2013 20th Asia-Pacific Software Engineering Conference

Detecting Program Changes from Edit History of Source Code

Eijirou Kitsu
Grad. Sch. Information Science and Engineering
Ritsumeikan University
Kusatsu, Japan
kitsu@fse.cs.ritsumei.ac.jp

Takayuki Omori
Dept. Computer Science
Ritsumeikan University
Kusatsu, Japan
takayuki@fse.cs.ritsumei.ac.jp

Katsuhisa Maruyama
Dept. Computer Science
Ritsumeikan University
Kusatsu, Japan
maru@cs.ritsumei.ac.jp

***Abstract*—Detecting program changes helps maintainers to figure out the evolution of the changed program. For this, several line-based difference tools have been proposed, which extract differences between two versions of the program. Unfortunately, these tools do not provide enough support to program comprehension since a single commitment stored in a version control system contains multiple changes that are intermingled with each other. Therefore, the maintainers have to untangle them by hand. This work is troublesome and time-consuming. This paper proposes a novel mechanism that automatically detects individual program changes. For this, it restores snapshots of the program from the history of edit operations for the target source code and compares class members that result from syntax analysis for respective snapshots. In addition, the mechanism provides several options of aggregating fine-grained changes detected based on the edit history. The maintainers can select their suitable levels of summarization of program changes. The paper also shows experimental results with a running implementation of the change detection tool. Through the experiment, the detection mechanism presents various kinds of summarized information on program changes, which might facilitate maintainers' activities for program comprehension.**

***Keywords*-Software change detection, program analysis, software evolution, change-aware development environment**

I. Introduction

Software maintainers have to comprehend a program before they start modifying it. Although careful review of the program code is often done, reviewing the current snapshot of the code is not enough to understand its evolution. The maintainers should also grasp how the code has been changed before. For this, line-based difference tools (e.g., UNIX diff utilities) can be frequently used. They extract differences between the original revisions of a program stored in a version control system (VCS). The collection of these differences is considered as change history of the program.

Unfortunately, several limitations of this type of change detection have been pointed out by Robbes and Lanza [1]. In general, developers can check in their source files in a VCS whenever they want to commit them. Therefore, a single commitment contains multiple changes intermingling with each other [2]. In other words, one change stored in the VCS is not always equal to one a developer expects. Line-based difference approaches are hard to untangle such changes in a commitment and to detect an individual changes from them. If maintainers must this work by hand, it is troublesome and time-consuming for them. To aid them to comprehend a program they want to modify, an automatic mechanism for detecting individual changes is desired.

To overcome the problem that multiple changes are tangled in the same commitment, new approaches have been proposed in [3]–[5]. In these approaches, development environments automatically record fine-grained changes developers have performed on the editors running in the environments. This allows them to keep track of individual changes they have performed before. For example, SpyWare [3] records all kinds of program modifications of source code whenever its syntactical changes are performed on a Squeak environment. Syde [4] is a plugin running on one of popular development environments, Eclipse. It records small changes based on the Robbes' change-based software evolution model [1]. OperationRecorder [5], which was proposed in our previous work, records all edit operations that might alter the contents of source files under editing in development sessions.

In these ways, recording fine-grained changes provides the potential for assisting program comprehension. However, these changes are often too small for human to understand. Developers or maintainers do not want to know details of what code fragments were inserted in a method that was added into an existing class. Instead, they want to know the overall fact that a new method was added. In other words, they want to grasp program changes not code changes. In the context of object-oriented programs, the changes with respect to classes, methods, and fields could be appropriate for program changes [6], [7].

This paper proposes a new mechanism that detects program changes from code edit history that recorded in past development sessions. It compares two versions of source code, both of which can be restored from edit operations. By using the edit operations, the difference between the two versions contains only fine-grained changes. This helps the mechanism to extract individual changes done in the past without untangling them. In addition, the mechanism aggregates the extracted individual changes. This is because

1530-1362/13 $31.00 © 2013 IEEE
DOI 10.1109/APSEC.2013.48

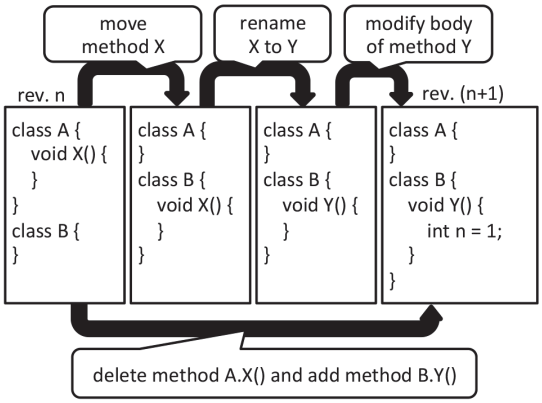

Figure 1. Intermingled multiple changes and detection of them.

aggregation of precise individual changes could produce precise program changes. By extracting small differences and aggregating them, the mechanism can detect more precise program changes and provide them to maintainers. Unfortunately, the conventional line-based difference mechanisms might present imprecise program changes since the detected differences always depend on timing of developers' commitments. On the other hand, the proposed mechanism has no concern with their actions with respect to code commitments.

