IEICE TRANS. INF. & SYST., VOL.E103–D, NO.11 NOVEMBER 2020

PAPER

ChangeMacroRecorder: Accurate Recording of Fine-Grained Textual Changes of Source Code*

Katsuhisa MARUYAMA[†a)], **Shinpei HAYASHI**[††], ***and*** **Takayuki OMORI**[†], ***Members***

SUMMARY Recording source code changes comes to be well recognized as an effective means for understanding the evolution of existing software and making its future changes efficient. Therefore, modern integrated development environments (IDEs) tend to employ tools that record fine-grained textual changes of source code. However, there is still no satisfactory tool that accurately records textual changes. We propose ChangeMacroRecorder that automatically and silently records all textual changes of source code and in real time correlates those textual changes with actions causing them while a programmer is writing and modifying it on the Eclipse's Java editor. The improvement with respect to the accuracy of recorded textual changes enables both programmers and researchers to exactly understand how the source code was evolved. This paper presents detailed information on how ChangeMacroRecorder achieves the accurate recording of textual changes and demonstrates how accurate textual changes were recorded in our experiment consisting of nine programming tasks.

key words: *fine-grained changes, change recording, change representation, integrated development environments*

1. Introduction

Software decays unless it keeps up with an ever-changing context where it is managed and maintained [1]. Moreover, it is hard to perfectly construct high reliable and secure software from the initial stage of its development. Therefore, software evolution is inevitable and the source code of a software system is continually modified. Under this situation, programmers (developers or maintainers) want to obtain exact information on when, how, and by whom the source code was changed. This is because source code tends to involve unexpected faults and/or undesirable vulnerabilities if a new change in the code accidentally overrides a past change that was performed according to a special requirement. Exact information on source code evolution is also beneficial for researchers who tackle empirical studies to elucidate source code evolution and programming activities. For example, Hora et al. pointed out the threat of untracked changes on results of MSR (mining software repositories) studies [2].

Unfortunately, commit-based versioning systems, such as Subversion, Git, and Mercurial, are unsatisfactory to keep track of the fine-grained evolution of source code since they store the limited information into their repositories [3], [4]. To overcome this dissatisfaction, change recording (or logging) approaches can obtain the details of changes that show what actually happened [5] although source code changes are overlapped or tangled [6]. For example, replaying fine-grained changes of source code of interest is more favorable to programmers than simply observing differences between its revisions when they would figure out refactoring applied to the source code [7]. For effective and efficient management of code- and design-technical debts [8], it is significant to investigate when and why code smells occur in source code [9], as well as how they actually emerge. Fine-grained source code changes, which cannot be stored in commit-based versioning systems, are likely to support this investigation since they accelerate more exact understanding of source code evolution. Additionally, such source code changes would be useful for resolving merge conflicts between variants of the source code [10].

From the above reasons, there are currently several change-oriented tools that treat changes (or transformations) as first-class entities. To our knowledge, among these tools, only OperationRecorder [11] (which has developed by two of the authors), Fluorite [12], and CodingTracker [4] can reify the changes as a series of operations storing information on the insertion, deletion, and replacement of a text string. OperationRecorder and Fluorite directly output fine-grained textual changes at finer levels of granularity by keeping track of edits (modifications and updates) of source code text on an Eclipse's Java editor. CodingTracker collects fine-grained textual changes to infer AST nodes on which the changes were performed on Eclipse's Java editor. Although these three tools fruit in their respective purposes, there are still unsatisfactory points on the accuracy of textual changes that they record. The accuracy means whether a recording tool can accurately record textual changes without excess and deficiency.

This paper proposes **ChangeMacroRecorder** (abbreviated as CMR hereafter). It is an Eclipse plug-in that automatically and silently records both fine-grained textual changes of source code and actions involving those changes while a programmer is writing and modifying her source code. Eclipse is still one of popular integrated development environments (IDEs) for Java programmers. Recorded textual changes and actions are represented as a series of change macros (hereafter simply called *macros*). CMR is

Manuscript received January 18, 2020.
Manuscript revised May 28, 2020.
Manuscript publicized August 24, 2020.

[†]The authors are with Department of Information Science and Engineering, Ritsumeikan University, Kusatsu-shi, 525–8577 Japan.
[††]The author is with School of Computing, Tokyo Institute of Technology, Tokyo, 152–8550 Japan.

*This is a paper on system development.

a) E-mail: maru@cs.ritsumei.ac.jp

DOI: 10.1587/transinf.2020EDK0001

yet another change-recording tool; nevertheless, it improves the accuracy of recorded textual changes. In our experiment consisting of nine programming tasks in source code evolution, CMR produced more accurate textual changes than the existing two tools, OperationRecorder and Fluorite. The reason why we selected these two tools in our comparative experiment is that their source code or executable code are currently available, which run on recent releases of Eclipse. Although the experiment used the small size of source code, this is the first trial that assesses the capability of change-recording tools with respect to the accurate recording.

The implementation of CMR was previously presented in the tool demonstration paper [13]. Yet the previous paper never presented information on its evaluation. In this paper, we additionally provide evaluation results based on our experiment and discuss factors that affect its effectiveness and usefulness.

The paper includes the following contributions:

- It proposes a publicly available tool called CMR, which records eleven kinds of change macros.
- It presents detailed information on how CMR achieves the accurate recording of textual changes.
- It demonstrates that CMR is superior to the existing tools that directly record textual changes with an experiment consisting of nine programming tasks.

The development of CMR continues. The source code of the latest version of CMR and sample programs using it is available from the web site[†].

The remainder of the paper is organized as follows. In Sect. 2, we give an example of source code changes and explain a shortcoming of existing tools that record them. In Sect. 3, we present a running implementation of CMR, which provides an improved mechanism that overcomes the shortcoming of the existing tools. In Sect. 4, we assess the effectiveness of CMR by showing experimental results. In Sect. 5, we discuss factors that affect the effectiveness and usefulness of CMR. In Sect. 6, we present related work. Finally, we conclude with a summary and future work in Sect. 7.

2. Motivation for Improvement

To explain concrete a shortcoming that cause the dissatisfaction on the accuracy of recorded textual changes, we take an example of textual changes that were actually performed on source code. In Fig. 1, a triangle with the symbol c_1, c_2, or c_3 denotes an intentional source code change involving several textual changes. A programmer first opened the file `P.java` declaring the class `P` and inserted the text string "`int efg = 0;`" into the body of the method `abc()`. This insertion was recorded as c_1. Then, she changed the name of `abc()` into `xyz` by applying an automated Rename Method refactoring. Actually, she activated this refactoring under the selection of the string `abc` and directly updated it to the

[†]http://www.jtool.org/cmr/

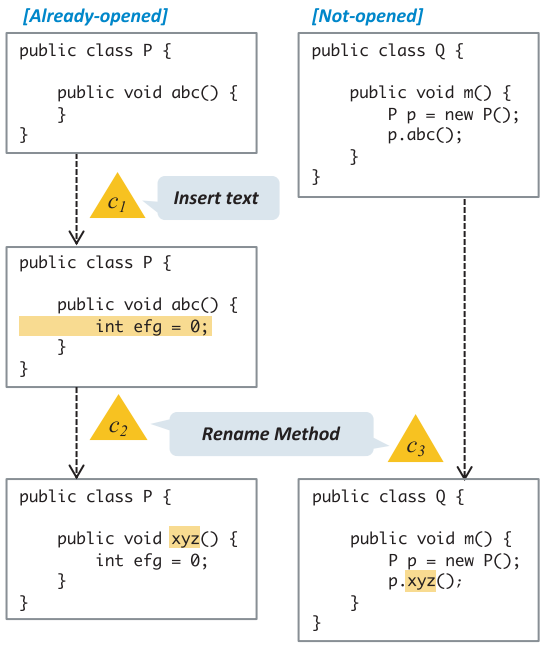

Fig. 1 Textual changes performed on the source code.

string `xyz` on the source code text. This replacement was recorded as c_2. At the same, 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()`. This update on the file `Q.java` corresponds to c_3.

OperationRecorder (called OR hereafter) obtains edit operations from the undo history of Eclipse. This history exists in order to undo past textual edits performed on already-opened files under the control of the editor, but ignore ones in not-opened files. Fluorite (called FLT hereafter) logs document-change events by using built-in document listeners of Eclipse. These listeners capture events occurring in files that have been already opened (to be precise, is currently activated) on the editor, but exclude ones in not-opened files. As a result, in Fig. 1, neither OR nor FLT could capture c_3 silently occurring in `Q.java` since it was not opened at the execution of the Rename Method refactoring.

We emphasize that the deficiency of textual changes appearing in c_3 clearly hampers exact understanding of source code evolution. For example, if source code changes of a subsequent refactoring override c_3, the actual textual changes cannot be restored. This happens frequently since several refactoring transformations tend to update the contents of multiple files at the same time no matter whether some of them are already opened or not. To record accurate textual changes, change-recording tools must support the recording of textual changes performed on not-opened files.

Here, we cannot give enough information on the capability of CodingTracker since its implementation is not available to download. According to [4], CodingTracker does not probably output textual changes with respect to c_3 in real time during the recording process although it can capture events for resource changes performed on the outside of

Table 1 Change macros (nine atomic macros and two compound-related macros).

