# ChangeMacroRecorder: Recording Fine-Grained Textual Changes of Source Code

Katsuhisa Maruyama
Ritsumeikan University, Japan
maru@cs.ritsumei.ac.jp

Shinpei Hayashi
Tokyo Institute of Technology, Japan
hayashi@c.titech.ac.jp

Takayuki Omori
Ritsumeikan University, Japan
tomori@is.ritsumei.ac.jp

***Abstract*—Recording code changes comes to be well recognized as an effective means for understanding the evolution of existing programs and making their future changes efficient. Although fine-grained textual changes of source code are worth leveraging in various situations, there is no satisfactory tool that records such changes. This paper proposes a yet another tool, called ChangeMacroRecorder, which automatically records all textual changes of source code while a programmer writes and modifies it on the Eclipse's Java editor. Its capability has been improved with respect to both the accuracy of its recording and the convenience for its use. Tool developers can easily and cheaply create their new applications that utilize recorded changes by embedding our proposed recording tool into them.**

***Index Terms*—Fine-grained changes, change recording, integrated development environments**

## I. Introduction

Software evolution is inevitable to keep up with an ever-changing context where it is managed and maintained [1]. To keep track of the fine-grained evolution of source code, commit-based versioning systems are unsatisfactory since they store the limited information into their repositories. To overcome this dissatisfaction, change recording (or logging) approaches obtain the details of changes that show what actually happened [2] although code changes are overlapped or tangled [3], [4]. Therefore, there are currently several change recording tools including SpyWare [5], Syde [6], ChEOPS [7], ChEOPSJ [8], OperationRecorder [9], Fluorite [10], and CodingTracker [3].

Among these tools, OperationRecorder and Fluorite attain the recordings of changes with finer levels of granularity by keeping track of modifications and updates of source code text on integrated development environments (IDEs). Textual changes are represented by the addition, deletion, and replacement of a text string. Here, we would emphasize that textual changes are versatile in supporting various kinds of programming activities since text is free from the successful build of an abstract syntax tree (AST) or other models. Moreover, changes of source code entities or AST nodes can be later inferred from the textual changes [11].

This paper proposes ChangeMacroRecorder (abbreviated as CMR). It is an Eclipse plugin that automatically records fine-grained textual changes of source code and actions involving those changes while programmers (developers or maintainers) write and modify their source code. Recorded textual changes and actions constitute a series of change macros (hereafter simply called macros). CMR is a yet another tool that improves the capability with respect to both the accuracy of recording textual changes and the convenience for using them.

Using CMR, tool developers such as tool vendors or researchers would be able to easily and cheaply create their new applications that leverage (e.g., analyze and visualize) fine-grained code changes. A remarkable application is a postponable refactoring tool [12], which allows a programmer to suspend the execution of an automatic refactoring if its preconditions are not satisfied and to restart the suspended refactoring once all the preconditions are satisfied. It embeds CMR to capture textual changes related to code fragments that might be affected by the suspended refactoring.

## II. Motivation

Although OperationRecorder and Fluorite are currently available, they are both designed to simply record textual changes performed on the Eclipse's Java editor and their related events. Therefore, their capabilities are unsatisfactory from two viewpoints of the accuracy and convenience of recorded textual changes. Using an example shown in Figure 1, which illustrates a series of textual changes and their corresponding macros recorded by CMR, we will explain two drawbacks that cause the above dissatisfaction. A macro (represented by $m_i$) corresponds to a unit of recorded textual change and stores information on the change. The details of macros will be described in Section III-A.

### A. Accuracy of Recording

OperationRecorder essentially obtains edit operations from the undo history of Eclipse. This history often loses textual changes that are distributed to multiple files. Fluorite logs document change events by using the built-in document listeners of Eclipse. These listeners capture events occurring in files that have been already opened on the editor, but exclude ones in not-opened files. Consequently, textual changes that both the tools can record are inaccurate.