The main contributions of this paper include:

- A novel mechanism to detect more precise program changes from edit operations that developers have performed in the past,
- A running implementation that automatically detects program changes, and
- Three case studies that show the use of the mechanism.

The rest of the paper is organized as follows: Section II presents a motivating example. Section III explains a difference extraction mechanism of the program change detection. Section IV explains a difference aggregation mechanism. Section V presents a running implementation of the program change detection. Section VI shows experimental results including three case studies. Section VII concludes with a brief summary and remaining issues.

II. Motivating Example

Intermingled changes often hinder the detection of a precise program change. We present an example that one of the existing line-based difference tools detects an imprecise program change. Fig. 1 shows a situation when the following three modifications have been finished.

1) The method `X()` was moved from the class `A` to the class `B` (Move Method),
2) The moved method `X()` of `B` is renamed into `Y()` (Rename Method), and
3) Any code was inserted into the body of the renamed method `Y()` of `B` (Change Method Body).

Here, the version of the leftmost source code is n (called revision n) and that of the rightmost source code is $n+1$ (called revision $(n+1)$). In other words, the three modifications were performed under the change between the two revisions n and $(n+1)$. In this situation, a usual UNIX diff tool presents the following information:

```
2,3d1
<     void X() {
<     }
6a5,7
>     void Y() {
>         int n = 1;
>     }
```

The top three lines denote the deletion of the method `X()`, and the last four lines denote the addition of the method `Y()`. As compared with the actual three modifications, the lack of some information was occurred. In this case, a special technique using the similarity of respective methods is required. Therefore, it is troublesome to detect individual program changes enclosed in the difference information provided by the UNIX diff tool. The proposed mechanism aims at revealing individual changes under this situation.

Besides this kind of detection, the mechanism tries to aggregate the detected changes and produce their summary. It allows maintainers to select one from among several options of summarization. For example, some maintainers want to obtain a panoramic view of change history. In this case, one program change (Move Method) might be better to help them rather than the three individual changes. On the other hand, there is some maintainers who have to figure out the change history in detail. For them, an option of preserving the individual changes is preferable. To satisfy various kinds of needs, the selection of the summarization level is essential since the suitable level of summarization depends on needs of maintainers.

III. Detection of Program Changes

Table I denotes the classification of program changes the proposed mechanism can detect through syntax analysis of Java source code.

We define these program changes based on ones used in impact analysis of Java programs [6], [7].

Each program change is stored in the change database, which has four kinds of information.

1) Identification number (CID) that is automatically assigned in order to identify the change.
2) Starting time and ending time of the change. The starting time indicates the editing time of the earliest edit operation among all edit operations related to the change. The ending time indicates the editing time of the latest edit operation among the related edit operations. These times are automatically recorded when the change is performed.
3) Names of a file, a package, and a class containing the change. These names are extracted from snapshots of source code before and after the change.

Table I
TYPES OF PROGRAM CHANGES.

Types	Meaning
AC	Adding a new class to a class or package
DC	Deleting an existing class from a class or package
MC	Moving an existing class to another class or package
CCN	Changing the name of a class
AF	Adding a new field to a class
DF	Deleting an existing field from a class
MF	Moving an existing field to another class
CFN	Changing the name of a field
CFT	Changing the type of a field
AM	Adding a new method to a class
DM	Deleting an existing method from a class
MM	Moving an existing method to another class
CMN	Changing the name of a method
CMT	Changing the type of a method
CMP	Changing any parameter of a method
CMB	Changing the body of a method

4) Type of a change, which is one of the changes shown in Table I.

A. Overview of the detection mechanism

Fig. 2 shows an overview of the detection mechanism. It uses editing operations provided by OperationRecorder [5], which records all editing operations of source code. An edit operation has information on the time when it was performed, the location where it was performed, and the text actually added and deleted. By applying edit operations to the contents of the initial source code, we can obtain any snapshot of the source code.

The detection mechanism newly proposed in this paper consists of three phases. In the first phase, the mechanism restores all snapshots of Java source files existing in a project. In other words, it generates one snapshot related to all the Java source files once one edit operation is performed. Consider that an edit operation was performed at the time t. The source code S at t, which is denoted by S_t, can be obtained by applying every edit operation of S that have been performed between the times 0 and t.

In the second phase, the mechanism tries to parse Java source files contained in each snapshot. If a source file is successfully parsed, it constructs its abstract syntax tree (AST) and extracts information related to classes and their members. The collection of all class information for the snapshot of the source file S_t is denoted by $I(S_t)$. Here, we define two adjacent source files S_i and S_j $(i < j)$ both of which are parseable and there is no parseable source file S_k that satisfies $i < k < j$.

In the final phase, the mechanism analyzes class information stored in $I(S_i)$ and $I(S_j)$ where S_i and S_j are adjacent source files. The detail of this phase will be explained in Sections III-B.

B. Algorithm Analyzing Class Information

The detection mechanism analyzes class information to detect program changes shown in Table I. The class information has the following three attributes.

