Lightweight Operation History Graph for Traceability on Program Elements

Takayuki OMORI† a), Katsuhisa MARUYAMA†, and Atsushi OHNISHI†, Members

SUMMARY History data of edit operations are more beneficial than those stored in version control systems since they provide detailed information on how source code was changed. Meanwhile, a large number of recorded edit operations discourage developers and researchers from roughly understanding the changes. To assist with this task, it is desirable that they easily obtain traceability links for changed program elements over two source code snapshots before and after a code change. In this paper, we propose a graph representation called Operation History Graph (OHG), which presents code change information with such traceability links that are inferred from the history of edit operations. An OHG instance is generated by parsing any source code snapshot restored by edit histories and combining resultant abstract syntax trees (ASTs) into a single graph structure. To improve the performance of building graph instances, we avoided maintaining every program element. Any program element presenting the inner-structure of methods and non-changed elements are omitted. In addition, we adopted a lightweight static analysis for type name resolving to reduce required memory resource in the analysis while the accuracy of name resolving is preserved. Moreover, we assign a specific ID to each node and edge in the graph instance so that a part of the graph data can be separately stored and loaded on demand. These decisions make it feasible to build, manipulate, and store the graph with limited computer resources. To demonstrate the usefulness of the proposed operation history graph and verify whether detected traceability links are sufficient to reveal actual changes of program elements, we implemented tools to generate and manipulate OHG instances. The evaluation on graph generation performance shows that our tool can reduce the required computer resource as compared to another tool authors previously proposed. Moreover, the evaluation on traceability shows that OHG provides traceability links with sufficient accuracy as compared to the baseline approach using GumTree.

key words: graph format, program change summary, fine-grained changes, program change understanding, software evolution

1. Introduction

In these years, many studies on recording and leveraging developers’ operations on integrated development environments (IDEs) were conducted for better understanding of software evolution [1], [2]. Along this line, we developed an Eclipse plug-in called OperationRecorder [3], which records operations performed on the Eclipse’s Java editor. The recorded operation history contains not only edit information, which indicates how each source file was changed, but also developers’ interactions with the IDE (i.e., tool use). Using the history, developers and researchers (called users hereafter) can retrospectively obtain a past source code snapshot (a state of a source file) at an arbitrary time. This helps users to understand detailed program changes in software development and maintenance activities.

With respect to practical use of operation history, recorded edit operations should be associated with changed program elements (i.e., nodes in an abstract syntax tree (AST)). A changed program element associated with a certain edit operation can be inferred using the offset value (position in the source file) where the edit was performed [3], [4]. However, change reification remains insufficient to explicitly present the inferred association links connecting the changed elements before and after code changes. Here, such an association link is called a traceability link. To help users effectively understand source code changes, a traceability link as a straightforward representation of changes on program elements is strongly required. A traceability link presents a change (e.g., rename, move, and signature modification) of a program element. Traceability links can be used to accurately judge if a program element before and after code changes is the same one or not. Important usage scenarios of those links include detecting and understanding refactoring transformations and exploring the origin of each code fragment. However, when a program element was repeatedly moved and/or renamed, the traceability of the program element tends to be lost, and users cannot precisely estimate what refactoring transformations were applied. Accurate traceability links can ameliorate such a problem.

To this end, we proposed a graph-based format called OpG2 for representing change history of abstract program structure [5], [6]. The proposed graph combines multiple simplified ASTs (SASTs) derived from every code snapshot restored from edit operations, and contains links that present changes on each program element. Each SAST eliminates detailed information on program elements inside methods. This elimination is reasonable since our proposed graph is focused on quick understanding of the overall source code changes at the syntax level. If users want to comprehend the change details of program elements that are targeted through the quick understanding process, they can use a replay tool such as OperationReplayer [7], which facilitates investigating correspondent edit operations.