In the example of code edit shown in Figure 1(a), the programmer first opened file P.java declaring the class P and inserted the text "`int efg = 0;`" into the body of the method `abc()`. The file open action was recorded as $m_1$ and $m_2$, and the text insertion and its related actions were recorded as $m_3$~$m_{16}$. Here, $m_i$~$m_j$ denotes every macro sandwiched between $m_i$ and $m_j$. Then, she changed the name

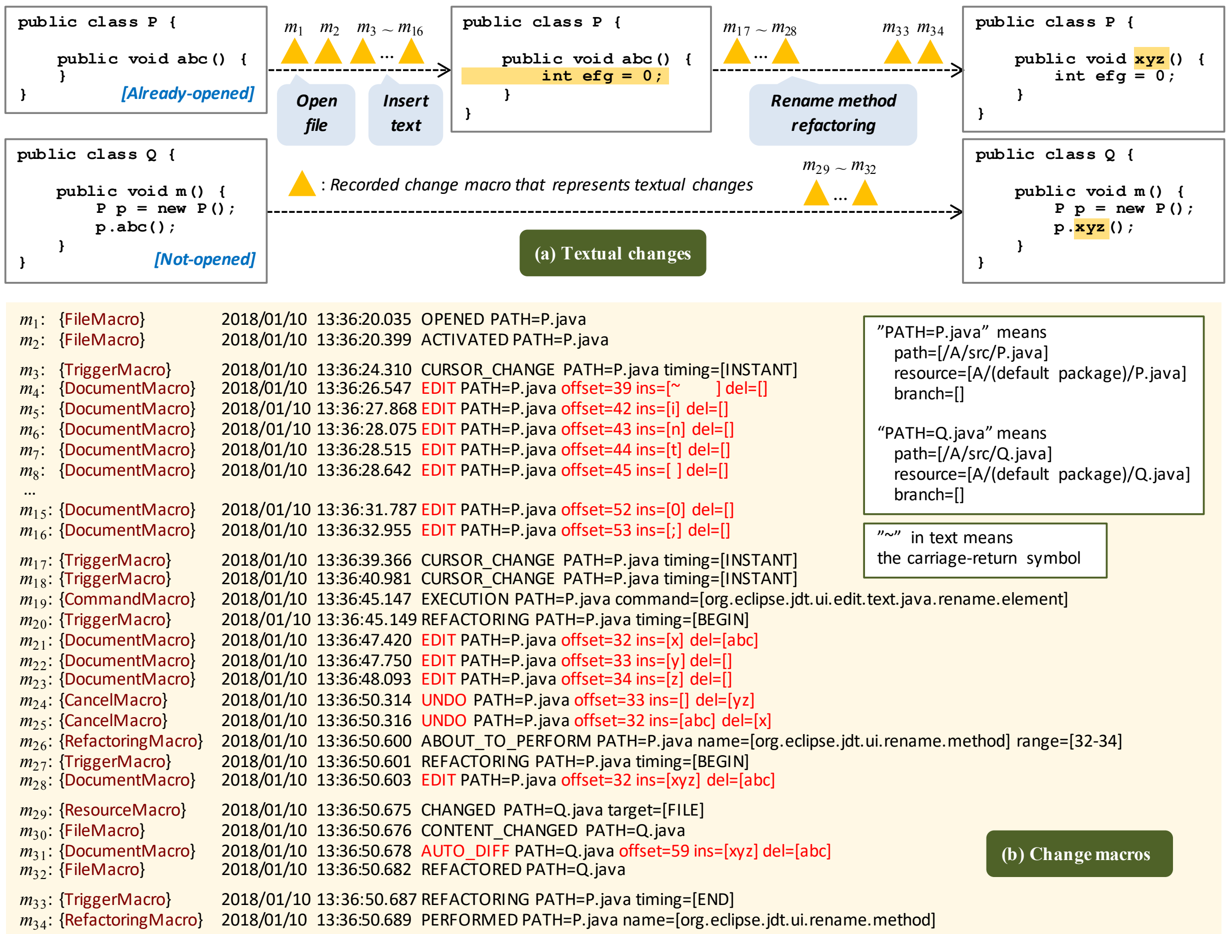


Fig. 1. Example of (a) textual changes actually performed and (b) their corresponding (raw) change macros recorded by CMR.