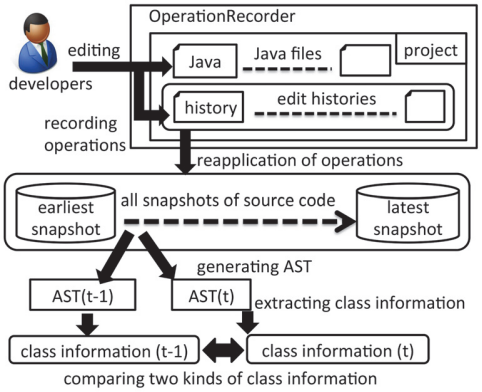

Figure 2. Overview of the detection mechanism.

1) Names The fully qualified name of a class and the name of a Java source file containing the class. The $c.name$ and $c.fname$ represent the names of a class c and its source file, respectively. An anonymous class is ignored since it has no name.
2) Fields A list of all fields defined in the class. Each of them has two properties: its name and type. For a field f, its name and type are represented by $f.name$ and $f.type$, respectively.
3) Methods A list of all methods defined in the class. Each of them has four properties: its name, return type, the list of its parameter types, and the AST of its body. For a method m, its name, return type, parameter type list, the AST of body are represented by $m.name$, $m.type$, $m.par$, and $m.body$, respectively.

The analysis of class information are done by Algorithms 1, 2, 3, and 4. *ClassChange*(S_{t1}, S_{t2}) tries to detect **AC**, **DC**, and **CCN**. It uses two adjacent source files S_{t1} and S_{t2} for the times $t1$ and $t2$ $(t1 < t2)$, which might contain such program changes. In the algorithms, the function $\mathsf{ops}(t1, t2)$ represents a set of all edit operations performed form $t1$ through $t2$. The function $\mathsf{max}(v1, v2)$ returns the greater of two values $v1$ and $v2$. $\#S$ denote the size of a set S and $S[n]$ denotes the n-th element in S.

We should explain the meaning of "an program element (a class, a field, or a method) was removed or inserted by edit operations" in detail. Consider all edit operations in $\mathsf{ops}(t1, t2)$ are applied to a snapshot S_{t1}. If a snapshot S_{t2} does not contain an element e_r existing in S_{t1}, e_r was removed by $\mathsf{ops}(t1, t2)$. If S_{t2} contains an element e_i not existing in S_{t1}, e_i was inserted by $\mathsf{ops}(t1, t2)$.

FieldChange(C_{t1}, C_{t2}) tries to detect **AF**, **DF**, **CFN**, and **CFT**. On the other hand, *MethodChange*(C_{t1}, C_{t2}) tries to detect **AM**, **DM**, **CMN**, **CMT**, **CMP**, and **CMB**. Both of them use class information related to class pairs (C_{t1}, C_{t2}).

Algorithm 1: *ClassChange*(S_{t1}, S_{t2})

$C_{t1} \leftarrow$ a set of all classes existing in S_{t1};
$C_{t2} \leftarrow$ a set of all classes existing in S_{t2};
foreach $c_{t1} \in C_{t1}$ **do**
  if c_{t1} *was removed by* `ops(`$t1, t2$`)` **then**
    the change type of c_{t1} is **DC**; remove c_{t1} from C_{t1};
    the changes type of fields in c_{t1} are all **DF**;
    the changes type of methods in c_{t1} are all **DM**;
  end
end
foreach $c_{t2} \in C_{t2}$ **do**
  if c_{t2} *was inserted by* `ops(`$t1, t2$`)` **then**
    the change type of c_{t2} is **AC**; remove c_{t2} from C_{t2};
    the change types of fields in c_{t2} are all **AF**;
    the change types of methods in c_{t2} are all **AM**;
  end
end
foreach $c_{t1} \in C_{t1}$, $c_{t2} \in C_{t2}$ **do**
  if c_{t1}.*name* $=$ c_{t2}.*name* **then**
    FieldChange(c_{t1}, c_{t2});
    MethodChange(c_{t1}, c_{t2});
    remove c_{t1} from C_{t1}; remove c_{t2} from C_{t2};
  end
end
sort classes remaining in C_{t1} in the order of their appearance;
sort classes remaining in C_{t2} in the order of their appearance;
for $n \leftarrow 1$ **to** `max(`$\#C_{t1}, \#C_{t2}$`)` **do**
  if $C_{t1}[n]$.*name* $\neq$ $C_{t2}[n]$.*name* **then**
    the change types of $C_{t1}[n]$ and $C_{t2}[n]$ are both **CCN**;
    FieldChange$(C_{t1}[n], C_{t2}[n])$;
    MethodChange$(C_{t1}[n], C_{t2}[n])$;
    remove $C_{t1}[n]$ from C_{t1}; remove $C_{t2}[n]$ from C_{t2};
  end
end
the change types of classes remaining in C_{t1} are all **DC**;
the change types of fields in C_{t1} are all **DF**;
the change types of methods in C_{t1} are all **DM**;
the change types of classes remaining in C_{t2} are all **AC**;
the change types of fields in C_{t2} are all **AF**;
the change types of methods in C_{t2} are all **AM**;

Algorithm 2: *FieldChange*(C_{t1}, C_{t2})