Change macro	Description
DocumentMacro	Textual changes directly performed on source code text
CancelMacro	Cancellation of the textual changes in previous DocumentMacro
CopyMacro	Execution of an action that copies text string
CommandMacro	Execution of a command service
CodeCompletionMacro	Execution of code completion by content assist
RefactoringMacro	Execution of refactoring
GitMacro	Execution of a Git event
ResourceMacro	Changes of the properties or the contents of a resource
FileMacro	Execution of file operations
TriggerMacro	Timing when an action provoking compounded textual changes is started, ended, or canceled, or timing of the change of the cursor location
CompoundMacro	Group of change macros

its IDE.

3. Implementation

In this section, we first describe change macros recorded by CMR and then explain improvements made in its implementation to overcome the shortcoming of OR and FLT.

Here, we do not have a proof that textual changes (not AST changes) are suitable for the representation of fine-grained source code changes. Meanwhile, textual changes are convenient for chronologically replaying how the source code was written and modified since it is easy to restore its snapshots from recorded textual changes. Moreover, text is free from the successful build of the AST or other models. If needed, changes of source code entities (classes etc.) or AST nodes can be later inferred from the textual changes [4], [14]. In other words, textual changes are versatile enough to be manipulated in various kinds of tools that support source code evolution. For these reasons, CMR embraces the direct recording of textual changes as well as both OR and FLT.

CMR limits its consideration to textual changes of Java source files and actions related to them. Hence, for example, it does not record textual changes in configuration files or build artifact files, a command action that runs a binary program. Additionally, it does not handle the update of Java source files that are performed outside of Eclipse.

3.1 Change Macros

To capture textual changes to be recorded and detect the programmer's actions related to them, CMR employs seven listeners (IDocumentListener, IResourceChangeListener, etc.) embedded in Eclipse. The captured textual changes and detected actions constitute eleven kinds of change macros: nine atomic macros and two special macros that are used for compounding atomic macros. Table 1 lists these change macros.

The first two of the atomic macros directly reify the 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 these textual changes. Here, in the execution of refactoring under the inline mode, a programmer directly edits source code on the editor instead of inputting a changed text string in the dialog outside the editor. Several change recording tools including CMR record this textual change as the normal one although it might be volatile, since they capture every textual change made on the editor and cannot know whether a recorded textual change is volatile or not just when recording it. A volatile textual change is also recorded during code completion since Eclipse inserts a completion-candidate string although it is immediately deleted. Unfortunately, the recording tools cannot also distinguish a textual change made by a programmer from one derived from code completion just when recording the change. A volatile textual change that appears due to the implementation of Eclipse is expected to be canceled. To remove such a volatile textual change later, CMR introduces **CancelMacro** that is responsible for canceling a textual change that appears in its previous DocumentMacro.

The remaining seven atomic macros denote the occurrences of the programmer's actions that might relate to textual changes of source code. **CopyMacro** stores a copied text string 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 a command service (including cut, copy, and paste actions), a code completion action by content assist, an automated refactoring, and an event operating source files stored in a Git repository (the "refs changing" and "index changing" events), respectively. **ResourceMacro** represents the changes of the properties (name and location etc.) or the contents of a resources (a file, a packages, or a 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 actions for files, and Git actions for files.

The one of the two compound-related macros is **TriggerMacro** having the four timing indicators BEGIN, END, CANCEL, and INSTANT. The former three indicate triggers to begin, end, and cancel pre-defined actions (refactoring, code completion, undoing, redoing, and Git command) that might cause textual changes made at the same time. INSTANT indicates the occurrence of an instantaneous action such as a change of the cursor location. It provides a hint of the com-

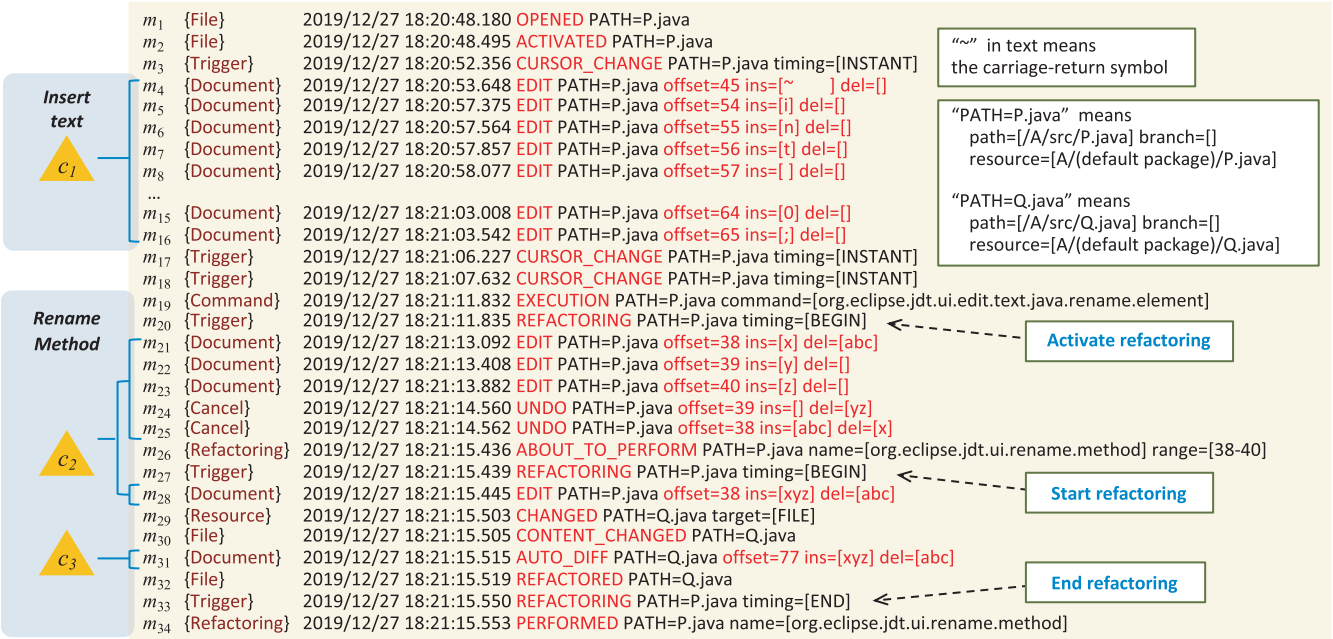

Fig. 2 Series of change macros actually recorded by CMR.

pletion or interruption of running actions and programmer's editing activities. A begin-end pair of TriggerMacro derives **CompoundMacro**, which groups atomic macros caused by the same programmer's action.

3.2 Accurate Recording of Textual Changes

Figure 2 shows change macros that CMR has actually recorded in the source code evolution shown in Fig. 1. For example, m_1, m_4, and m_{26} indicate FileMacro, DocumentMacro, and RefactoringMacro, respectively. Here, m_i~m_j denotes every macro sandwiched between m_i and m_j. The m_4~m_{16}, m_{21}~m_{25} plus m_{28}, and m_{31} correspond to the code changes c_1, c_2, and c_3 shown in Fig. 1, respectively. All of the eleven 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. DocumentMacro also stores an inserted text string (ins), a deleted text string (del), and the offset of the leftmost character (offset) of the inserted and/or deleted text string in the source code.

DocumentMacro has totally eight kinds of actions. Six of them are labeled with EDIT, CUT, PASTE, UNDO, REDO, and COMPLETE, which simply indicate editing, cutting, pasting, undoing, redoing, and code completion, respectively. Whereas the six actions are all usual in normal code editing, the two remaining actions labeled with AUTO_DIFF and IRREGULAR_DIFF attain the accurate recording without deficiency of textual changes.

We consider that existing approaches do not address two possible situations where deficient recording arises. The first situation 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, as mentioned in Sect. 2. In this situation, the existing tools often overlook indirect textual changes in not-opened files. To overcome this fault, CMR obtains the previous snapshot of a not-opened file by using the local history feature built in Eclipse and checks if its content is updated during the execution of a programmer's action. If any update occurs, it calculates textual differences between the contents of the file before and after the execution in real time by using *diff* utility. Each of the differences is transformed into either an inserted text string and/or a deleted one. CMR automatically records DocumentMacro with AUTO_DIFF, which stores such inserted and detected texts and their offsets. In Fig. 2, m_{31} corresponds to this macro since the file Q.java was not opened at the execution of the applied Rename Method refactoring.

The other situation is that change-recording tools sometimes disregard textual changes due to the limitations of their recording implementations. Although 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 as far as possible in this situation, CMR temporarily generates source code 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 every time it records each DocumentMacro. If there is any irregular discrepancy detected, CMR automatically records DocumentMacro with IRREGULAR_DIFF, which contains textual differences to reconcile the discrepancy.