OpG2 has a prominent feature on its static analysis. In general, compiling a restored snapshot requires analyzing references to program elements in other classes. This semantic analysis involves several source files related to the restored snapshot. Thus, its process is heavy although an edit operation only affects a single source file. To avoid this, OpG2 employs a lightweight semantic analysis method

Manuscript revised November 15, 2020.
Manuscript publicized December 15, 2020.
†The authors are with College of Information Science and Engineering, Ritsumeikan University, Kusatsu-shi, 525–8577 Japan.
a) E-mail: tomori@is.ritsumei.ac.jp
DOI: 10.1587/transinf.2020EDP7129

Copyright © 2021 The Institute of Electronics, Information and Communication Engineers
that infers a fully qualified type name from an apparent one (simple type name) described within source code by finding its corresponding import declaration. This inference promotes the correct discrimination of different types with the same apparent name without consuming a lot of memory resource.

Unfortunately, the graph representation of OpG2 still has a serious problem with regard to the performance in building its instances. An OpG2 instance contains many SASTs since each edit operation derives a source code snapshot that produces a SAST. In general, almost all program elements are unchanged between two successive SASTs before and after an edit operation is performed. Nevertheless, the graph representation of OpG2 requires storing every program element. Thus, it consumes a lot of memory and disk space. Moreover, an OpG2 instance must be treated as a monolithic tree. Therefore, a large graph instance must be loaded at once and may use up the computer’s memory resource.

To overcome these problems of OpG2, we propose Operation History Graph (abbreviated as OHG) in this paper. OHG inherits OpG2’s benefit but improves its graph building performance and traceability (correctness of traceability links between changed program elements). OHG combines multiple SASTs as well as the conventional graph but does not contain all nodes within the original SASTs. OHG omits nodes corresponding to unchanged program elements from a SAST after the change. Moreover, it supports partial loading of graph instances by assigning a unique identification (ID) to each node and edge. The memory region that stores a part of an OHG instance can be immediately released after its use. Thus, the maximum memory consumption to generate and manipulate a graph can be limited. Experimental results with seven Java projects demonstrate OHG attained a large reduction in the number of program elements and the shortening of the consumed memory size and disk space to generate its instances. Particularly, the memory size reduction makes OHG suitable for practical use with a standard PC although the execution time increased.

Next, we should mention the correctness of traceability links between program elements within snapshots before and after a change. Here, the traceability of a program element means a feature that the element can be identified as the same one over ASTs before and after an edit operation that changes the element. Even with existence of edit operations, it is hard to correctly detect traceability links from them. For example, Coding Tracker [4] cannot detect moving a program element and may result in loss of a necessary traceability link related to the move. OpG2 exploits rename operations that a user explicitly performed to generate traceability links. However, the correctness of traceability links was still limited. To achieve a high correctness of traceability in OHG, we also leverage heuristic change patterns based on recorded edit operations to detect the move and rename actions of program elements. Experimental results show that traceability links of OHG achieve a high accuracy.

The contributions of the paper are as follows:

1. It presents a graph format that represents the history of source code changes to help developers and researchers easily trace changes on program elements.
2. It presents a lightweight graph generation method, including both application of node omission techniques and type inference based on import statements. Experimental results show that it can reduce the number of program elements in graph instances and memory consumption in graph generation as compared to a conventional approach.
3. It presents a method to generate accurate traceability links. Experimental results show the correctness of traceability links in the graph instances.

The rest of this paper is structured as follows: Sect. 2 gives detailed background of this study and related works. Section 3 describes the representation of our proposed graph. Section 4 explains implementation of our tools to generate and read graph instances. Section 5 shows evaluation experiments. Section 6 discusses on the effect of our method and threats to validity. Section 7 concludes the paper with a short summary and future works.

2. Background and Related Works

This section explains the background of our study and related works.

2.1 Operation Histories and Change Understanding

In both software engineering research and business software development, revision histories of source code that can be obtained from version control systems (VCSs) are deemed as the de-facto standard of software change data. Such a revision history contains source code snapshots but not code changes that developers actually performed. Therefore, a code change is calculated as a source code difference (e.g., added/deleted line) between two snapshots of successive revisions of the same file.