$F_{t1} \leftarrow$ a set of all fields in C_{t1};
$F_{t2} \leftarrow$ a set of all fields in C_{t2};
foreach $f_{t1} \in F_{t1}$ **do**
  if f_{t1} *was removed by* `ops(`$t1, t2$`)` **then**
    the change type of f_{t1} is **DF**; remove f_{t1} from F_{t1};
  end
end
foreach $f_{t2} \in F_{t2}$ **do**
  if f_{t2} *was inserted by* `ops(`$t1, t2$`)` **then**
    the change type of f_{t2} is **AF**; remove f_{t2} from F_{t2};
  end
end
foreach $f_{t1} \in F_{t1}$, $f_{t2} \in F_{t2}$ **do**
  if f_{t1}.*name* $=$ f_{t2}.*name* $\wedge$ f_{t1}.*type* $=$ f_{t2}.*type* **then**
    remove f_{t1} from F_{t1}; remove f_{t2} from F_{t2};
  end
end
foreach $f_{t1} \in F_{t1}$, $f_{t2} \in F_{t2}$ **do**
  if f_{t1}.*name* $\neq$ f_{t2}.*name* $\wedge$ f_{t1}.*type* $=$ f_{t2}.*type* **then**
    the change types of f_{t1} and f_{t2} are both **CFN**;
    remove f_{t1} from F_{t1}; remove f_{t2} from F_{t2};
  else if f_{t1}.*name* $=$ f_{t2}.*name* $\wedge$ f_{t1}.*type* $\neq$ f_{t2}.*type* **then**
    the change types of f_{t1} and f_{t2} are both **CFT**;
    remove f_{t1} from F_{t1}; remove f_{t2} from F_{t2};
  end
end
the change types of fields remaining in F_{t1} are all **DF**;
the change types of fields remaining in F_{t2} are all **AF**;

MoveChange(Ds, As) tries to detect **MC**, **MF**, and **MM**. In this algorithm, x.*startingTime* and x.*endingTime* indicate the starting time and the ending time of a change x. The x.*text* denotes the deleted or inserted text by x. It receives two sets of all changes with respect to deletion and addition, which are Ds and As, respectively. A change in Ds indicates either **DC**, **DF**, or **DM** while a change in As indicates either **AC**, **AF**, or **AM**. If the change type of d is replaced with **MC**, its information except for the change type is inherited from that of d. If a is canceled, a is removed from the change database. In this algorithm, d and a are considered as **MC**, **MF**, or **MM** only when they appear in different files. If d and a occur within the same file, these changes seem to achieve a simple relocation.

IV. Aggregation of Program Changes

We present the change detection mechanism in Section III. Unfortunately, it gives rise to the repetition of the same kind of changes. This is because it tries to detect changes every time restored source code can be syntactically parsed. For example, **CMB** are frequently detected since parseable source code often appears during modification of the method body. To help maintainers to understand past changes, consecutive similar changes for the same program element should be aggregated. Here, consecutive changes mean that there is no other change between them under all the changes are sorted in chronological order.

We define two kinds of consecutive changes based on temporal distance and spatial distance. The former indicates how much time does it take to make a next change from a previous one. If two consecutive changes occur within the predefined time period, they are considered as candidates for aggregation with respect to temporal distance. This is called temporal aggregation of changes. The time period is the time difference between the ending time of the previous change and the starting time of the next one.

The spatial distance indicates how far apart are two locations where consecutive changes modify. To find this distance, an AST is created from the source code immediately after the previous change. On the AST, a branch-node directly enclosing the previous change is marked. The branch-nodes represent if-statements, while-statements, do-while-statements, for-statements, switch-statements, and try-statements. For example, the change alters several statements

Algorithm 3: *MethodChange*(C_{t1}, C_{t2})

$M_{t1} \leftarrow$ a set of all methods in C_{t1};
$M_{t2} \leftarrow$ a set of all methods in C_{t2};
foreach $m_{t1} \in M_{t1}$ **do**
  if m_{t1} *was removed by* `ops(`$t1, t2$`)` **then**
    the change type of m_{t1} is **DM**;
    remove m_{t1} from M_{t1};
  end
end
foreach $m_{t2} \in M_{t2}$ **do**
  if m_{t2} *was inserted by* `ops(`$t1, t2$`)` **then**
    the change type of m_{t2} is **AM**;
    remove m_{t2} from M_{t2};
  end
end
foreach $m_{t1} \in M_{t1}$, $m_{t2} \in M_{t2}$ **do**
  if $m_{t1}.name = m_{t2}.name \wedge m_{t1}.type = m_{t2}.type \wedge$
    $m_{t1}.par = m_{t2}.par \wedge m_{t1}.body = m_{t2}.body$ **then**
    remove m_{t1} from M_{t1}; remove m_{t2} from M_{t2};
  end
end
foreach $m_{t1} \in M_{t1}$, $m_{t2} \in M_{t2}$ **do**
  if $m_{t1}.name \neq m_{t2}.name \wedge m_{t1}.type = m_{t2}.type \wedge$
    $m_{t1}.par = m_{t2}.par \wedge m_{t1}.body = m_{t2}.body$ **then**
    the change types of m_{t1} and m_{t2} are both **CMN**;
    remove m_{t1} from M_{t1}; remove m_{t2} from M_{t2};
  else if $m_{t1}.name = m_{t2}.name \wedge m_{t1}.type \neq m_{t2}.type \wedge$
    $m_{t1}.par = m_{t2}.par \wedge m_{t1}.body = m_{t2}.body$ **then**
    the change types of m_{t1} and m_{t2} are both **CMT**;
    remove m_{t1} from M_{t1}; remove m_{t2} from M_{t2};
  else if $m_{t1}.name = m_{t2}.name \wedge m_{t1}.type = m_{t2}.type \wedge$
    $m_{t1}.par \neq m_{t2}.par \wedge m_{t1}.body = m_{t2}.body$ **then**
    the change types of m_{t1} and m_{t2} are both **CMP**;
    remove m_{t1} from M_{t1}; remove m_{t2} from M_{t2};
  else if $m_{t1}.name = m_{t2}.name \wedge m_{t1}.type = m_{t2}.type \wedge$
    $m_{t1}.par = m_{t2}.par \wedge m_{t1}.body \neq m_{t2}.body$ **then**
    the change types of m_{t1} and m_{t2} are both **CMB**;
    remove m_{t1} from M_{t1}; remove m_{t2} from M_{t2};
  end
end
the change types of methods remaining in M_{t1} are all **DM**;
the change types of methods remaining in M_{t2} are all **AM**;