3.3 Making Change Macros Concise

A concise representation of textual changes is convenient

Fig. 3 Series of post-processed macros produced by CMR.

for human to understand source code evolution. Moreover, textual changes bundled based on the programmer's actions promote fast and easy understanding. To this end, CMR post-processes raw macros through three steps: combining textual changes, canceling verbose textual changes, and bundling macros. The combination and cancellation steps make macros concise, and the bundling step makes them grouped. Figure 3 shows a series of macros that were obtained by applying the post-processing to the macros shown in Fig. 2. Whereas the symbol m indicates one of the raw macros, m', which is with the prime mark, indicates one of the post-processed macros. Two macros with the same index number such as m_x and m'_x store the same information. The m'_{x-y} denotes a macro generated through the post-process.

Programmers and researchers seldom prefer the undue separation of textual changes. In Fig. 1, the consecutively typed characters "`i`", "`n`", and "`t`" were separated in the naive recording for c_1, although the text string "`int`" is usually a keyword in Java source code. In most cases, each character constituting an identifier or a keyword is too fine-grained to be considered based on programmers' intuitions. Thus, most of existing change-recording tools employ combination of consecutive textual changes.

To combine textual changes that are stored in different DocumentMacro, CMR employs as default a delimiter-based combination strategy. This strategy might be understandable and can be easily implemented. It concatenates consecutive two texts if they contain no pre-defined delimiter. The default delimiters are all characters appearing in the string "␣`\t\n\r,.;()[]{}`" (␣ means a space character). Looking at m_5~m_8 shown in Fig. 2, neither m_5, m_6, nor m_7 stored one of the delimiters. Therefore, all textual changes stored in these macros were combined (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 m_8 stored the blank character that is not contained in the delimiters. As a result, CMR produced DocumentMacro m'_{5-7} storing the inserted text string "`int`" and m'_8 storing the blank character. In general, it is hard to pre-define suitable delimiters since they depend on the usage of recorded macros. Hence, CMR allows users to freely customize the delimiters and to replace the prepared delimiter-based strategy with their own combination ones (e.g., time-period based combination of textual changes).

CancelMacro appears only within a begin-end or begin-cancel pair of TriggerMacro. To find DocumentMacro corresponding to CancelMacro, CMR repeats to compare inserted and deleted text strings of CancelMacro with those of each DocumentMacro enclosed by the TriggerMacro pair. This comparison is performed in the inverse order of when macros are recorded. CancelMacro itself and its corresponding DocumentMacro are removed together. In Fig. 2, for example, m_{24} and m_{25} canceled m_{21}, m_{22}, and m_{23}, and thus these five macros removed. A cancellation step is feasible in most cases but sometimes fails in which DocumentMacro corresponding to CancelMacro is not found. In this failure case, both of them remain as-is.

In Fig. 1, all textual changes involved in c_2 and c_3 should be correlated with the applied Rename Method refactoring. Bundling macros related to each other helps programmers and researchers to know which programmer's action triggers which textual changes. To this end, CMR identifies a begin-end pair of TriggerMacro in real time and then produces CompoundMacro that bundles macros enclosed by the TriggerMacro pair. In Fig. 2, m_{20} and m_{33} composed a begin-end pair of the applied refactoring, and raw macros sandwiched between them are bundled. Consequently, CompoundMacro m'_{20} enclosed six macros (m'_{26}, m'_{28}, m'_{29}, m'_{30}, m'_{31}, and m'_{32}) shown in Fig. 3. Through this bundling step, a series of post-processed macros excludes TriggerMacro. Note that the occurrences of TriggerMacro are not necessarily paired. In the example shown in Fig. 2, m_{27} and m_{33} became unpaired since CMR skipped m_{27} to detect the outermost begin-end. If it detects a begin-cancel pair of TriggerMacro, it aborts a running bundling-process and converts macros enclosed by TriggerMacro into unenclosed ones. A similar abortion is performed when any code is automatically completed in the Eclipse's content assist.

3.4 Usage

CMR is designed to be embedded by users (researchers and developers) who enrich programming support tools that leverage fine-grained source code changes. For this purpose, CMR adopts the event-listener model to notify the developed tools of change macros immediately after it captures textual edits or detects actions related to those edits. To receive macros recorded by CMR and stop the receiving, a user program registers or unregisters a receiver instance implementing only the two methods of the given interface: one for receiving recorded macros as-is and the other for receiving post-processed ones. Each receiver instance holds its own delimiters that are used when combining consecutive text strings in the post-process.

In our research projects, two representative application tools embed CMR, whose source code are all publicly available. ChangeTracker[†] is almost compatible with OR, which receives post-processed macros and converts them into six kinds of edit operations of fine-grained textual changes. Operation data are stored in a repository for each project once changed source code files are saved, closed, or deleted. The other application is a postponable refactoring tool [15][††], 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 receives every raw macro from CMR to monitor in real time code fragments that might be affected by the suspended refactoring.

4. Evaluation

To evaluate the effectiveness of CMR from the viewpoint of the accuracy of recorded textual changes, we address the following two questions:

Q1 What macros and how many macros can CMR record?
Q2 Can CMR record macros without excess and deficiency of textual changes made on source code?

To answer Q1 and Q2, we conducted an experiment in which textual changes are actually recorded while evolving source code.

4.1 Experiment

In our experiment, one of the authors (called a programmer in this experiment) had changed an existing Java program for a puzzle game (Tetris). Table 2 shows the lines of code for the initial and final versions of the program, which consist of twelve and thirteen source files, respectively. The purpose of the source code changes is to eliminate code smells in the target program though the application of several refactorings and small revisions. The programmer explored the whole of its source code to detect smells such as inelegant package and class structures, ungainly names, and inefficient algorithms. Then, he tried to fix the disturbing source code entities that contain some of the detected smells. The source code changes finally preserved the external behavior of the initial source code.

Table 2 LOC of the Java program used in the experiment.

Source file	Initial	Final
`Block.java`	172	177
`BlueBlock.java`	16	19
`CyanBlock.java`	14	15
`GreenBlock.java`	16	19
`MagentaBlock.java`	16	19
`OrangeBlock.java`	16	19
`RedBlock.java`	16	19
`YellowBlock.java`	16	19
`GameInfo.java`	93	96
`Pit.java`	177	185
`Tetris.java`	121	9
`Tile.java`	70	75
`GameFrame.java`	–	125
Total	743	796

The changes are performed through the following nine tasks.

- T1 Renamed the method `setPosXY()` of the class Block to `setPos()` through an automated Rename Method refactoring under the inline mode within the file `Block.java`. This refactoring affected a not-opened file `Pit.java` in addition to the opened file `Block.java`.
- T2 Created two packages block and main and moved all the twelve classes into either of them by using drag-and-drop operations of Java files defining the moved classes. Whereas `Block.java` was already opened, the remaining eleven files were not opened.
- T3 Edited code within `Tile.java` and `Tetris.java` to remove emergent errors due to the file moving in the task T2. The changes were enclosed in these two edited files that were opened.
- T4 Activated the quick fix feature of Eclipse to remove a trivial warning appearing within `Tetris.java`. This action was performed by activating the Eclipse's content assist. The change was enclosed in the fixed file that was opened.
- T5 Revised code within `Tile.java` and `Pit.java` to free no longer used objects from memory. This task involved undo, copy, and paste actions. The Eclipse's code completion was also performed in this task.
- T6 Renamed the class Tetris to GameFrame through an automated Rename Class refactoring. As a result, the file-name of `Tetris.java` was changed to `GameFrame.java`. This refactoring updated the contents of `Tetris.java`, `GameFrame.java`, `Block.java`, and `Pit.java`, all of which were opened.
- T7 Created a new class Tetris through an automated Create Class refactoring. Eclipse automatically created `Tetris.java` defining Tetris and opened it. The default code was inserted into `Tetris.java`.

[†]http://www.jtool.org/ChangeTracker2/
[††]http://www.jtool.org/PostRefactor/

T8 Moved the method `main()` of GameFrame into Tetris by activating an automated Move Class refactoring. The changes were enclosed in `GameFrame.java` and `Tetris.java`, both of which were opened.
T9 Extracted a new method `action()` from the method `keyPressed()` of GameFrame through an automated Extract Method refactoring. `GameFrame.java` defining GameFrame was opened during the application of this refactoring.

All the tasks were performed with the Eclipse 2019–12 for Enterprise Java Developers on a MacBook Pro (3.1 GHz Intel Core i7 and 16 GB RAM), running MacOS 10.14.3. We selected these nine tasks so that they involved the programmer's actions as various as possible since the diversity of the tasks was important in this experiment. Moreover, we considered that more likely actions [16] were suitable for assessing the effectiveness of change-recording tools. Based on this consideration, the tasks aggressively involved several popular refactorings (Rename Method, Rename Class, Move Method, Move Class, and Extract Method) [17], in addition to the programmer's usual edits. The quick fix and code completion actions are a bit minor but are preferable for programmers who use IDEs. Additionally, no Git action was included in the tasks since neither OR nor FLT supports this action.

We dared not include actions undoing a refactoring in the nine tasks since Eclipse employing OR did not allow the programmer to undo the refactorings applied in this experiment. However, the recording results using CMR and FLT for the refactoring undo might be curious. Hence, we made an additional experiment with the tasks including five refactorings (Rename Method, Move Class, Rename Class, Move Method, and Extract Method) and their undoing. As a result, CMR accurately recorded all textual changes with no excess or deficiency in all the five undo tasks, whereas FLT has deficiency in the three undo tasks.