Problems of using VCSs in studying software evolution has been pointed out by several researchers [1], [4], [8], [9]. For example, multiple code changes under different intentions are often committed together into a single revision. Moreover, several changes are sometimes overwritten by other changes before their commit. This makes a part of code changes eternally lost from the change history. These problems prohibit developers from correctly understanding how the source code was changed.

To address the above issues, several methods have been proposed to record fine-grained code changes that a developer actually performed [11]. Even with such code change histories, supporting change understanding is still insufficient. For example, some tools directly record changes in Java programs as a sequence of changes in program elements [4], [10]. They have a flaw that exact source code representations cannot be restored with their histories.
Moreover, direct recording of changes in program elements does not guarantee a high traceability on their changes. Other tools record fine-grained textual changes in source code [3], [11], [12]. These tools are suitable for restoration of past source code snapshots, but users cannot obtain high-level change histories without further analyses of snapshots. Moreover, all of these existing tools simply treat change histories as a single sequence of change events, making it difficult to trace change history of a program element. To help users effectively understand code changes, traceability links which present changes of program elements are highly required.

To precisely understand recorded code changes, we can replay recorded code changes using replay tools [7], [13]. However, a history investigation by replaying each operation is time-consuming. Although several recent replay tools provide kinds of replay-support functions, such as filtering and navigation, they are still insufficient to quickly grasp abstract software change history (e.g., when a program element was created, deleted, and renamed). We consider that grasping such an abstract history before replaying allows users to identify a part that should be investigated carefully. Thus, the users’ effort of replay can be reduced.

2.2 Conventional Element Matching

Detecting traceability links of changed program elements in software evolution data is traditionally considered as an important problem. Many studies on program element matching technique were conducted to obtain traceability links [14]–[16]. In these years, GumTree [17] (abbreviated as GT) is widely used to generate edit scripts representing AST changes. Moreover, several researchers have tried to improve GT to obtain more accurate and comprehensive edit scripts [18]–[21]. Higo et al. [22] successfully improved the traceability of Java methods which is based on Git mechanisms.

All of the above studies depend on revision histories in VCSs instead of observing developers’ edit operations. This may incur loss of traceability on program elements. Torre et al. [23] reported that 27% cases where GT produces are incorrect or sub-optimal results. GT sometimes generates imprecise edit scripts because of mismatches or spurious matches of program elements. Hora et al. [24] assessed the threat of untracked changes in software evolution. In their experiment, a refactoring detection tool was used to estimate applied refactoring transformations with revision histories. The refactoring kinds include renaming, moving, and extracting program elements (i.e., class and method). As the result, they found the change history of a program element is sometimes split by a refactoring transformation. Particularly, the change history of long-lived elements are often split.

2.2.1 Algorithm of GumTree

Since GT [17] was used as the baseline tool in our experiment, we briefly explain its algorithm. GT compares two ASTs and finds matched AST nodes presenting the same program element. The comparison is composed of top-down and bottom-up phases. The former finds isomorphic subtrees which are common in two ASTs. The latter finds matched nodes that the result of the top-down phase does not contain. This process starts from the matched subtrees and compares extended ones with a dice function. When it finds new matched nodes, nodes with the same label in their subtree are also deemed matched. Finally, GT outputs an edit script calculated by RTED algorithm [25]. The script is composed of an insertion, deletion, move, and update of a program element. Program elements with the same label (i.e., kind of a program element) may be matched with each other even if they have different values (i.e., actual tokens). If matched nodes have different parents, they are deemed moved. If they have the same parent and different values, they are deemed updated.

2.3 Lightweight Parsing for Code Snapshots

Alexandru et al. [26] proposed a tool called LISA, which can reduce the number of AST nodes derived from multiple source code snapshots in VCS data. LISA generates syntax trees for multiple revisions. Generally, when syntactic analysis for a long change history is performed, many ASTs must be generated and much memory is required. LISA can save memory by omitting unmodified AST nodes. Although our study is focused on snapshots derived from edit histories instead of VCS data, we diverted the main idea of LISA into our study. Our proposed method omits unchanged program elements from the resultant graph. Since edit operations are much more frequent than commits, the omission technique is crucial to show the feasibility of our graph generation. Clearly, the node omission technique itself is not novel. However, our proposal is the first attempt in both applying it to edit histories and evaluating its result, to our knowledge.