of `abc()` into `xyz` ($m_{17}$~$m_{34}$) by activating an automatic rename-method refactoring. Actually, she only both activated the refactoring and replaced the string `abc` with the string `xyz` in the inline dialog ($m_{21}$~$m_{23}$). The refactoring module behind the Eclipse's editor found a reference to `abc()` within the method `m()` of the class Q and updated it to refer `xyz()` ($m_{31}$). The attention point is that the file Q.java declaring Q was not opened on at the execution of the rename refactoring.

Unfortunately, neither OperationRecorder nor Fluorite can capture this textual change occurring in Q.java. In other words, no edit operation or event log corresponding to $m_{31}$ can be recorded. Moreover, existing tools excluding Coding-Tracker are incompatible with textual changes performed on the outside of their IDEs. In fact, CodingTracker can record events for outside resource changes but does not output their corresponding textual changes during the recording process. Since the violation of the time-series consistency of recorded textual changes hinders the comprehension of source code evolution, tool developers would require change recording tools to record textual changes with higher accuracy.

### *B. Convenience for Use*

Existing change recording tools including Operation-Recorder and Fluorite do not intend to facilitate the use of recorded changes although some of them can simply amend the changes. Therefore, tool developers must implement the functionality of investigating recorded changes in their own application tools. However, their tasks to classify, simplify, and aggregate recorded changes are greatly difficult if the changes do not contain information enough to be amended later. In general, to classify and simplify textual changes based on the contexts in which the changes occur, it is reasonable to record the execution of various kinds of programmers' and IDEs' actions with clearly distinct forms. Moreover, the change aggregation task needs to know the exact time of starting and ending an action (e.g., refactoring or code completion) involving multiple textual changes. The time of executing an action (e.g., saving a file, deleting a project) might be sometimes useful for determining the timing of analyzing textual changes. Unfortunately, almost all the existing tools fall short in presenting them without such convenient information.

For example, looking at the recorded macros shown in Figure 1(b), $m_{19}$, $m_{26}$, and $m_{34}$ indicate the activating, starting, and ending a refactoring, respectively. Additionally, $m_{20}$, $m_{27}$, and $m_{33}$ were inserted at their adequate positions. These macros are essential for determining that textual changes of $m_{28}$ and $m_{31}$ were involved with the applied refactoring. With respect to the simplification of macros, $m_{21}$, $m_{22}$, and

$m_{23}$ are accordingly verbose since the textual changes of these macros were undone by $m_{24}$ and $m_{25}$ in the duration between the activating and starting points of the refactoring. Tool developers would desire such information on the contexts of textual changes, in addition to simple information on them.

Here, we do not believe that only presenting rich information satisfies many tool developers. They also desire a feasible implementation that simplifies and aggregates textual changes, using the collected rich information. For example, Evolizer [13] is a platform that introduces two metamodels to ease development of change analysis tools. FeedBaG++ [14] employs a platform that provides tooling around enriched events. Unfortunately, there are seldom such tools for presenting powerful treatment for recorded textual changes.

Moreover, tool developers do not prefer the undue separation of textual changes. In Figure 1(b), the consecutively typed characters "`i`", "`n`", "`t`" were separately recorded in $m_5$, $m_6$, and $m_7$, although the text "`int`" was a keyword in source code. Since each character of an identifier or a keyword is too fine-grained to be managed based on programmers' intuitions, compressing consecutive textual changes would be certainly needed in most cases. To our knowledge, many of the existing change recording tools present inelegant textual changes as-is.

## III. Implementation

This section explains macros and significant improvements made in the implementation of CMR.

### A. Change Macros

To extract textual changes and detect their related actions, CMR employs seven modules that implement their respective dedicated listeners (e.g., IDocumentListener and IResourceChangeListener) embedded in Eclipse. The extracted textual changes and detected actions constitute nine kinds of basic macros.