To compare outcomes using CMR with those using currently available recording tools, we obtained the implementations of OR[†] and FLT[††]. For already-opened files, CodingTracker perhaps has the same capability as CMR does. However, we did not employ it since its implementation was not available. In the experiment, we first installed CMR and FLT on the same Eclipse as its plug-ins, and collected data for the nine tasks. We next prepared another Eclipse on which only OR was installed and collected data for the same tasks. The reason why we used the two variants of Eclipse is that the implementation of OR seems to be tricky workaround. It intentionally inserts a space character at the beginning of the edited file and immediately deletes the inserted character to forcibly change the dirty status of the file. CMR and FLT obediently collect these nonsense textual changes and then record false macros storing them. OR eliminates such false operations in its recording process since it can know which textual changes were inserted by itself. By separating recording processes using CMR and FLT from that using OR, the truthful recording is achieved. Meanwhile, the programmer had to perform perfectly the same edits and actions twice. Thus, we captured a screencast at the first recording using CMR and FLT. At the next recording, we replicated the edits and actions according to the screencast.

4.2 Assessment Criterion

The kinds and granularity of macros[†††] recorded by the three tools differ each other. For example, neither OR nor FLT have a particular macro corresponding to a resource change whereas CMR has such a macro. Additionally, the granularity levels of recorded textual changes do not conform in the three tools. For example, the same textual change of inserting the text string "`a;`" might be recorded as a single macro storing the whole text string "`a;`" or divided into multiple macros "`a`" and "`;`". These facts denote that simply comparing the total number of recorded macros never evinces the accuracy of them. Thus, we must define an assessment criterion for counting macros, which is not influenced by the differences of the change representations among the three tools.

To this end, we excluded macros representing non-textual changes when determining the excess or deficiency of textual changes. Additionally, we assume that the three tools certainly store a single textual change as a single macro that depends on their respective granularity levels of textual changes to be recorded. In other words, the ideal number of macros recorded in each task is equals to the number of individual textual changes that the tool must recognize in the task. If there is excess of textual changes in recording, an extra macro is recorded. On the other hand, if there is deficiency of textual changes in recording, its corresponding macro is not recorded. Based on this idea, we assess the accuracy of recording by counting macros storing individual textual changes. Such macros are classified into three types: adequate, extra, and deficient.

An *adequate* macro stores a textual change actually performed by a programmer or through IDE features. To check whether a certain macro is adequate or not, we restored the contents of the source code before and after the macro was recorded, and examined a textual difference between the contents. If the difference conforms textual changes that are observed on source code being edited, the macro is deemed adequate. Here, bordering textual changes having the same change type (insertion, deletion, or replacement) or ones having different change types but the same offset value can be freely unified into a single macro. Therefore, the numbers of adequate macros recorded by the respective tools in the same task differ based on their design policies or implementation decisions. A macro becomes *extra* if it unfortunately stores a textual change that cannot

[†]http://www.ritsumei.ac.jp/~tomori/operec.html
[††]http://www.cs.cmu.edu/~fluorite/

[†††]In this experiment, macros in CMR, operations in OR, and events in FLT are collectively called macros.

be observed on the edited source code or a volatile textual change that remains due to the failure or the absence of its cancellation. A *deficient* macro is not actually recorded but must exist to store a textual change that can be observed on the edited source code.

Here, it is trivial to count adequate and extra macros since they are explicitly recorded. More specifically, to count adequate and extra macros, we first classify each of the recorded macros as either adequate or extra, and then count adequate and extra ones for each tool. On the other hand, counting deficient macros is impossible since there is no macro that stores a textual change that is disregarded in the recording. Thus, we quit counting deficient macros directly. Instead, we tried to extract isolated chunks from disregarded textual changes. In general, almost all change-recording tools including CMR, OR, and FLT do not allow a single macro to store separated text strings (having different offset values) as the inserted or deleted text. In other words, a single macro never contains textual changes that update different files or alter non-adjacent text fragments within the same file. Moreover, the tools should not merge textual changes in case that code fragments in them overlap or are derived from multiple actions performed separately. Such textual changes should be stored into separate macros. From the above, to count deficient macros, we first identified all disregarded textual changes in the recording by the observation of edited source code, and then divided them into multiple chunks that are individually stored into separate deficient macros. This way makes it possible to count deficient macros.

4.3 Experimental Results

Table 3 summarizes the numbers of macros (macros, operations, or events) recorded by CMR, OR, and FLT under the respective tasks. The CMR_r and CMR_p columns indicate the number of raw macros and post-processed ones, respectively. In the OR column, operations tagged with `<normalOperation>`, `<fileOperation>`, `<menuOperation>`, and `<compoundOperation>` correspond to DocumentMacro, FileMacro, CommandMacro, and CompoundMacro, respectively. In the FLT column, an event tagged with `<DocumentChange>` corresponds to DocumentMacro. Events tagged with `<Command>` are divided into a normal command corresponding to CommandMacro or a file open operation corresponding to FileMacro. CompoundMacro never appears in FLT. Both OR and FLT have no operations and events that explicitly correspond to CancelMacro, RefactoringMacro, ResourceMacro, and TriggerMacro. The macros in green boxes represent textual changes (insertion, deletion, and replacement of text) that directly update the contents of source code. In this experiment, there was no DocumentMacro labeled with IRREGULAR_DIFF, which automatically resolves the restoration failure.

Through an manual examination of recorded macros, restored snapshots, and actual edits, based on the assessment criterion described in Sect. 4.2, we determined adequate, extra, and deficient macros for all the nine tasks. For example, Fig. 4 shows detailed information on the examination result for the task T1. The correspondence relationships between macros were found according to the action types, the contents (the type, the inserted or deleted text, and their offset

Table 3 Numbers of recorded macros.

Task	Macro	CMR_r	CMR_p	OR	FLT
T1: Rename Method	Document	4	3	1	3
	Cancel	1	1	–	–
	File	5	5	3	1
	Command	1	1	1	5
	Refactoring	2	2	–	–
	Resource	1	1	–	–
	Trigger	7	–	–	–
	Compound	–	1	1	–
T2: Move Class	Document	34	34	8	6
	File	43	43	2	0
	Command	0	0	0	2
	Refactoring	4	4	–	–
	Resource	35	35	–	–
	Trigger	4	–	–	–
	Compound	–	2	2	–
T3: Edit Code	Document	19	6	5	3
	File	7	7	7	3
	Command	2	2	2	11
	Trigger	3	–	–	–
T4: Quick Fix	Document	8	8	8	5
	File	1	1	3	0
	Command	2	2	2	6
	Trigger	1	–	–	–
	Compound	–	0	1	–
T5: Revise Code	Document	80	41	16	16
	Copy	1	1	1	1
	File	6	6	6	3
	Command	5	5	5	30
	CodeCompletion	2	2	–	–
	Trigger	11	–	–	–
	Compound	–	2	2	–
T6: Rename Class	Document	36	7	6	56
	Cancel	27	0	–	–
	File	7	7	2	1
	Command	1	1	1	7
	Refactoring	2	2	–	–
	Resource	3	3	–	–
	Trigger	5	–	–	–
	Compound	–	1	1	–
T7: Create Class	Document	1	1	0	0
	File	3	3	2	1
	Command	0	0	0	1
	Resource	1	1	0	–
	Trigger	0	–	–	–
T8: Move Method	Document	5	5	5	2
	File	5	5	5	3
	Command	2	2	3	6
	Refactoring	2	2	–	–
	Resource	0	0	–	–
	Trigger	8	–	–	–
	Compound	–	1	1	–
T9: Extract Method	Document	7	7	7	7
	File	3	3	3	0
	Command	2	2	2	7
	Refactoring	2	2	–	–
	Resource	0	0	–	–
	Trigger	9	–	–	–
	Compound	–	1	2	–
	Total	420	271	116	186

	File	CMR$_r$	CMR$_p$	OR	FLT
1	Block.java	{File} OPENED	{File} OPENED	{File} OPEN	{Command} FileOpen
2	Block.java	{File} ACTIVATED	{File} ACTIVATED	{File} ACT	{Command} MoveCaret
3	Block.java	{Trigger} CURSOR_CHANGE [INSTANT]		{File} ACT	{Command} SelectText
4	Block.java	{Trigger} CURSOR_CHANGE [INSTANT]			
5	Block.java	{Command} EXECUTION (Rename)	{Command} EXECUTION (Rename)	{Menu} Rename	{Command} Rename
6	Block.java	{Trigger} REFACTORING [BEGIN]			
7	Block.java	{Trigger} CURSOR_CHANGE [INSTANT]			
8	Block.java	{Trigger} REFACTORING [CANCEL]			
9	Block.java	{Doc} EDIT ins=[] del=[Y]	*Extra* {Doc} EDIT ins=[] del=[XY]		*Extra* {Doc} Delete del=[XY]
10	Block.java	{Doc} EDIT ins=[] del=[X]			{Command} DELETE_PREVIOUS
11	Block.java				{Command} InsertString
12	Block.java	{Cancel} UNDO ins=[XY] del=[]	*Extra* {Cancel} UNDO ins=[XY] del=[]		*Extra* {Doc} Insert ins=[XY]
13	Block.java	{Refactoring} ABOUT_TO_PERFORM	{Refactoring} ABOUT_TO_PERFORM		
14	Block.java	{Trigger} REFACTORING [BEGIN]	{Compound} REFACTORING	{Compound} Rename Method	
15	Block.java	{Doc} EDIT ins=[setPos] del=[setPosXY]	!{Doc} EDIT ins=[setPos] del=[setPosXY]	!{Normal} ins=[setPos] del=[setPosXY]	{Doc} Replace ins=[] del=[XY]
16	Block.java	{File} REFACTORED	!{File} REFACTORED		
17	Pit.java	{Resource} CHANGED	!{Resource} CHANGED		
18	Pit.java	{File} CONTENT_CHANGED	!{File} CONTENT_CHANGED		
19	Pit.java	{Doc} AUTO_DIFF ins=[] del=[XY]	!{Doc} AUTO_DIFF ins=[] del=[XY]	*Deficient*	*Deficient*
20	Pit.java	{File} REFACTORED	!{File} REFACTORED		
21	Block.java	{Trigger} REFACTORING [END]			
22	Block.java	{Refactoring} PERFORMED	{Refactoring} PERFORMED		

Fig. 4 Macros recorded in the task T1 (including the automated Rename Method refactoring).

values), and the time order of respective macros. The experimental data, screencasts, examination results are available from the web site[†].

