng. (ICSE'13)*. 1411–1414.

Zhongpeng Lin and Jim Whitehead. 2014. Using fine-grained code change metrics to simulate soft. evolution. In *Proc. of the 5th Int. Workshop on Emerging Trends in Soft. Metrics (WETSoM'14)*. ACM, New York, NY, 15–18. http://doi.acm.org/10.1145/2593868.2593871

Ernst Lippe and Norbert van Oosterom. 1992. Operation-based merging. *SIGSOFT Softw. Eng. Notes* 17, 5 (Nov. 1992), 78–87. http://doi.acm.org/10.1145/142882.143753

Alex Loh and Miryung Kim. 2010. LSdiff: A program differencing tool to identify systematic structural differences. In *Proc. of the 32nd ACM/IEEE Int. Conf. on Soft. Eng. (ICSE'10)*, Vol. 2. 263–266.

Cosmin Marsavina, Daniele Romano, and Andy Zaidman. 2014. Studying fine-grained co-evolution patterns of production and test code. In *Proc. of the 14th IEEE Int. Working Conf. on Source Code Analysis and Manipulation (SCAM'14)*. 195–204.

Katsuhisa Maruyama, Eijiro Kitsu, Takayuki Omori, and Shinpei Hayashi. 2012. Slicing and replaying code change history. In *Proc. of the 27th IEEE/ACM Int. Conf. on Automated Soft. Eng. (ASE'12)*. ACM, New York, NY, 246–249. http://doi.acm.org/10.1145/2351676.2351713

Tom Mens and Tom Tourwé. 2004. A survey of software refactoring. *IEEE Trans. Softw. Eng.* 30, 2 (Feb. 2004), 126–139. DOI:http://dx.doi.org/10.1109/TSE.2004.1265817

Bertrand Meyer, Christine Choppy, Jørgen Staunstrup, and Jan van Leeuwen. 2009. Viewpoint: research evaluation for computer science. *Commun. ACM* 52, 4 (Apr. 2009), 31–34. DOI:http://dx.doi.org/10.1145/1498765.1498780

Hausi Müller and Norha Villegas. 2014. *Runtime Evolution of Highly Dynamic Software*. Springer, Berlin, 229–264. DOI:http://dx.doi.org/10.1007/978-3-642-45398-4_8

Gail C. Murphy, Mik Kersten, and Leah Findlater. 2006. How are java soft. developers using the eclipse ide? *IEEE Softw.* 23, 4 (2006), 76–83.

Nachiappan Nagappan and Thomas Ball. 2005. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*. ACM, New York, NY, 284–292. DOI:http://dx.doi.org/10.1145/1062455.1062514

Stas Negara, Mihai Codoban, Danny Dig, and Ralph E. Johnson. 2013. *Mining continous code changes to detect frequent program transformations*. Technical Report. University of Illinois at Urbana-Champaign. http://hdl.handle.net/2142/43889

Stas Negara, Mihai Codoban, Danny Dig, and Ralph E. Johnson. 2014. Mining fine-grained code changes to detect unknown change patterns. In *Proc. of the 36th Int. Conf. on Soft. Eng. (ICSE'14)*. ACM, New York, NY, 803–813. http://doi.acm.org/10.1145/2568225.2568317

Stas Negara, Mohsen Vakilian, Nicholas Chen, Ralph E. Johnson, and Danny Dig. 2012. Is it dangerous to use version control histories to study source code evolution? In *Proc. of the 26th European Conf. on Object-Oriented Programming (ECOOP'12)*. Springer-Verlag, Berlin, 79–103. http://dx.doi.org/10.1007/978-3-642-31057-7_5

Oscar Nierstrasz, Marcus Denker, Tudor Girba, and Adrian Lienhard. 2006. analyzing, capturing and taming soft. change. In *Proc. of the Workshop on Revival of Dynamic Languages (co-located with ECOOP'06) (RDL'06)*. Nantes, France.

Dirk Ohst, Michael Welle, and Udo Kelter. 2003. Differences between versions of uml diagrams. *SIGSOFT Softw. Eng. Notes* 28, 5 (Sept. 2003), 227–236. http://doi.acm.org/10.1145/949952.940102

Takayuki Omori, Shinpei Hayashi, and Katsuhisa Maruyama. 2015. A survey on methods of recording fine-grained operations on integrated development environments and their applications (in japanese). *Computer Softw.* 32, 1 (Feb. 2015), 60–80.