Algorithm 4: *MoveChange*(Ds, As)

$DCs \leftarrow$ all **DC**s included in Ds;
$ACs \leftarrow$ all **AC**s included in As;
foreach $d \in DCs$, $a \in ACs$ **do**
  if $d.endingTime < a.startingTime \wedge d.text = a.text \wedge$
    there are no other ***AC****s between* d *and* a **then**
    $C_d \leftarrow$ a class related to d;
    $C_a \leftarrow$ a class related to a;
    if $C_d.fname \neq C_a.fname$ **then**
      replace the change type of d with **MC**; cancel a;
    end
  end
end
$DFs \leftarrow$ all **DF**s included in Ds;
$AFs \leftarrow$ all **AF**s included in As;
foreach $d \in DFs$, $a \in AFs$ **do**
  if $d.endingTime < a.startingTime \wedge d.text = a.text \wedge$
    there are no other ***AF****s between* d *and* a **then**
    $F_d \leftarrow$ a field related to d;
    $F_a \leftarrow$ a field related to a;
    if $F_d.fname \neq F_a.fname$ **then**
      replace the change type of d with **MF**; cancel a;
    end
  end
end
$DMs \leftarrow$ all **DM**s included in Ds;
$AMs \leftarrow$ all **AM**s included in As;
foreach $d \in DMs$, $a \in AMs$ **do**
  if $d.endingTime < a.startingTime \wedge d.text = a.text \wedge$
    there are no other ***AM****s between* d *and* a **then**
    $M_d \leftarrow$ a method related to d;
    $M_a \leftarrow$ a method related to a;
    if $M_d.fname \neq M_a.fname$ **then**
      replace the change type of d with **MM**; cancel a;
    end
  end
end

enclosed in the true-branch of a certain if-statement. The branch-node corresponding to this if-statement is marked. Similarly, a branch-node enclosing the next change on the AST is marked. Consider a pathway between these marked branch-nodes on the AST. The spatial distance counts the number of edges on the path. If the distance is shorter than the predefined number, two consecutive changes are considered as candidates for aggregation. This is called spatial aggregation of changes. The distance is available when both of the changes are **CMB** since it does not present a significant difference for other program changes.

A. Aggregation Algorithm

Table II shows several heuristic rules for aggregating program changes. The rules in the column "Rules" are written in the Extended Backus-Naur Form (EBNF). In each of the rules, two program changes interleaving a comma are consecutive ones. In addition, we introduce two new symbols: **CFI** and **CMI**. **CFI** is either **CFN** or **CFT**. **CMI** is either **CMN**, **CMT**, **CMP** or **CMB**.

An overview of the aggregation algorithm is shown Fig 3. It first finds pairs of consecutive program changes. Next, it checks whether these are candidates for temporal aggregation or spatial aggregation. The two candidate changes are linked and several clusters containing the linked candidate changes are finally generated. For example, the leftmost two program changes **AM** and **CMN** were linked in Fig 3. After linking all candidate changes, two clusters are generated.

The generated clusters will be matched with the right-hand patterns of the rules shown in Table II. If a matched pattern is found, program changes belonging to the found pattern are aggregated. Actually, these changes are replaced with a new left-hand change. All the information of the new program change is copied from the information on the earliest program change in the replaced program changes. The ending time of the new change is rewritten, which is

Table II
RULES OF AGGREGATION.