4.3.1 Characteristics of Recorded Macros

Through our exploratory examination of macros actually recorded, we found two notable points. Looking at the numbers shown in Table 3, OR and FLT do not provide macros corresponding to RefactoringMacro and CodeCompletionMacro whereas CMR does. To be precise, OR and FLT can record macros corresponding to refactoring if it is performed through an explicitly executed command. However, several refactorings are usually performed without executing commands. Unfortunately, OR and FLT never record macros corresponding to such refactorings. In fact, they recorded no macro corresponding to the Move Class refactoring in the task T2. Moreover, in the task T5, OR captured a textual change on code completion as a simple insertion of the text and recorded a macro storing the inserted text whereas CMR and FLT recorded macros corresponding to the activated code completion. CMR also correlated a macro storing the inserted text with CodeCompletionMacro.

Looking at Table 3 again, CMR or OR recorded compound macros in the six tasks or seven ones, whereas FLT never recorded such macros. For example, in the task T1 (see Fig. 4) using CMR, the two kinds of TriggerMacro labeled with `REFACTORING [BEGIN]` and `REFACTORING [END]` (at the lines 14 and 21 in the column CMR$_r$, respectively) constituted CompoundMacro (at the line 14 in the column CMR$_p$) that encloses its following six macros. The group represented by this compound macro helps programmers and researchers understand that the two textual changes (at the lines 15 and 19) were derived from the Rename Method refactoring but the other textual changes (if existed) were not related to this refactoring. A similar compound macro (at the line 14) was recorded in the task T1 using OR although it did not capture a textual change within the not-opened file. It was almost impossible to determine which macros were related to the applied refactoring by observing macros recorded by FLT.

Answer to Q1. CMR overall tends to record a greater number of macros than OR and FLT. Especially, it can record beneficial macros related to refactoring and resource changes, which might implicitly involve textual changes. In addition, CMR (and OR) is capable of producing a compound macro that reveals the correlation of the recorded textual changes and actions involving them, unlike FLT.

4.3.2 Accuracy of Recorded Macros

Table 4 summarizes the comparative results of the numbers of adequate (*ad*), extra (*et*), and deficient (*dt*) macros. In each of the tasks, the number of recorded macros actually recorded is calculated by $ad + et$, and the ideal number of them is calculated by $ad + dt$. Cells colored with blue indicate the achievement of the accurate recording. Although the numbers of macros in the same task differs among the three tools, they performed the accurate recording in the three tasks T3, T4, and T9. CMR unfortunately failed the accurate recording for the opened file in the task T1 and T6, and for not-opened files in the task T2. Limiting to textual changes on the opened files, OR failed in the four tasks T2, T5, T6, and T7. FLT failed in the five tasks T1, T2, T6, T7, and T8. Of course, both OR and FLT recorded no macro for the not-opened files in the tasks T1, T2, and T7.

[†]http://www.jtool.org/cmr/exp-data/

Table 4 The numbers of adequate (*ad*), extra (*et*), and deficient (*dt*) macros.

Task	CMR (= CMR_p)						OR						FLT					
	opened			*not-opened*			*opened*			*not-opened*			*opened*			*not-opened*		
	ad	*et*	*dt*	*ad*	*et*	*dt*	*ad*	*et*	*dt*	*ad*	*et*	*dt*	*ad*	*et*	*dt*	*ad*	*et*	*dt*
T1: Rename Method	1	2	0	1	0	0	1	0	0	0	0	1	1	2	0	0	0	1
T2: Move Class	14	0	0	20	0	6	7	1	2	0	0	26	6	0	2	0	0	26
T3: Edit Code	6	0	0	–	–	–	5	0	0	–	–	–	3	0	0	–	–	–
T4: Quick fix	8	0	0	–	–	–	8	0	0	–	–	–	5	0	0	–	–	–
T5: Revise Code	41	0	0	–	–	–	16	0	2	–	–	–	16	0	0	–	–	–
T6: Rename Class	6	0	2	1	0	0	5	1	3	0	0	1	2	54	6	0	0	1
T7: Create Class	1	0	0	–	–	–	0	0	1	–	–	–	0	0	1	–	–	–
T8: Move Method	5	0	0	–	–	–	5	0	0	–	–	–	2	0	1	–	–	–
T9: Extract Method	7	0	0	–	–	–	7	0	0	–	–	–	7	0	0	–	–	–
Success rate	7/9			2/3			5/9			0/3			4/9			0/3		

In the task T1, CMR recorded two extra macros (at the lines 9 and 12 shown in Fig. 4) in the open file Block.java. These macros were derived from the programmer's inline inputting of a new name and IDE's canceling of the inputting in the application of the Rename Method refactoring. Essentially, volatile textual changes stored in them should be made outside the edited source code and should not appear on the source code as mentioned in Sect. 3.1. CMR is designed to cancel macros storing such volatile textual changes, but failed this cancellation in the task T1 unfortunately. In contrast, it successfully removed 54 macros that stored volatile textual changes and canceled them in the task T6. In this experiment, the extra macros containing these volatile textual changes did not derive the inconsistent restoration of snapshots of all changed files. We will discuss the underlying cause of this cancellation failure in Sect. 5.1.1.

Based on the implementation decision of using the undo history in OR, it never records volatile textual changes. Moreover, in the tasks T2 and T6, two extra macros represent false operations due to its tricky workaround implementation that inserts an empty character at the beginning of the edited file. FLT always captures volatile textual changes and leaves them. It recorded 2 and 54 extra macros storing volatile textual changes in the tasks T1 and T6, respectively.

Unfortunately, two deficient macros were identified for the opened files in the task T6 using CMR although raw macros corresponding to these two macros were recorded correctly. In this task, the Rename Class refactoring involved the rename of a file containing the renamed class. The deficiency might result from an adverse fact file renaming vanished the original file before the post-processing had been finished. Moreover, there were deficient macros for the not-opened files in the task T2 using CMR. The Move Class refactoring attached a declaration of the package containing the moved class and inserted import declarations to access classes not existing in its packages. Unfortunately, CMR overlooked textual changes related to the import declarations in six of the not-opened files although it exploited the local history of these files. In this experiment, the deficient macros did not derive the inconsistent restoration since each FileMacro rescued the whole contents of the respective changed files. However, the existence of the deficient macros decreased the accuracy of the recording. We will discuss a probable cause of this deficiency in Sect. 5.1.2.

Both OR and FLT disregarded textual changes made within opened files in several tasks, in addition to those made within not-opened files. In general, the Move Class refactoring or the Rename Class refactoring embarrasses OR and FLT since these refactorings involve the process that both deletes an existing file and creates a new file. In this process, textual changes are performed on the new file before its opening. The failure of recording such textual changes was due to policies in which both the tools do not properly manage resource changes (the move or rename of files in this experiment). To make matters worse, during the application of the Move Method refactoring in the task T8, FLT discarded textual changes within the file defining the source class of the moved method since it keeps track of textual changes in only the activated files that the programmer is just editing. To complement disregarded textual changes within opened files, OR and FLT freshly read the whole contents of the changed files once they are opened or activated. However, they cannot record textual changes overlapping during the time period between two successive file operations.

A different kind of deficiency emerged using OR. In the task T5, the programmer inserted a text and immediately undid it, and all the three tools detected the execution of the undo action. Nevertheless, OR recorded no macro corresponding to the insertion and deletion of text since it uses the undo history of Eclipse, whereas both CMR and FLT stored a textual change undone later and one undoing it into two separate macros. The existence of these macros is useful for analyzing and leveraging textual changes since doing and undoing a textual change sometimes denotes trial-and-error processes, stagnation in programming, and frequently occurred mistakes.