The first two macros directly update of source code text. Each textual change (by typing text, cutting text, pasting text, and activating the undo or redo action) is represented by the insertion and/or deletion that made to source code. **DocumentMacro** stores an inserted text, a deleted text, and the offset (of the leftmost character) of the inserted or deleted text in the source code. **CancelMacro** is always paired with DocumentMacro. As described in Section II-B, a series of code manipulations in a rename refactoring under the inline mode causes the consistent result of source code but the recorded textual changes seem to be verbose. To give the opportunity of simplifying this verbosity, CMR introduces CancelMacro that undoes previous DocumentMacro corresponding to a programmer's text input.

The remaining seven macros denote the occurrences of programmers' actions that might or might not involve textual changes of the source code. **CopyMacro** stores a copied text and its offset to recover the occurrences of copy-paste actions although a copy action never changes source code text. **CommandMacro**, **CodeCompletionMacro**, **RefactoringMacro**, and **GitMacro** represent the execution of command services (including cut, copy, and paste actions), code completion actions by content assist, refactoring actions, and Git actions, respectively. **ResourceMacro** represents the property change of resources (files, packages, and projects). **FileMacro** stores source code text as needed when detecting the execution of file operations (adding, removing, opening, closing, saving, and activating), move and rename refactoring actions for files, and Git actions for files.

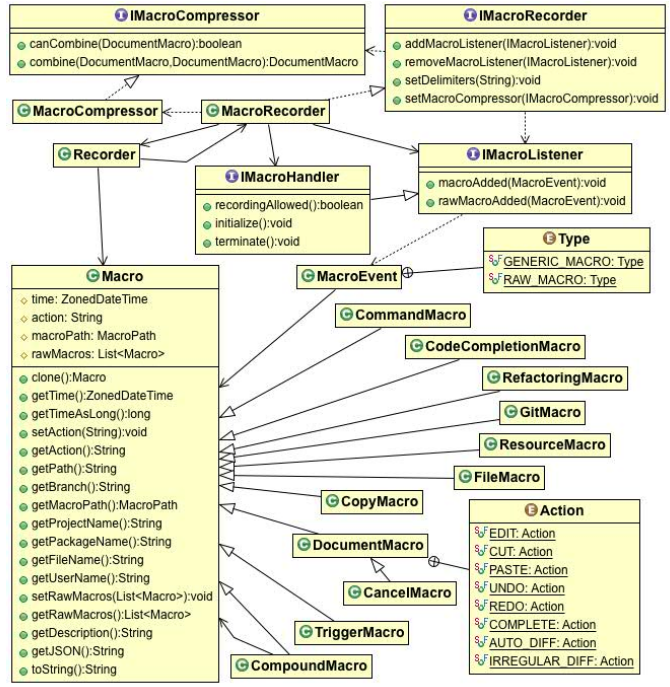


Fig. 2. Classes and interfaces related to change macros.

**TriggerMacro** is a special macro that just indicates a trigger to begin, end, or cancel pre-defined actions (refactoring, code completion, undoing, and redoing, Git command) that might cause composite changes. It also indicates a change of the cursor location, which means the completion or interruption of running actions and programmer's editing activities. A begin-end pair of TriggerMacro derives **CompoundMacro** that composes textual changes that are simultaneously made by the same action.

All eleven (nine plus additional two) kinds of macros store common information on the time when a textual change or an action was performed, the name of a changed or affected file, and the names of a project, package, and Git branch related to the file. Figure 2 depicts a class diagram containing classes that implement the eleven macros. It also shows primary classes and interfaces that provide the functionality of managing the macros. Some non-essential attributes, operations, and associations are omitted to simplify the diagram. Usage of some classes and interfaces will be explained in Section IV.