Takayuki Omori and Katsuhisa Maruyama. 2008. A change-aware development environment by recording editing operations of source code. In *Proc. of the 5th Int. Working Conf. on Mining Soft. Repositories (MSR'08)*. ACM, New York, NY, 31–34. http://doi.acm.org/10.1145/1370750.1370758

Takayuki Omori and Katsuhisa Maruyama. 2009. Identifying stagnation periods in soft. evolution by replaying editing operations. In *Proc. of the 16th Asia-Pacific Soft. Eng. Conf. (APSEC'09)*. 389–396.

Takayuki Omori and Katsuhisa Maruyama. 2010. Flexibly highlighting in replaying operation history. In *Proc. of the Int. Workshop on Empirical Soft. Eng. in Practice (IWESEP'10)*. 59–60.

Takayuki Omori and Katsuhisa Maruyama. 2011. An editing-operation replayer with highlights supporting investigation of program modifications. In *Proc. of the 12th Int. Workshop on Principles of Soft. Evolution and the 7th Annual ERCIM Workshop on Soft. Evolution (IWPSE-EVOL'11)*. ACM, New York, NY, 101–105. http://doi.acm.org/10.1145/2024445.2024464

Ali Ouni, Marouane Kessentini, and Houari Sahraoui. 2013. Search-based refactoring using recorded code changes. In *Proc. of the 17th European Conf. on Soft. Maintenance and Reengineering (CSMR'13)*. 221–230.

Kai Petersen, Robert Feldt, Shahid Mujtaba, and Michael Mattsson. 2008. Systematic mapping studies in soft. eng.. In *Proc. of the 12th Int. Conf. on Evaluation and Assessment in Soft. Eng. (EASE'08)*. British Computer Society, Swinton, UK, 68–77.

Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. 2015. Guidelines for conducting systematic mapping studies in software engineering: an update. *Inf. Soft. Technol.* 64, 0 (2015), 1–18.

Kyle Prete, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim. 2010. Template-based reconstruction of complex refactorings. In *Proc. of the 26th IEEE Int. Conf. on Soft. Maintenance (ICSM'10)*. IEEE Computer Society, Washington, DC, 1–10. http://dx.doi.org/10.1109/ICSM.2010.5609577

Yin Qian, Sai Zhang, and Zhengwei Qi. 2008. Mining change patterns in aspectj soft. evolution. In *Proc. of the 2008 Int. Conf. on Computer Science and Soft. Eng. (CSSE'08)*, Vol. 2. 108–111.

Napol Rachatasumrit and Miryung Kim. 2012. An empirical investigation into the impact of refactoring on regression testing. In *Proc. of the 28th IEEE Int. Conf. on Soft. Maintenance (ICSM'12)*. 357–366.

Md Saidur Rahman and Chanchal K. Roy. 2014. A change-type based empirical study on the stability of cloned code. In *Proc. of the 14th IEEE Int. Working Conf. on Source Code Analysis and Manipulation (SCAM'14)*. 31–40.

Xiaoxia Ren, Barbara G. Ryder, Maximilian Stoerzer, and Frank Tip. 2005. Chianti: a change impact analysis tool for java programs. In *Proc. of the 27th Int. Conf. on Soft. Eng. (ICSE'05)*. ACM, New York, NY, 664–665. http://doi.acm.org/10.1145/1062455.1062598

Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. 2004. Chianti: a tool for change impact analysis of java programs. *SIGPLAN Notices* 39, 10 (Oct. 2004), 432–448. http://doi.acm.org/10.1145/1035292.1029012

Charles Rich and Richard C. Waters. 1990. *The Programmer's Apprentice*. ACM Press.

Romain Robbes. 2007. Mining a change-based soft. repository. In *Proc. of the 4th Int. Workshop on Mining Soft. Repositories (MSR'07)*. 15–15.

Romain Robbes and Michele Lanza. 2005. Versioning systems for evolution research. In *Proc. of the 8th Int. Workshop on Principles of Soft. Evolution (IWPSE'05)*. IEEE Computer Society, Washington, DC, 155–164. http://dx.doi.org/10.1109/IWPSE.2005.32

Romain Robbes and Michele Lanza. 2007a. A change-based approach to soft. evolution. *Electronic Notes in Theoretical Computer Science* 166, 0 (2007), 93–109. http://www.sciencedirect.com/science/article/pii/S1571066106005317 Proc. of the {ERCIM} Working Group on Soft. Evolution (2006).