Rules	Descriptions
ϕ = **AC**, {**CCN**}, **DC**;	Addition of a class and its deletion is regarded as no changes
ϕ = **AF**, {**CFI**}, **DF**;	Addition of a field and its deletion is regarded as no changes
ϕ = **AM**, {**CMI**}, **DM**;	Addition of a method and its deletion is regarded as no changes
AC = **AC**, **CCN**, {**CCN**};	Addition of a class and modification of its name is aggregated into a single **AC**
AF = **AF**, **CFI**, {**CFI**};	Addition of a field and modification of its property is aggregated into a single **AF**
AM = **AM**, **CMI**, {**CMI**};	Addition of a method and modification of its property is aggregated into a single **AM**
DC = **CCN**, {**CCN**}, **DC**;	Eventual deletion of a class is aggregated into a single **DC**
DF = **CFI**, {**CFI**}, **DF**;	Eventual deletion of a field is aggregated into a single **DF**
DM = **CMI**, {**CMI**}, **DM**;	Eventual deletion of a method is aggregated into a single **DM**
MC = (**CCN** | **MC**), {**CCN** | **MC**};	Multiple moves of a class are aggregated into a single **MC**
MF = (**CFI** | **MF**), {**CFI** | **MF**};	Multiple moves of a field are aggregated into a single **MF**
MM = (**CMI** | **MM**), {**CMI** | **MM**};	Multiple moves of a method are aggregated into a single **MM**
CCN = **CCN**, {**CCN**};	Multiple modifications of the name of a class are aggregated into a single **CCN**
CFN = **CFN**, {**CFN**};	Multiple modifications of the name of a field are aggregated into a single **CFN**
CFT = **CFT**, {**CFT**};	Multiple modifications of the type of a field are aggregated into a single **CFT**
CMN = **CMN**, {**CMN**};	Multiple modifications of the name of a method are aggregated into a single **CMN**
CMT = **CMT**, {**CMT**};	Multiple modifications of the return type of a method are aggregated into a single **CMT**
CMP = **CMP**, {**CMP**};	Multiple modifications of any parameter of a method are aggregated into a single **CMP**
CMB = **CMB**, {**CMB**};	Multiple modifications of the body of a method are aggregated into a single **CMB**

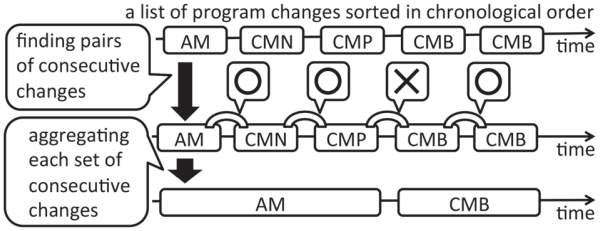

Figure 3. Example of aggregation.

Table III
DETECTION AND AGGREGATION RESULTS.

CID	Starting	Ending	Type	Target	Aggregation
1	14:40:02	14:40:02	**AC**	`A`	**AC**
2	14:40:10	14:40:10	**AC**	`B`	**AC**
3	14:40:21	14:40:25	**AM**	`X`	**AM**
8	14:43:14	14:43:21	**MM**	`X(4,5)`	**MM**
6	14:43:29	14:43:29	**CMN**	`X, Y`	
7	14:43:35	14:43:35	**CMB**	`Y`	

equal to the latest time in the replaced changes. Pattern matching is continued until all the clusters are checked. Consequently, the two changes appear as shown in Fig 3.

V. IMPLEMENTATION

Our change detection tool runs on Eclipse. It reads a history file of edit operations provided by OperationRecorder and a configuration file about its summarization level. It outputs a file listing every detected program change.

To demonstrate the feasibility of the tool, we prepare two Java source files each of which contains the class A or B, respectively. The initial contents of these files are the same as those of the leftmost source code in the Fig. 1. Then, we have made three modifications in the same way by hand. OperationRecorder automatically recorded edit operations for these modifications. The Move Method change was carried out by a pair of cut and paste operations.

Table III shows results of detecting program changes and aggregating them. The "Starting" indicates the starting time of each of the changes. The "Ending" indicates the ending time. The "Type" means the type of change. In "Target", some information is described, including program elements related to the change. Here, the description CID[1] means a program change with CID = 1. The abbreviated description CID[1–3] means program changes with CID = 1, 2, and 3.

In the detection, CID[4] and CID[5] were replaced with CID[8]. CID[1–3] in Table III were done for preparing the initial source code. The tool could detect three program changes except for the initial preparation. CID[8], CID[6], and CID[7] correspond to Move Method of `X()`, Rename Method of `X()`, and Change Method Body of `Y()`, respectively. This agrees with the three modifications shown in Fig. 1.

In Table III, the "Aggregation" denotes the results of aggregation. The predefined value of temporal distance for aggregation was one minute. In this case, CID[8], CID[6], and CID[7] were aggregated in this order. The resulting change was Move Method.

VI. EVALUATION EXPERIMENT

We conduct a simple evaluation experiment with the tool described in Section V. An experimental target is the edit operation history recorded in GUI game application development done by an undergraduate student studying Computer Science. The project has 27 Java source files. The total number of lines of code is 2,360. The number of the recorded editing operations is totally 15,925.

The following three aggregation policies are prepared in this experiment.

1M) The value of temporal distance is one minute
2S) The value of spatial distance is two steps
NC) No condition

Table IV
CHANGES DETECTED IN CASE STUDY A AND THEIR AGGREGATION.