The numbers of macros shown in Table 4 indicate to what extent the tools achieved the accurate recording. However, they are unsuitable for a direct comparison of the accuracy of recorded macros since the ideal numbers of macros that they should record in the same task differ each other. To do fair comparison, the number of textual changes to be recorded must be equalize. Meanwhile, equalization without considering their granularity levels of recorded textual changes is useless. If the baseline granularity of textual changes is larger than the granularity of macros recorded by

Table 5 The numbers of adequate (*ad*), extra (*et*), and deficient (*dt*) characters in the changed text.

Task	CMR (= CMR_p)						OR						FLT					
	opened			*not-opened*			*opened*			*not-opened*			*opened*			*not-opened*		
	ad	*et*	*dt*	*ad*	*et*	*dt*	*ad*	*et*	*dt*	*ad*	*et*	*dt*	*ad*	*et*	*dt*	*ad*	*et*	*dt*
T1: Rename Method	14	4	0	2	0	0	14	0	0	0	0	2	2	4	12	0	0	2
T2: Move Class	153	0	0	255	0	84	97	0	56	0	0	339	41	0	112	0	0	339
T3: Edit Code	37	0	0	–	–	–	37	0	0	–	–	–	33	0	4	–	–	–
T4: Quick fix	101	0	0	–	–	–	101	0	0	–	–	–	101	0	0	–	–	–
T5: Revise Code	218	0	0	–	–	–	208	0	10	–	–	–	196	0	22	–	–	–
T6: Rename Class	90	0	30	15	0	0	75	0	45	0	0	15	30	90	90	0	0	15
T7: Create Class	40	0	0	–	–	–	0	0	40	–	–	–	0	0	40	–	–	–
T8: Move Method	243	0	0	–	–	–	243	0	0	–	–	–	124	0	119	–	–	–
T9: Extract Method	1,173	0	0	–	–	–	1,173	0	0	–	–	–	1,173	0	0	–	–	–
Total	2,069	4	30	272	0	84	1,948	0	151	0	0	356	1,700	94	399	0	0	356
Precision	0.998			1.000			1.000			N/A			0.948			N/A		
Recall	0.986			0.764			0.928			0.0			0.810			0.0		
F-measure	0.992			0.866			0.963			N/A			0.873			N/A		

the tools, they often produce multiple macros that partially store textual changes with the baseline granularity level. For these cases, we cannot judge whether these macros are adequate or not with respect to the accurate recording.

To avoid this problem, we decided to define the baseline granularity by a character, which has the finest granularity level of changed source code text and can be no longer divided. Consequently, this decision enables a systematic way of counting the number of characters that should be stored in macros and are actually stored. Based on the decision, we picked up characters appearing in the changed source code text and checked if each of these characters was stored in macros recorded by the tools.

Table 5 shows the numbers of adequate (*at*), extra (*et*), and deficient (*dt*) characters. Adequate characters were correctly stored in their corresponding macros (true positive). Extra characters were adversely stored in macros despite not actually changing (false positive). Deficient characters were actually changed but were not stored in their corresponding macros (false negative). There were some inconsistencies between the number of macros and characters. For example, in the tasks T2 and T6 using OR, the existence of extra macros is inconsistent with no extra character. This is because OR recorded macros storing a zero-length text string in these tasks. Moreover, in the tasks T1 and T3, and T5 using FLT, there was no deficient macro; nevertheless, several deficient characters appeared. This was due to the implementation of FLT that automatically eliminates the common prefix of inserted and deleted text string in the same macro. This elimination somewhat makes against in this comparison.

In the same way as counting macros, the number of characters actually stored is calculated by $ad + et$, and the ideal number of them is calculated by $ad + dt$. Moreover, the precision and recall values can be calculated by the following formulas: $Precision = ad/(ad + et)$ and $Recall = ad/(ad + dt)$. *F-measure* is calculated by $2 \times Precision \times Recall/(Precision + Recall)$. Cells colored with purple indicate highest values with respect to the accurate recording.

Answer to Q2. CMR can record macros that store textual changes actually performed on not only opened files in 7/9 of the tasks but also not-opened ones in 2/3 of the tasks. On the other hand, OR and FLT sometimes disregard textual changes within the opened files and never record ones within the not-opened files. On the character-based comparison, CMR had a fairly high precision value (0.998) for the opened files although the value was slightly inferior to OR, and had the highest recall value (0.986) and F-measure value (0.992) among the three tools for the opened files. Moreover, CMR is obviously advantageous to CMR and FLT with respect to the precision, recall, and F-measure values for the not-opened files. These results demonstrate that CMR outperforms two other tools.

5. Discussion

We discuss factors that affect the effectiveness and usefulness of CMR and describe implications for developing a change-recording tool.

5.1 Limitations

We adopt a design policy in which the implementation of CMR exploits only public APIs provided by Eclipse since the use of undisclosed modules has high potential to obsolete the current implementation or cause the reconstruction in the future. Unfortunately, Eclipse does not provide sufficient APIs on event notification. This brings two limitations.

5.1.1 Notification of Canceling Actions

The first limitation is derived from a fact that Eclipse does not notify the predefined listeners of cancellations of refactoring and code completion actions when a programmer stops to execute them. Hence, CMR cannot explicitly know these cancellations. To overcome this problem, CMR watches the movement of the cursor position on source code. If it detects the cursor movement that is caused by a programmer's mouse operation during the execution of refactoring or code completion, it deems the refactoring or the

code completion to be canceled.

This workaround works well in most cases, but might induce the inaccurate recording in case that a programmer do an innocuous cursor movement. For example, in the task T1 described in Sect. 4.1, CMR detected this innocuous cursor movement recorded as a macro (appearing at the line 7 in Fig. 4) and then recorded an inaccurate macro (appearing at the line 8) that cancels the executing refactoring, although the refactoring was never canceled in fact. As a result, this cancellation macro blocks the removal of two extra macros (appearing at the lines 9 and 10), which store volatile textual changes.

5.1.2 Notification of Resource Changes

The second limitation is slightly serious, which happens from the timing of notification of a resource change. Eclipse sends a resource change event immediately after the resource is changed but does not send it before the resource change. Hence, CMR cannot obtain the content of a file before its change. To certainly capture a change of a not-opened file, its contents before and after the change are required. However, it is inefficient to monitor the contents of all files including not opened files if a large number of files exist in a project. To avoid this inefficient monitoring, CMR obtains the previous content of a file from its local history managed by Eclipse if the file has not been opened yet.

Unfortunately, this workaround has the potential to cause the loss of textual changes within not-opened files. In the application of refactoring involving a resource change, Eclipse should immediately notify the predefined listener of the change. Nevertheless, it might silently update the contents of the refactored file and refresh its local history before the notification. This silent refreshment hinders CMR from correctly obtaining the previous contents of the refactored file.

In our perspective based on the results described in Sect. 4.3.2, this problem happens only when refactoring (e.g., Move Class) alters the file path of a not-opened file. In Eclipse, moving a file might be achieved by both deleting a file and creating a new file having the same contents. In case that a silent refreshment is made between the deletion and creation, CMR cannot accurately recording textual changes during the application of refactorings that change file paths. For an opened file, no problem occurs since the document listener binding to it can capture every textual change caused by the applied refactoring.

5.2 Performance

In our experiment mentioned Sect. 4.1, the programmer never felt the overhead time to record textual changes, using the three tools for the thirteen Java files, totaling 700+ lines of code. To clearly articulate the performance impact of CMR on the programmer's editing activities for larger scale of source code, we conducted a supplemental experiment, using the source code of JDK1.8.0_131, consisting of 7,714 Java files, totaling 2,402,972 lines. We prepared three sets of source code: a large one containing the whole source code, a medium one containing packages whose names start with java and javax (3,746 files, totaling 1,457,468 lines), and a small one containing packages that start with java.lang, java.awt, and javax.swing (consisting of 1,342 files, totaling 645,251 lines).

The process of recording macros except for both DocumentMacro and TriggerMacro simply outputs information on events obtained through several built-in listeners of Eclipse. In our experiment, the time period of this process was too small to measure. Meanwhile, the recording process of DocumentMacro and TriggerMacro includes the post-processing steps for raw macros described in Sect. 3.3. In addition, DocumentMacro requires either checking the inconsistency of the inserted and deleted text stored for an opened file during editing (to generate one labeled with EDIT or IRREGULAR_DIFF) or calculating textual differences between the contents before and after a resource change for a not-opened file (to generate one labeled with AUTO_DIFF). A long execution time of the inconsistency checking or the difference calculation might intercept programmer's edits of source code.

To verify this, we preliminarily measured the execution times in recording macros that store textual changes by the automated application of Extract Method, Inline Method, and Rename Method refactorings. As a result, Extract Method and Inline Method, which updated only a single file, were performed in less than one second, and the execution times were almost the same under eight trials using and not using CMR. In contrast, a long time was required when Rename Method, which must rewrite every occurrence of the renamed element, concurrently updated a considerable number of files. In our trials, Rename Method updating about 400 files consumed more than 10 seconds.

From these preliminary results, we concluded that rename refactorings are reasonable to examine the worst performance impact of CMR and selected the following three refactorings.

- R1 Rename the method `append(String)` of the class java.lang.StringBuilder. Textual changes are distributed in many files that refer to this method.
- R2 Rename the method `setName(String)` of the class java.awt.Component. This method exists in a longer file having 10,155 lines.
- R3 Rename the class `java.lang.StringBuffer`. Many files refer to this class.