Romain Robbes and Michele Lanza. 2007b. Characterizing and understanding development sessions. In *Proc. of the 15th IEEE Int. Conf. on Program Comprehension (ICPC'07)*. 155–166.

Romain Robbes and Michele Lanza. 2008a. Example-based program transformation. In *Model Driven Eng. Languages and Systems*, Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter (Eds.). Lecture Notes in Computer Science, Vol. 5301. Springer, Berlin, 174–188. http://dx.doi.org/10.1007/978-3-540-87875-9_13

Romain Robbes and Michele Lanza. 2008b. How program history can improve code completion. In *Proc. of the 23rd IEEE/ACM Int. Conf. on Automated Soft. Eng. (ASE'08)*. IEEE Computer Society, Washington, DC, 317–326. http://dx.doi.org/10.1109/ASE.2008.42

Romain Robbes and Michele Lanza. 2008c. Spyware: a change-aware development toolset. In *Proc. of the 30th Int. Conf. on Soft. Eng. (ICSE'08)*. ACM, New York, NY, USA, 847–850. http://doi.acm.org/10.1145/1368088.1368219

Romain Robbes and Michele Lanza. 2010. Improving code completion with program history. *Automated Softw. Eng.* 17, 2 (Jun. 2010), 181–212. http://dx.doi.org/10.1007/s10515-010-0064-x

Romain Robbes, Michele Lanza, and Mircea Lungu. 2007. An approach to soft. evolution based on semantic change. In *Proc. of the 10th Int. Conf. on Fundamental Approaches to Soft. Eng. (FASE'07)*. Springer-Verlag, Berlin, 27–41.

Romain Robbes, Damien Pollet, and Michele Lanza. 2008. Logical coupling based on fine-grained change information. In *Proc. of the 15th Working Conf. on Reverse Eng. (WCRE'08)*. 42–46.

Romain Robbes, Damien Pollet, and Michele Lanza. 2010. Replaying ide interactions to evaluate and improve change prediction approaches. In *Proc. of the 7th IEEE Working Conf. on Mining Soft. Repositories (MSR'10)*. 161–170.

Martin Robillard, Robert Walker, and Thomas Zimmermann. 2010. Recommendation systems for soft. eng. *IEEE Softw.* 27 (2010), 80–86.

Daniele Romano and Martin Pinzger. 2011. Using source code metrics to predict change-prone java interfaces. In *Proc. of the 2011 27th IEEE Int. Conf. on Soft. Maintenance (ICSM'11)*. IEEE Computer Society, Washington, DC, 303–312. http://dx.doi.org/10.1109/ICSM.2011.6080797

Daniele Romano and Martin Pinzger. 2012. Analyzing the evolution of web services using fine-grained changes. In *Proc. of the 19th Int. Conf. on Web Services (ICWS'12)*. 392–399.

Daniele Romano, Paulius Raila, Martin Pinzger, and Foutse Khomh. 2012. Analyzing the impact of antipatterns on change-proneness using fine-grained source code changes. In *Proc. of the 19th Working Conf. on Reverse Eng. (WCRE'12)*. 437–446.

Barbara G. Ryder and Frank Tip. 2001. Change impact analysis for object-oriented programs. In *Proc. of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Soft. Tools and Eng. (PASTE'01)*. ACM, New York, NY, 46–53. http://doi.acm.org/10.1145/379605.379661

Anita Sarma, David Redmiles, and André van der Hoek. 2008. Empirical evidence of the benefits of workspace awareness in software configuration management. In *Proc. of the 16th Int. Symposium on Foundations of Soft. Eng. (FSE'08)*. ACM Press, 113–123.

Curtis Schofield, Brendan Tansey, Zhenchang Xing, and Eleni Stroulia. 2006. Digging the development dust for refactorings. In *Proc. of the 14th IEEE Int. Conf. on Program Comprehension (ICPC'06)*. 23–34.

Ian Skerrett. 2013. *Eclipse Community Survey Results for 2013*. Market analysis report. Eclipse Foundation. Retrieved from http://www.eclipse.org/org/press-release/20130612_eclipsesurvey2013.php.

Quinten David Soetens and Serge Demeyer. 2012. Cheopsj: change-based test optimization. In *Proc. of the 16th European Conf. on Soft. Maintenance and Reengineering (CSMR'12)*. 535–538.

Quinten David Soetens, Serge Demeyer, and Andy Zaidman. 2013a. Change-based test selection in the presence of developer tests. In *Proc. of the 17th European Conf. on Soft. Maintenance and Reengineering (CSMR'13)*. 101–110.