CID	Starting	Ending	Type	1M	2S	NC
1	10:58:28	10:58:28	**CMB**	**CMB**	**CMB**	**CMB**
2	10:58:29	10:58:29	**CMB**			
3	10:58:45	10:58:45	**CMB**			
4	10:58:45	10:58:45	**CMB**			
5	11:09:56	11:09:56	**CMB**	**CMB**		
6	11:34:22	11:34:22	**CMB**	**CMB**		

Table V
CHANGES DETECTED IN CASE STUDY B AND THEIR AGGREGATION.

CID	Starting	Ending	Type	1M	2S	NC
1	15:46:35	15:46:45	**CMB**	**CMB**	**CMB**	**CMB**
2	15:46:51	15:46:51	**CMB**			
3	15:46:55	15:46:55	**CMB**			
4	15:46:56	15:46:56	**CMB**			
5	15:47:04	15:47:04	**CMB**			
6	15:47:05	15:47:05	**CMB**			
7	15:47:13	15:47:20	**CMB**			
8	15:47:28	15:47:29	**CMB**			
9	15:47:42	15:47:42	**CMB**		**CMB**	
10	15:47:45	15:47:47	**CMB**			
11	15:48:28	15:48:28	**CMB**			
12	15:48:30	15:48:30	**CMB**			
13	15:48:40	15:48:40	**CMB**			
14	15:48:40	15:48:40	**CMB**			

On the policy 1M, program changes whose temporal distance is no more than one minute are considered to be aggregated. On the policy 2S, program changes whose spatial distance is no more than two steps on AST links are considered to be aggregated. On the policy NC, consecutive program changes would be aggregated based on only the rules shown in Table II, without any restriction on temporal distance and spatial distance. Under these policies, we present three case studies: A, B, and C. These case studies show what program changes our tool can detect and aggregate.

A. Case Study A

This case study demonstrates the use of our tool in modifying the contents of a certain method. In these modifications, a method invocation and a condition of an `if`-statement within the method have been rewritten. With usual software development, every modification was not separated and must be enclosed in one development session. In other words, one change like **CMB** was detected.

On the other hand, our tool can detect fine-grained program changes for such modifications and aggregate them. Table IV lists the detected program changes and the results of aggregating them. The symbols 1M, 2S, and NC denote the three aggregation policies, respectively. In this case study, six changes could be detected and all of them were identified as **CMB**.

Looking at Table IV, we observed an over 10-minute time period between CID[4] and CID[5]. It is assumed that a request before CID[4] and one after CID[5] are clearly different. Manually investigating the details of all the 6 changes, CID[5] and CID[6] rewrote their respective conditions of `if`-statements whereas CID[1–4] among them were performed to satisfy a change request of "swapping the first actual parameter of a method invocation for the second one." We also observed an over 20-minute time period between CID[5] and CID[6]. This might be because these two changes are not needed to be done at the same time. In fact, they are independent with each other.

Table VI
CHANGES DETECTED IN CASE STUDY C AND THEIR AGGREGATION.

CID	Starting	Ending	Type	EL	1M	2S	NC
1	17:43:11	17:43:42	**AF**	A	**AF**	**AF**	**AF**
2	17:43:45	17:43:45	**AF**	B	**AF**	**AF**	**AF**
3	17:43:46	17:43:46	**AF**	C	**AF**	**AF**	**AF**
4	17:43:47	17:43:47	**AF**	D	**AF**	**AF**	**AF**
5	17:43:47	17:43:47	**AF**	E	**AF**	**AF**	**AF**
6	17:45:00	17:45:00	**AF**	F	**AF**	**AF**	**AF**
7	17:45:12	17:45:12	**CFN**	F			
8	17:45:12	17:45:12	**CFN**	F			
9	17:45:15	17:45:15	**CFN**	E	**CFN**	**CFN**	**CFN**
10	17:45:15	17:45:15	**CFN**	E			
11	17:45:16	17:45:16	**CFN**	E			
12	17:45:19	17:45:19	**CFN**	D	**CFN**	**CFN**	**CFN**
13	17:45:20	17:45:20	**CFN**	D			
14	17:45:30	17:45:30	**CFN**	C	**CFN**	**CFN**	**CFN**
15	17:45:30	17:45:30	**CFN**	C			
16	17:45:31	17:45:31	**CFN**	C			
17	17:45:35	17:45:35	**CFN**	C			
18	17:45:38	17:45:38	**CFN**	B	**CFN**	**CFN**	**CFN**
19	17:45:38	17:45:38	**CFN**	B			
20	17:45:44	17:45:44	**CFN**	D	**CFN**	**CFN**	**CFN**
21	17:45:44	17:45:44	**CFN**	D			
22	17:46:28	17:46:28	**CFN**	A	**CFN**	**CFN**	**CFN**
23	17:46:29	17:46:29	**CFN**	F	**CFN**	**CFN**	**CFN**
24	17:46:31	17:46:31	**CFN**	E	**CFN**	**CFN**	**CFN**
25	17:46:33	17:46:33	**CFN**	D	**CFN**	**CFN**	**CFN**
26	17:46:34	17:46:34	**CFN**	C	**CFN**	**CFN**	**CFN**
27	17:46:35	17:46:35	**CFN**	B	**CFN**	**CFN**	**CFN**

On the aggregation policies 2S and NC, all program changes are aggregated into one **CMB**. These results do not present an evidence for benefits of our tool. On the other hand, the tool could aggregate CID[1-4] with the aggregation policy 1M into one **CMB**. Moreover, it could separate both CID[5] and CID[6] from CID[1-4], and could identify CID[5] and CID[6] as independent changes. This result shows that temporal aggregation well distinguishes the overall intents of changes. Each cluster of the changes might help a developer understand the evolution of the modified method.