In the application of all these automated refactorings, only one file during editing was opened and other files had never been opened. In this situation, CMR records some or none of DocumentMacro labeled with EDIT and many ones labeled with AUTO_DIFF. This situation is based on results of our preliminary measurement in Rename Method refactoring. Here note that only 99 files can be opened at the same time on Eclipse. Therefore, in this measurement, we prepared one file defining a method to be renamed and

Table 6 Execution times when using and not using CMR and the overhead times (Δ).

Refac-toring	Large (7,714 files)					Medium (3,746 files)					Small (1,342 files)				
	fs	*ms*	t_w (s)	t_o (s)	Δ (s)	*fs*	*ms*	t_w (s)	t_o (s)	Δ (s)	*fs*	*ms*	t_w (s)	t_o (s)	Δ (s)
R1	394	2,646	12.681	3.504	9.178	192	1,396	7.295	2.111	5.185	46	348	2.115	0.665	1.451
R2	31	104	1.096	0.567	0.528	29	74	1.031	0.619	0.412	29	74	0.975	0.476	0.499
R3	323	1,090	10.832	4.292	6.541	115	419	5.685	2.150	3.535	35	99	2.282	1.319	0.963

98 files referring the method. The execution time when all the 99 files were opened took up to 1.7 seconds longer than that when one of them was opened. This means the status (opened or not-opened) of files has an insignificant effect on programmer's editing activities.

Table 6 shows the measured results of the execution times to update the contents of files during the three refactorings in the three sets of source code. The *fs* and *ms* indicate the number of unique files updated by each of the refactorings and the number of recorded DocumentMacro, respectively. The t_w and t_o denote an average of two execution times per refactoring, using and not using CMR, respectively. Note that neither t_w nor t_o contains the execution time of tasks that check preconditions and write the updated files into the disk. These tasks are commonly performed in the refactorings that are recorded or not. The t_o was measured using an Eclipse with the dedicated plug-in that aims to output the starting and ending time of an applied refactoring. The symbol Δ is the calculation result of $t_w - t_o$, which means the overhead time of recording macros in CMR.

In Table 6, a notable point is that the overhead times for R2, which target a relatively small number of the updated files and the recorded macros, were all so small (less than one second) as to be negligible. Moreover, it is clear that the overhead times (Δ) depend on the number of updated files (*fs*) and the number of DocumentMacro (*ms*)[†]. The more files were updated (or the more macros were recorded), the longer overhead time it took. On the other hand, the overhead time seems to be little related to the size of source code (the total number of files existing in the respective sets).

Although the maximum overhead time was more than 9 seconds when R1 was applied in the large set, we do not consider that this overhead time is an obstacle to programming using IDEs. Almost all programmers have few opportunities to apply a refactoring that updates more than 100 files together in their real programming. Moreover, such a refactoring consumes a long execution time by its nature due to the precondition checking and file writing, without respect to macro recording. For these reasons, CMR is unlikely to have a serious negative impact on the performance of the execution time.

5.3 Threats to Validity

There are several threats to the validity of our evaluation.

[†]Note that *ms* has highly positive correlation with *fs*. The value of correlation coefficient (Pearson product-moment) between *fs* and *ms* was 0.926.

5.3.1 Construct Validity

To achieve fair evaluation under a situation where the three tools record different granularity levels of textual changes, we defined the criterion that judges which macros are adequate, extra, and deficient for storing textual changes, as mentioned in Sect. 4.2. Especially, the concept of isolated chunks of source code text enabled us to systematically count the number of deficiency macros. Through a careful examination of actually recorded macros, we strove to prevent the criterion from being influenced by the design policies of the three tools. However, we have no proof that this criterion makes a correct judgment for every textual change that is both recorded and not recorded in our experiment. Hence, we cannot negate a possibility that this criterion is unconsciously biased. Moreover, it is obviously hard to establish a widely accepted theory that determines what textual changes should be recorded in order to support the exact understanding of source code evolution. Without a reasonable theory, a simple quantitative evaluation using the number of recorded macros seems to be unfit.

5.3.2 Internal Validity

In our experiment, all the tasks were performed with a particular version of Eclipse on MacOS. As mentioned in Sect. 5.1, the implementation of CMR much depends on the event notification mechanism of Eclipse. If such mechanisms differ between versions and platforms, experimental results might also differ.

In our experiment described in Sect. 4.1, we performed the same edits and actions twice for the nine tasks. However, all of them were not performed at the same timing (the second recording was done while watching the screencast of the first one). Additionally, OR recorded extra save-operations for files whose status were forcibly changed since refactoring can be applied only the saved files. The timing differences and extra save actions might invite extra and/or deficient macros.

FLT provides an option that combines commands with the same type. The delimiters in the text combination of CMR is also configurable. In our experiment, the default settings for the three tools were used. If the settings are changed, they might be able to achieve better results than those used in the experiment.

The final threat to internal validity is the selection of programming tasks. Of course, we tried not to make CMR advantageous in the selection. However, the selected tasks might have potential to bias the edits and actions performed

in the experiment since the programmer was one of the authors and had much knowledge of the design and implementation of change-recording tools. From this perspective, the experimental results might be twisted. To obtain results that show solid benefits, we would have to design experiments that use programming tasks in realistic software development and evolution.

5.3.3 External Validity

Due to the heavy burden of manual examination of recorded macros, our experiment contained only the nine programming tasks. These tasks are not of course representative for all kinds of programming tasks and styles in industry and non-industry organizations. Therefore, different results would be obtained from experiments with other tasks performed by a variety of programmers. Additionally, only one target source code was used in the experiment. Programs in other domains and ones with the higher (or lower) quality characteristics would bring different results. To check whether our experimental results can be generalized to describe any situation in software development and evolution, a large number of experiments consisting of various kinds of programming tasks evolving various kinds of source code are needed.

5.4 Implications

Although CMR targets Eclipse for its running environment, we obtained two implications independent with a particular IDE through its design and implementation.

The first implication is related to the kinds of macros. The nine atomic macros except for CancelMacro represent events that can be captured through several built-in listeners of Eclipse. Although other IDEs embed different types of event listeners, the recording of the same or similar macros seems to be essential for presenting textual changes and actions that might cause those textual changes. Irrespective of the features of IDEs, it is better to record macros like TriggerMacro, which represent the starting and ending times of the programmer's actions since they are useful for producing a macro group like CompoundMacro. This grouping helps programmers and researchers easily and quickly find correlations between recorded textual changes and actions related to them. CancelMacro might be either necessary or unnecessary, depending on what mechanism each IDE employs in refactoring under the inline mode and code completion. Note that, of course, more kinds of macros can be taken into consideration if a target IDE provides further advance ways of changing source code text.

The other implication is a recommendation for IDE developers. Both the limitations of CMR are derived from an unsophisticated mechanism of event notification in Eclipse as mentioned in Sect. 5.1. We recommend that all IDEs should implement a mechanism that can certainly receive both before-the-fact and after-the-fact events that represent the occurrences of all actions they support. The upfront investment for this mechanism surely facilitates the implementation of tools that record textual changes on source code not only being edited but not-being edited, and makes such tools more reliable since no workaround is needed.

6. Related Work

There are currently several change-recording tools that have been proposed over a decade, all of which treat changes (or transformations) as first-class entities. In particular, the concept of a change-oriented programming environment (COPE) [18] is widely accepted. It envisions a future in which all IDEs will aggressively record, share, and utilize source code changes to improve the efficiency of source code evolution tasks and increase the quality of ever evolving source code.

Robbes et al. developed SpyWare [3], a tool for recording programmer's source code edits and activities such as refactoring and development session. Each source code change is represented by the addition, deletion, or move of a node or a sub-tree of the AST. The recording of source code changes attains by hooking change notification provided by the Smalltalk IDE. At this time, Spyware has been ported to the Eclipse as Syde [19].

Ebraert et al. developed ChEOPS [20] that records the history of changes on the Smalltalk IDE, In ChEOPS, a programmer does not write source code but instead explicitly applies change operations to the source code, by using FAMIX [21] that is a programming language-independent model to represent source code entities. ChEOPS currently supports Java as ChEOPSJ [22].

FeedBaG++ [23] collects enriched event streams with source code snapshots in programming activities performed on Visual Studio. The collected snapshots are represented in the form of simplified syntax trees (SSTs) that contain information on source code changes. It has an aspect of the tool platform since it provides several tooling facilities to analyze and manipulate SSTs.

Unlike SpyWare, ChEOPS, and FeedBaG++, OperationRecorder (OR) [11] and Fluorite (FLT) [12] directly records changes of source code text itself although it cannot be correctly parsed. Therefore, all of them can collect textual changes that a programmer actually performs an Eclipse's editor and record them along with their timestamps. A large advantage is that the source code existing in the past can be easily restored from recorded textual changes.

To collect various kinds of change operations performed on Eclipse and the programmer's actions, Nagara et al. proposed CodingTracker [4] that replaces several modules in Eclipse. This tool infers a program entity on which a change was performed by using the offset value and the length of the changed text on the source code. In other words, it converts textual changes on the source code into change operations (addition, deletion, and property update) of AST nodes. To our knowledge, CodingTracker attains the high accuracy of recording but is not popularly used

at present. This is because it is too much dependable on the Eclipse's codebase and hard to follow a new release of Eclipse.