Quinten David Soetens, Peter Ebraert, and Serge Demeyer. 2010. Avoiding bugs pro-actively by change-oriented programming. In *Proc. of the 1st Workshop on Testing Object-Oriented Systems (ETOOS'10)*. ACM, New York, NY, Article 7, 7 pages. http://doi.acm.org/10.1145/1890692.1890699

Quinten David Soetens, Javier Pérez, and Serge Demeyer. 2013b. An initial investigation into change-based reconstruction of floss-refactorings. In *Proc. of the 29th IEEE Int. Conf. on Soft. Maintenance (ICSM'13)*. 384–387.

Daniel Ståhl and Jan Bosch. 2014. Modeling continuous integration practice differences in industry software development. *J. Syst. Softw.* 87 (2014), 48–59. DOI:http://dx.doi.org/10.1016/j.jss.2013.08.032

Reinhout Stevens and Coen De Roover. 2014. Querying the history of soft. projects using qwalkeko. In *Proc. of the IEEE Int. Conf. on Soft. Maintenance and Evolution (ICSME'14)*. 585–588.

Maximilian Stoerzer, Barbara G. Ryder, Xiaoxia Ren, and Frank Tip. 2006. Finding failure-inducing changes in java programs using change classification. In *Proc. of the 14th ACM SIGSOFT Int. Symposium on Foundations of Soft. Eng. (SIGSOFT'06/FSE-14)*. ACM, New York, NY, 57–68. http://doi.acm.org/10.1145/1181775.1181783

Gabriele Taentzer, Claudia Ermel, Philip Langer, and Manuel Wimmer. 2014. A fundamental approach to model versioning based on graph modifications: from theory to implementation. *Softw. Syst. Model.* 13, 1 (2014), 239–272. DOI:http://dx.doi.org/10.1007/s10270-012-0248-x

Xiangchen Tan, Tie Feng, and Jiachen Zhang. 2007. Mapping soft. design changes to source code changes. In *Proc. of the 8th ACIS Int. Conf. on Soft. Eng., Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007 (SNPD'07)*, Vol. 2. 650–655.

Paolo Tonella, Marco Torchiano, Bart Du Bois, and Tarja Systä. 2007. Empirical studies in reverse engineering: state of the art and future trends. *Emp. Soft. Eng.* 12, 5 (2007), 551–571.

Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian P. Bailey, and Ralph E. Johnson. 2012. Use, disuse, and misuse of automated refactorings. In *Proc. of the 34th Int. Conf. on Soft. Eng. (ICSE'12)*. IEEE Press, Piscataway, NJ, 233–243.

Peter Weißgerber and Stephan Diehl. 2006a. Are refactorings less error-prone than other changes? In *Proc. of the 3rd Int. Workshop on Mining Soft. Repositories (MSR'06)*. ACM, New York, NY, 112–118. http://doi.acm.org/10.1145/1137983.1138011

Peter Weißgerber and Stephan Diehl. 2006b. Identifying refactorings from source-code changes. In *Proc. of the 21st IEEE/ACM Int. Conf. on Automated Soft. Eng. (ASE'06)*. 231–240.

Claes Wohlin. 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE'14)*. ACM, New York, NY, 38:1–38:10.

Zhenchang Xing. 2005. Design mentoring based on design evolution analysis. In *Proc. of the 27th Int. Conf. on Soft. Eng. (ICSE'05)*. ACM, New York, NY, 660–660.

Zhenchang Xing and Eleni Stroulia. 2003. Recognizing refactoring from change tree. In *Proc. of the Int. Workshop on REFactoring: Achievements, Challenges, Effects (REFACE'03)*. 41–44.

Zhenchang Xing and Eleni Stroulia. 2004a. Data-mining in support of detecting class co-evolution. In *Proc. of the 16th Int. Conf. on Soft. Eng. & Knowledge Eng. (SEKE'04)*. 123–128.

Zhenchang Xing and Eleni Stroulia. 2004b. Understanding class evolution in object-oriented software. In *Proc. of the 12th IEEE Int. Workshop on Program Comprehension (WPC'04)*. 34–43.

Zhenchang Xing and Eleni Stroulia. 2004c. Understanding phases and styles of object-oriented systems' evolution. In *Proc. of the 20th Int. Conf. on Soft. Maintenance (ICSM'04)*. 242–251.