### B. Accurate Recording of Change Macros

As shown in Figure 2, the actions of DocumentMacro are divided into eight kinds, which are defined in the enum class Ac-

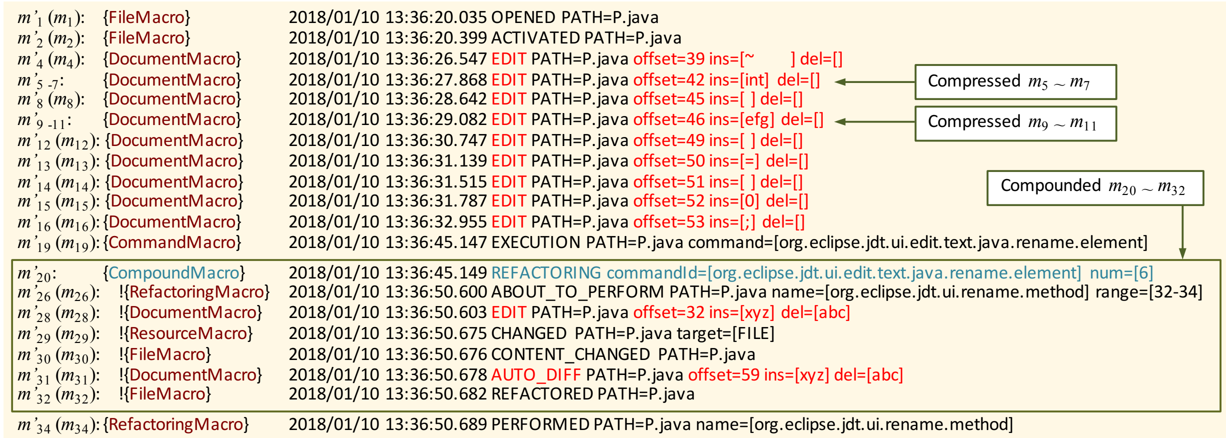


Fig. 3. Treated change macros that CMR generates.

tion dangling the class DocumentMacro. The constants `EDIT`, `CUT`, `PASTE`, `UNDO`, `REDO`, and `COMPLETE` simply indicate editing, cutting, pasting, undoing, redoing, code completion, respectively. Whereas these six actions are all usual in normal code editing, the two remaining constants `AUTO_DIFF` and `IRREGULAR_DIFF` correspond to special actions that attain the accurate recording of textual changes.

We consider that the conventional simple change recording does not address two possible cases that might decrease the accuracy of recording. The first case is that particular refactorings update not only the content of a file that has been already opened but that of a file that has not been opened on the editor. In this case, conventional change recording tools often overlook indirect textual changes in not-opened files. To overcome this drawback, CMR monitors the local history of a not-opened file and checks if its content is updated during the execution of refactoring. If any update occurs, it calculates textual differences between the contents of the file before and after the execution of refactoring by using *diff* utility. Each of the differences is transformed into either an inserted text or a deleted text. CMR automatically records DocumentMacro with `AUTO_DIFF`, which stores the text and its offset. In the example shown in Figure 1(a), $m_{31}$ corresponds to this macro since the file Q.java was not opened at the execution of the applied rename refactoring.

The other case is that almost all conventional change recording tools often ignore unexpected code manipulation that is performed on the outside of them or changes that are incorrectly captured due to the limitations of their recording implementations. Some of the tools take the snapshots of files at a specific time (e.g., file saving), they cannot always preserve the consistency of recorded textual changes. To record consistent textual changes in this case, CMR temporarily generates a text by applying a currently recorded macro to the previous content of a changed file, and checks the discrepancy between the generated text and the current content of the file. If there is any irregular discrepancy detected, CMR automatically records DocumentMacro with `IRREGULAR_DIFF`, which contains textual differences to reconcile the discrepancy.

### C. *Convenient Use of Change Macros*

A human-understandable representation of changes is convenient for tool developers who exploit the changes. To this end, CMR performs the treatment of raw macros through three processes: aggregating macros, compressing textual changes, and simplifying verbose ones. Figure 3 shows a series of macros that was obtained by applying these treatment processes to the untreated raw macros shown in Figure 1(b). Whereas the symbol $m$ indicates one of the raw macros, $m'$ with the prime mark indicates one of the treated macros. Two change macros with the same index number store the same information. A sequence of treated change macros includes CompoundMacro but excludes both CancelMacro and TriggerMacro.