2.4 Graph Representation for Edit History

In a previous paper [13], we proposed OperationSliceReplayer. This tool generates a slice of an operation history with a graph representation called OpG. It can reduce the number of operations that should be replayed in investigating an operation history by generating a slice based on a focused program element. An OpG instance is composed of class members (i.e., fields and methods) and operations for them. OpG is beneficial for generating operation history slices and improving replay efficiency. However, OpG loses a part of a program element’s traceability since OpG cannot present a top-level class rename. Moreover, OpG’s edges are designed to gather edit operations related to a particular element, not for roughly understanding the change history of a program element.

We proposed OpG2 in our previous papers [5], [6]. In our experiment with OpG2, graph generation failed due to
an out-of-memory error when the operation history was not small. In addition, the correctness of traceability links of OpG2 was still not fully evaluated although we showed a small example of generating a program change summary in those previous papers. To cope with this, we improved OpG2 and made OHG as mentioned in Sect. 1.

3. Operation History Graph

This section formalizes OHG and explains its representation.

3.1 Unique Name

Any program element in an OHG instance has its unique name. Even in the same development project, a simple name may conflict with other program elements. Hence, OHG represents each type name in a fully qualified format including its package name. Each field and method is presented by a signature containing its type. To cope with overloading, a method’s signature contains parameter types. Thus, each element is recognized by its unique name. In this paper, a unique name contains its fully qualified type name to prevent name conflict. The followings are examples of unique names.

- The unique name of class C in package p is p.C.
- The unique name of field f whose type is int in C is p.C#f:int.
- Consider a method named m declared in C. The method has parameters whose types are String and Object in this order, and has no return value. In this case, the unique name of the method is p.C#m(java.lang.String,java.lang.Object): void.

A unique name is unique at a certain timestamp. However, if an element was deleted and another element whose signature is the same as former one is added later, those elements have the same unique name. As a result, an OHG instance sometimes has multiple nodes with the same unique name.

3.2 Graph Elements

An OHG instance consists of the following nodes and edges. Every node and edge has a single unique ID. The ID is a randomly generated alphabetic string with a designated length (default is 10), but the ID of the origin node is always composed of only ‘a’. An OHG instance initially has only a single origin node, a single null node, and an edge between them.

(1) Nodes

- The origin node is used as the origin of tracing the graph elements. A graph instance has only one origin node.
- The null node is used to present deleted program elements. A program element which has an edge to the null node is treated as a deleted one. A graph instance has only one null node.
- A Java compilation unit (jcu) node corresponds to a source file, which has an attribute for its file path.
- A type declaration (typeDecl) node represents a declaration of a type, which has an attribute for its unique name.
- A member declaration (memDecl) node represents a declaration of a member in a type declaration. It has an attribute for its unique name.
- A type reference (reference) node represents a type reference. It has an attribute for the unique name of the referred element.

Each node has the following attributes.

- The timeFrom and timeTo attributes indicate the lifetime of the element.
- The type attribute indicates the detailed type of the program element. For example, as for the type declaration node, the candidates of the value of this attribute are class, interface, and enum.

Here, we explain omission of nodes in OHG instances. In general, OHG must treat a bunch of code snapshots due to the frequency of developers’ edit operations. Therefore, OHG adopts two kinds of node omission techniques:

(a) Inner-method omission

The first one omits the inner-structure of methods. Since OHG is intended for quick understanding of change histories, detailed parts of each SAST are omitted while significant parts remain. As elements of a SAST, we selected a compile unit (corresponding to a source file), declarations of types (class, interface, and enum), their members (field, enum-constant, and method), and type references.