Zhenchang Xing and Eleni Stroulia. 2005a. Analyzing the evolutionary history of the logical design of object-oriented soft. *IEEE Trans. Soft. Eng.* 31, 10 (Oct. 2005), 850–868.

Zhenchang Xing and Eleni Stroulia. 2005b. Towards experience-based mentoring of evolutionary development. In *Proc. of the 21st IEEE Int. Conf. on Soft. Maintenance (ICSM'05)*. IEEE Computer Society, Washington, DC, 621–624. http://dx.doi.org/10.1109/ICSM.2005.95

Zhenchang Xing and Eleni Stroulia. 2005c. Towards mentoring object-oriented evolutionary development. In *Proc. of the 21st Int. Conf. on Soft. Maintenance (ICSM'05)*. 621–624.

Zhenchang Xing and Eleni Stroulia. 2005d. Umldiff: an algorithm for object-oriented design differencing. In *Proc. of the 20th IEEE/ACM Int. Conf. on Automated Soft. Eng. (ASE'05)*. ACM, New York, NY, 54–65. http://doi.acm.org/10.1145/1101908.1101919

Zhenchang Xing and Eleni Stroulia. 2006a. Refactoring detection based on umldiff change-facts queries. In *Proc. of the 13th Working Conf. on Reverse Eng. (WCRE'06)*. 263–274.

Zhenchang Xing and Eleni Stroulia. 2006b. Refactoring practice: how it is and how it should be supported - an eclipse case study. In *Proc. of the 22nd IEEE Int. Conf. on Soft. Maintenance (ICSM'06)*. 458–468.

Zhenchang Xing and Eleni Stroulia. 2006c. Understanding the evolution and co-evolution of classes in object-oriented systems. *Int. J. Soft. Eng. Knowl. Eng.* 16, 1 (2006), 23–51.

Zhenchang Xing and Eleni Stroulia. 2007a. Api-evolution support with diff-catchup. *IEEE Trans. Soft. Eng.* 33, 12 (Dec. 2007), 818–836. http://dx.doi.org/10.1109/TSE.2007.70747

Zhenchang Xing and Eleni Stroulia. 2007b. Differencing logical uml models. *Automated Soft. Eng.* 14, 2 (Jun. 2007), 215–259. http://dx.doi.org/10.1007/s10515-007-0007-3

YoungSeok Yoon and Brad A. Myers. 2011. Capturing and analyzing low-level events from the code editor. In *Proc. of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU'11)*. ACM, New York, NY, 25–30. http://doi.acm.org/10.1145/2089155.2089163

YoungSeok Yoon, Brad A. Myers, and Sebon Koo. 2013. Visualization of fine-grained code change history. In *Proc. of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'13)*. 119–126.

Andreas Zeller. 2007. The future of programming environments: integration, synergy, and assistance. In *Proc. of the 2nd Conf. on The Future of Soft. Eng. (FOSE'07)*. IEEE Computer Society, Washington, DC, 316–325.

Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. 2011. Localizing failure-inducing program edits based on spectrum information. In *Proc. of the 27th IEEE Int. Conf. on Soft. Maintenance (ICSM'11)*. IEEE Computer Society, Washington, DC, 23–32. http://dx.doi.org/10.1109/ICSM.2011.6080769

Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. 2012. Faulttracer: a change impact and regression fault analysis tool for evolving java programs. In *Proc. of the ACM SIGSOFT 20th Int. Symposium on the Foundations of Soft. Eng. (FSE'12)*. ACM, New York, NY, Article 40, 4 pages. http://doi.acm.org/10.1145/2393596.2393642

Sai Zhang, Zhongxian Gu, Yu Lin, and Jianjun Zhao. 2008a. Celadon: a change impact analysis tool for aspect-oriented programs. In *Companion of the 30th Int. Conf. on Soft. Eng. (ICSE Companion'08)*. ACM, New York, NY, 913–914. http://doi.acm.org/10.1145/1370175.1370184

Sai Zhang, Zhongxian Gu, Yu Lin, and Jianjun Zhao. 2008b. Change impact analysis for aspectj programs. In *Proc. of the 24th IEEE Int. Conf. on Soft. Maintenance (ICSM'08)*. 87–96.

Thomas Zimmermann, Andreas Zeller, Peter Weißgerber, and Stephan Diehl. 2005. Mining version histories to guide software changes. *IEEE Trans. Soft. Eng.* 31, 6 (Jun. 2005), 429–445.

Received October 2015; revised December 2016; accepted January 2017