As mentioned in Section III-A, a begin-end pair of TriggerMacro derives CompoundMacro. In Figure 1(b), $m_{20}$ and $m_{33}$ compose a begin-end pair of the applied refactoring, and thus CMR aggregated macros sandwiched between them. Finally, CompoundMacro $m'_{20}$ encloses the six macros with the exclamation mark through the other two treatment process, as shown in Figure 3. Note that the occurrences of TriggerMacro are not necessarily paired. In the example shown in Figure 1(b), $m_{27}$ and $m_{33}$ become unpaired since CMR skips $m_{27}$ to detect the outermost begin-end. If it detects a begin-cancel pair of TriggerMacro, it aborts a running aggregation process and converts macros enclosed by TriggerMacro into unenclosed ones. A similar aggregation process is performed when any code is automatically completed.

To compress textual changes that are stored in different DocumentMacro, CMR employs a delimiter-based compression strategy that relaxes "gluing" of CodingTracker since it might be understandable and can be easily implemented. It concatenates consecutive two texts if they contain no predefined delimiter. The default delimiters are all characters appearing in string "`␣\n\r,.;()[]{}`" (␣ means a space character). Looking at $m_5 \sim m_8$ shown in Figure 1(b), neither $m_5$, $m_6$, nor $m_7$ stored one of the delimiters. Therefore, all textual changes stored in these macros were compressed (i.e., the strings "i", "n", and "t" were concatenated into "int"). On the other hand, $m_8$ was not combined with $m_7$ since it stored the blank character. As a result, CMR generates DocumentMacro $m'_{5-7}$ storing the inserted text "int" and $m'_8$ storing the blank character. Here, tool developers can freely customize delimiters, and can also replace the prepared delimiter-based strategy with their own compression ones (e.g., time-period based compression of textual changes).

The simplification process is uncomplicated in the treatment of textual changes. CancelMacro is exactly responsible for canceling a verbose textual change and always appears in a begin-end or begin-cancel pair of TriggerMacro. It removes itself and its corresponding DocumentMacro from macros that are enclosed in CompoundMacro. In Figure 1(b), for example, $m_{24}$ and $m_{25}$ removed $m_{21}$, $m_{22}$, and $m_{23}$ along with themselves.

## IV. Usage

CMR is designed to be embedded into various application tools that leverage fine-grained textual changes, and adopts the event-listener model to notify the tools of recorded textual changes. This section explains how to utilize four primary interfaces IMacroRecorder, IMacroCompressor, IMacroListener, and IMacroHandler shown in Figure 2. Hereafter, code that a tool developer writes within her tool is called user code.

IMacroRecorder provides the functionality of managing the macro recording. Its concrete instance can be obtained by invoking the static method `getInstance()` of the class MacroRecorder. The methods `addMacroListener()` and `removeMacroListener()` of IMacroRecorder start and stop sending macros to a receiver instance of a class implementing IMacroListener, respectively. Once the user code registers or unregisters the receiver instance, it becomes able or unable to receive recorded macros. To customize delimiters, the user code passes a string containing all delimiter characters through the invocation to the method `setDelimiters()`. No compression is performed if `null` is given. On the other hand, the empty string denotes compression of all consecutive texts not chopped by any action. Moreover, user code can entirely replace a compression strategy of the class MacroCompressor with another one. In this case, it registers an instance of a class implementing the interface IMacroCompressor by invoking the method `setMacroCompressor()`.

To really receive macros, user code should prepare a receiver instance of a class that implements the methods `rawMacroAdded()` and `macroAdded()` of IMacroListener. Whereas the former receives recorded macros as-is, the latter receives treated ones. To be precise, CMR passes an instance of the class MacroEvent that stores a macro either with or without treatment, which can be distinguished based on its type (`GENERIC_MACRO` or `RAW_MACRO`).