Fine-grained source code changes might be able to be obtained from change recovery approaches. For example, ChangeDistiller [24], GumTree [25], and GumTree's extensions [26]–[28] try to produce edit scripts (the addition, deletion, update, and move of AST nodes) that transform one AST into another. Although produced edit scripts achieve the correct transformation, it is hard to infer the chronological order of those edit scripts and the occurrences of shadowed or overlapping changes. Therefore, in most cases, the edit scripts are not exactly the same as source code changes actually performed by a programmer.

7. Conclusions and Future Work

From the viewpoint of promoting the utilization of fine-grained textual changes of source code, a change-recording tool should accurately record textual changes that are performed on its target IDE. Our proposed CMR is a high-potential candidate for this tool although it is hard to generalize how much effective it is, since it could record macros that store textual changes actually performed on not only opened files but also not-opened ones with high accuracy in our experiment. Moreover, CMR records more kinds of macros and a greater number of than the existing tools OR and FLT, and is capable of producing a compound macro that reveals the correlation of the recorded textual changes and actions causing them.

Two future issues remain for the enhancement of our work. We have to collect a large volume of textual changes in real software development and evolution using CMR. Moreover, we have to conduct further experiments that assess how it can help future source code evolution. A sufficient number of reliable experimental results would encourage many programmers to install CMR into their Eclipse.

The other future work will discuss the expanse of the programmer's and IDE's actions to be recorded. The latest implementation of CMR dare not record several IDE interactions occurring when programmers run a program, make tests, debug a program, browse source code, retrieve code fragments, and so on. On the other hand, OR and FLT can record a few of such interactions. We know, of course, that rich information recorded by this tool is likely to facilitate the reasoning about the programmer's intention or the programming context. However, too much information is unmanageable for many programmers and researchers. For this reason, we have to consider a sophisticated model that manages fine-grained code changes and their related IDE interactions, which becomes truly common between tools running on a variety of IDEs. To this end, we will extend the features of CMR and migrate CMR from Eclipse to different IDE such as Visual Studio and IntelliJ IDEA.

Acknowledgments

This work was supported by JSPS KAKENHI Grant Numbers JP15H02685, JP15K15970, and JP18K11238.

References

[1] M.M. Lehman, "Programs, life cycles, and laws of software evolution," Proceedings of IEEE, vol.68, no.9, pp.1060–1076, 1980.
[2] A. Hora, D. Silva, M.T. Valente, and R. Robbes, "Assessing the threat of untracked changes in software evolution," Proceedings of the 40th International Conference on Software Engineering, ICSE '18, pp.1102–1113, 2018.
[3] R. Robbes and M. Lanza, "A change-based approach to software evolution," Electronic Notes in Theoretical Computer Science, vol.166, pp.93–109, 2007.
[4] S. Negara, M. Vakilian, N. Chen, R.E. Johnson, and D. Dig, "Is it dangerous to use version control histories to study source code evolution?," Proceedings of the 26th European Conference on Object-Oriented Programming, ECOOP '12, vol.7313, pp.79–103, 2012.
[5] Q.D. Soetens, R. Robbes, and S. Demeyer, "Changes as first-class citizens: A research perspective on modern software tooling," ACM Computer Surveys, vol.50, no.2, pp.18:1–18:38, 2017.
[6] K. Herzig and A. Zeller, "The impact of tangled code changes," Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, pp.121–130, 2013.
[7] T. Omori and K. Maruyama, "Comparative study between two approaches using edit operations and code differences to detect past refactorings," IEICE Transactions on Information Systems, vol.E101-D, no.3, pp.644–658, 2018.
[8] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," Journal of Systems and Software, vol.101, no.C, pp.193–220, 2015.
[9] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M.D. Penta, A.D. Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad (and whether the smells go away)," IEEE Transactions on Software Engineering, vol.43, no.11, pp.1063–1088, 2017.
[10] Y. Nishimura and K. Maruyama, "Supporting merge conflict resolution by using fine-grained code change history," Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER '16, pp.661–664, 2016.
[11] T. Omori and K. Maruyama, "A change-aware development environment by recording editing operations of source code," Proceedings of the 2008 International Working Conference on Mining Software Repositories, MSR '08, pp.31–34, 2008.
[12] Y. Yoon and B.A. Myers, "Capturing and analyzing low-level events from the code editor," Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools, PLATEAU '11, pp.25–30, 2011.
[13] K. Maruyama, S. Hayashi, and T. Omori, "ChangeMacroRecorder: Recording fine-grained textual changes of source code," Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering, SANER '18, pp.537–541, 2018.
[14] K. Maruyama, T. Omori, and S. Hayashi, "Slicing fine-grained code change history," IEICE Transactions on Information Systems, vol.E99-D, no.3, pp.671–687, 2015.
[15] K. Maruyama and S. Hayashi, "A tool supporting postponable refactoring," Proceedings of the 39th International Conference on Software Engineering Companion, ICSE '17, pp.133–135, 2017.
[16] G.C. Murphy, M. Kersten, and L. Findlater, "How are Java software developers using the Eclipse IDE?," IEEE Software, vol.23, no.4, pp.76–83, 2006.
[17] E. Murphy-Hill, C. Parnin, and A.P. Black, "How we refactor, and how we know it," IEEE Transactions on Software Engineering, vol.38, no.1, pp.5–18, 2012.

[18] D. Dig, R. Johnson, D. Marinov, B. Bailey, and D. Batory, "COPE: Vision for a change-oriented programming environment," Proceedings of the 38th International Conference on Software Engineering Companion, ICSE '16, pp.773–776, 2016.
[19] L. Hattori and M. Lanza, "Syde: A tool for collaborative software development," Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE '10, pp.235–238, 2010.
[20] P. Ebraert, J. Vallejos, P. Costanza, E.V. Paesschen, and T. D'Hondt, "Change-oriented software engineering," Proceedings of the 2007 International Conference on Dynamic Languages: In Conjunction with the 15th International Smalltalk Joint Conference 2007, ICDL '07, pp.3–24, 2007.
[21] S. Demeyer, S. Ducasse, and S. Tichelaar, "Why unified is not universal: UML shortcomings for coping with round-trip engineering," Proceedings of the 2nd International Conference on The Unified Modeling Language: Beyond the Standard, UML'99, vol.1723, pp.630–644, 1999.
[22] Q.D. Soetens and S. Demeyer, "ChEOPSJ: Change-based test optimization," Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering, CSMR '12, pp.535–538, 2012.
[23] S. Proksch, S. Nadi, S. Amann, and M. Mezini, "Enriching in-IDE process information with fine-grained source code history," Proceedings of the 24th International Conference on Software Analysis, Evolution, and Reengineering, SANER '17, pp.250–260, 2017.
[24] B. Fluri, M. Wursch, M. Pinzger, and H. Gall, "Change Distilling: Tree differencing for fine-grained source code change extraction," IEEE Transactions on Software Engineering, vol.33, no.11, pp.725–743, 2007.
[25] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14, pp.313–324, 2014.
[26] Y. Higo, A. Ohtani, and S. Kusumoto, "Generating simpler AST edit scripts by considering copy-and-paste," Proceedings of the 32nd ACM/IEEE International Conference on Automated Software Engineering, ASE '17, pp.532–542, 2017.
[27] K. Huang, B. Chen, X. Peng, D. Zhou, Y. Wang, Y. Liu, and W. Zhao, "ClDiff: Generating concise linked code differences," Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE '18, pp.679–690, 2018.
[28] V. Frick, T. Grassauer, F. Beck, and M. Pinzger, "Generating accurate and compact edit scripts using tree differencing," Proceedings of the IEEE International Conference on Software Maintenance and Evolution, ICSME '18, pp.264–274, 2018.

Katsuhisa Maruyama received his B.E. and M.E. degrees in electrical engineering and Ph.D. in information science from Waseda University, Japan, in 1991, 1993, and 1999, respectively. He is a professor of College of Information Science and Engineering, Ritsumeikan University. He has worked for NTT (Nippon Telegraph and Telephone Corporation) and NTT Communications Corporation before he joined Ritsumeikan University. He was a visiting researcher at Institute for Software Research (ISR) of University of California, Irvine (UCI). His research interests include software refactoring, program analysis, software reuse, object-oriented design and programming, and software development environments. He is a member of the IEEE, ACM, IPSJ, and JSSST.

Shinpei Hayashi received his B.E. degree in information engineering from Hokkaido University in 2004. He also received his M.E. and Ph.D. degrees in computer science from Tokyo Institute of Technology in 2006 and 2008, respectively. He is currently an associate professor of School of Computing at Tokyo Institute of Technology. His research interests include software changes and software development environments.

Takayuki Omori is an associate professor at College of Information Science and Engineering, Ritsumeikan University. He obtained his Ph.D. in Engineering from Ritsumeikan University in 2008. From September 2013 to March 2014, he was a visiting assistant professor at the University of British Columbia. His current research interests include software evolution, and fine-grained code changes in software development.