B. Case Study B

This case study also deals with modifications of the contents of a certain method. Table V shows the detected program changes and the results of aggregating them. A total of 14 changes were done in a very short time. In these modifications, two `if`-statements have been added. One of them was added to the top block of the body of the modified method, and the other was added to a block enclosed by a `for`-statement existing in the same top block.

With our manual investigation, the former addition of `if`-statement was done under CID[1–8] while the latter addition was done under CID[9–14]. All the 14 changes were aggregated into one **CMB**, resulting from the application of the aggregation policies 1M and NC. In other words, temporal aggregation was not useful for distinguishing changes having their different intents. On the other hand, the aggregation policy 2S could divide the changes into two clusters that were identified as their respective **CMB**s. The clusters CID[1–8] and CID[9–14] conform closely to two intents within the modifications. These results exemplify spatial aggregation rather than temporal aggregation works well in understanding the evolution of the modified method.

C. Case Study C

This case study deals with modifications related to a field not a method. A total of 6 fields were added to the same class. These fields were not independently written but were created through the duplication by copy-and-paste operations. Only the first field was newly added by typing and the remaining 5 fields were derived form the first one. Specifically, the names of the 5 fields were modified into their suitable ones.

Table VI shows the detected program changes and the result of aggregating them. The "EL" indicates which program element was changed. For example, CID[1] and CID[22] modified the code constituting the field labeled with `A`.

Referring to Table VI, the changes for each of the fields were not applied sequentially. Instead, such changes were interleaved with each other. A total number of the changes is 27. They were divided into 6 **AF**s in the copy-and-paste phase and 21 **CFN**s in the renaming phase. The results of aggregation under the three policies are exactly the same. The aggregation with respect to **AF** might be reasonable since the 6 **AF**s were applied to their respective fields.

The aggregation with respect to **CFN** conforms the implementation of the three policies. However, the created clusters can provide a little help for developers since changes of the same field were scattered around the renaming phase in the modifications. Troublesome distinguishment of program changes leaves for maintainers. This reveals the limitation of our aggregation based on the heuristic rules, temporal distance, and spatial distance. Our aggregation algorithm is still simple and thus its effectiveness strongly depends on the original order of program changes actually performed. To overcome this kind of limitation, a sophisticated algorithm such as aggregation considering dependences of changes would be designed and implemented.

VII. Conclusion

This paper presented a novel mechanism to detect individual program changes, which might assist maintainers to comprehend the evolution of the program. It restores two versions of source code from edit operations done in the past, and analyzes class information extracted from the restored source code. It can also aggregate the detected changes based on the heuristic rules having two options with respect to temporal distance and spatial distance. We made an experiment with a tool that was built for demonstrating the applicability of the detection mechanism. Three case studies in the experiment present the use of detected and aggregated program changes. Unfortunately, there is still no solid evidence that these changes could be truly useful for comprehending the evolution of the program.

We are going to explore other mechanisms detecting fine-grained code changes based on differences between ASTs. For this, ChangeDistiller [8] is informative. We are also going to improve the aggregation algorithm so that it can summarize program changes in various ways. Moreover, we plan to conduct further experimental evaluation with realistic software development projects. The advanced representation of detected and aggregated program changes will be tackled. The concept of generating human-readable documentation for program changes [9] would be adopted.

Acknowledgment

This work was partially sponsored by the Grant-in-Aid for Scientific Research (C) (24500050, 24700034) from the Japan Society for the Promotion of Science (JSPS).

References

[1] R. Robbes and M. Lanza, "A change-based approach to software evolution," *Electronic Notes in Theoretical Computer Science*, vol. 166, pp. 93–109, 2007.

[2] K. Herzig and A. Zeller, "The impact of tangled code changes," in *MSR'13*, 2013, pp. 121–130.

[3] R. Robbes and M. Lanza, "SpyWare: A change-aware development toolset," in *ICSE'08*, 2008, pp. 847–850.

[4] L. Hattori, M. D'Ambros, M. Lanza, and M. Lungu, "Software evolution comprehension: Replay to the rescue," in *ICPC'11*, 2011, pp. 161–170.

[5] T. Omori and K. Maruyama, "A change-aware development environment by recording editing operations of source code," in *MSR'08*, 2008, pp. 31–34.

[6] B. G. Ryder and F. Tip, "Change impact analysis for object-oriented programs," in *PASTE'01*, 2001, pp. 46–53.

[7] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, "Chianti: A tool for change impact analysis of java programs," *ACM SIGPLAN Not.*, vol. 39, pp. 432–448, 2004.

[8] B. Fluri, M. Wüersch, M. Pinzger, and H. C. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE TSE*, vol. 33, no. 11, pp. 725–743, 2007.

[9] R. P. Buse and W. Weimer, "Automatically documenting program changes," in *ASE'10*, 2010, pp. 33–42.