Besides the above basic usage, an extension point for plug-ins is provided so that user code can easily register a receiver instance if it must record the whole textual change while Eclipse is running. In this case, the user code should define a class for the receiver instance, which implements the methods of IMacroHandler in addition to the methods of IMacroListener. The tool developer specifies the receiver class in the plug-in configuration file. To avoid this work, CMR also provides a wizard that both creates a template of the receiver class and registers it. The methods `initialize()` and `terminate()` are invoked when Eclipse starts and stops, respectively. The method `recordingAllowed()` determines whether the change recording is allowed or not. A receiver instance is successfully registered if `true` is returned, otherwise it is never registered.

## V. Conclusion

To promote the utilization of fine-grained textual changes of source code, a tool developer would expect a tool that can accurately record them and make them more convenient. Our proposed CMR is a candidate for this tool. The source code of its implementation is available at GitHub [1] with Eclipse Public License 1.0 (EPL-1.0). Moreover, several screencast demonstrations including the example of Figure 1 are presented at the site. An immediate future work is to collect a large volume of textual changes in real software development and maintenance using CMR.

## Acknowledgment

This work was sponsored by the Grant-in-Aid for Scientific Research (15H02685, 15K15970, 15H02683, 26730042).

## References

[1] Meir M. Lehman. Programs, life cycles, and laws of software evolution. *Proc. IEEE*, 68(9):1060–1076, 1980.
[2] Quinten David Soetens, Romain Robbes, and Serge Demeyer. Changes as first-class citizens: A research perspective on modern software tooling. *ACM Computer Surveys*, 50(2):18:1–18:38, 2017.
[3] Stas Negara, Mohsen Vakilian, Nicholas Chen, Ralph E. Johnson, and Danny Dig. Is it dangerous to use version control histories to study source code evolution? In *Proc. ECOOP '12*, pages 79–103, 2012.
[4] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *Proc. MSR '13*, pages 121–130, 2013.
[5] Romain Robbes and Michele Lanza. A change-based approach to software evolution. *Electronic Notes in Theoretical Computer Science*, 166:93–109, 2007.
[6] Lile Hattori and Michele Lanza. Syde: a tool for collaborative software development. In *Proc. ICSE '10*, pages 235–238, 2010.
[7] Peter Ebraert, Jorge Vallejos, Pascal Costanza, Ellen Van Paesschen, and Theo D'Hondt. Change-oriented software engineering. In *Proc. ICDL '07*, pages 3–24, 2007.
[8] Quinten D. Soetens and Serge Demeyer. ChEOPSJ: Change-based test optimization. In *Proc. CSMR '12*, pages 535–538, 2012.
[9] Takayuki Omori and Katsuhisa Maruyama. A change-aware development environment by recording editing operations of source code. In *Proc. MSR '08*, pages 31–34, 2008.
[10] YoungSeok Yoon and Brad A. Myers. Capturing and analyzing low-level events from the code editor. In *Proc. PLATEAU '11*, pages 25–30, 2011.
[11] Katsuhisa Maruyama, Takayuki Omori, and Shinpei Hayashi. Slicing fine-grained code change history. *IEICE Trans. Inf. Syst.*, E99(3):671–687, 2015.
[12] Katsuhisa Maruyama and Shinpei Hayashi. A tool supporting postponable refactoring. In *Proc. ICSE '17 (Companion)*, pages 133–135, 2017.
[13] Harald C. Gall, Beat Fluri, and Martin Pinzger. Change analysis with evolizer and changedistiller. *IEEE Software*, 26(1):26–33, 2009.
[14] Sebastian Proksch, Sarah Nadi, Sven Amann, and Mira Mezini. Enriching in-IDE process information with fine-grained source code history. In *Proc. SANER '17*, pages 250–260, 2017.

[1] https://github.com/katsuhisamaruyama/ChangeMacroRecorder