Actually, we also added references to methods and fields in the previous version of OHG [5]. However, we eventually omitted them since its name resolving was imperfect. An edit history recorded by OperationRecorder only contains source file contents and their changes. Because of the polymorphism mechanism of Java, it is hard to correctly identify the target of reference (e.g., invoked method) from the history. Fortunately, import statements in a restored source code can be used to resolve type names. In our preliminary experiment, almost all type name references are correctly resolved with import statements. If a developer uses on-demand import (using wildcard character ‘*’), the resolving process fails. However, it rarely appears in our histories since it is a non-recommended development custom. Therefore, we selected type references as a significant element of a SAST, whereas references to other elements are omitted.
(b) Non-changed element omission

The second one omits non-changed program elements as mentioned in 2.3. Each program element in a graph has \texttt{timeFrom} and \texttt{timeTo} attributes, which can present the lifetime of the element. By using these attributes, OHG can omit nodes for non-changed elements without loss of information.

(2) Edges

An edge has the following attributes and connects two nodes. Note that every node in a graph must be connected to at least one other node. That is, no node is isolated.

- The \texttt{from} and \texttt{to} attributes present the IDs of the source and destination of the edge, respectively.
- The \texttt{type} attribute indicates the type of the edge. The same type of edge between the same two nodes cannot be generated. The possible types are as follows:
  1. A \texttt{null} edge is generated between the origin node and the null node to ease accessing to the null node.
  2. A \texttt{root} edge connects from the origin node to the root of each SAST (i.e., Java compilation unit).
  3. A \texttt{child} edge forms the tree structure of SAST. For example, a \texttt{jcu} node usually has a \texttt{child} edge to \texttt{typeDecl} node for a class declared in the file. The \texttt{typeDecl} node has \texttt{child} edges to its members (e.g., fields and methods).
  4. A \texttt{traceability} edge (i.e., traceability link) connects nodes of the same program element in successive SASTs if its unique name was changed due to a rename or move. If a program element is deleted, a traceability edge is generated from the node to the null node. When a node is deleted, moved, or renamed, traceability edges for its descendents are not explicitly generated since their changes are apparent by virtue of the ancestor's traceability edge.

Figure 1 shows an example of an OHG instance. The \texttt{origin} node and \texttt{null} node are connected by the \texttt{null} edge. This graph has two SASTs (/p/C.java and /p/D.java). Nodes under the \texttt{jcu} node of /p/D.java are omitted in this figure. A SAST is formed with \texttt{child} edges. The \texttt{typeDecl} node of p.C has four child \texttt{memDecl} nodes. In this case, two of them are connected with a traceability edge. This means that p.C initially had three members (p.C#d:int, p.C#e:Type, and p.C#f:int) and that p.C#f:int was renamed into p.C#g:int. Moreover, the other traceability edge indicates p.C#g:int was deleted after the rename operation. The corresponding node's \texttt{timeFrom} and \texttt{timeTo} attributes indicate the exact time of the rename and deletion although they are omitted in the figure.

3.3 Graph Representation in XML

A graph instance is stored in XML files. Each node and edge is stored as a single XML element under a path corresponding to its ID. For example, a \texttt{jcu} node whose ID is aBcDeFgHiJ is stored as

```
<a><B><c>...<J><jcu>...</jcu></J>...</a>
```

in a.xml in a specified directory. When the number of XML elements corresponding to SAST nodes in a single file exceeds the specified threshold (default is 2), the file is separated. In this case, the initial part of the ID is used as a directory name (e.g., a/B.xml).

All elements have \texttt{id} attribute and two child elements (\texttt{inbounds} and \texttt{outbounds}). The \texttt{inbounds} and \texttt{outbounds} elements have the IDs of inbound and outbound edges, respectively. The \texttt{origin} and \texttt{null} elements only have those common attribute and elements. A \texttt{jcu} element also has file and \texttt{package} attributes. A \texttt{typeDecl}, \texttt{memDecl}, and \texttt{reference} elements have \texttt{fqn} attribute. The \texttt{fqn} attribute of a declaration node indicates the unique name of the element. The \texttt{fqn} attribute of a \texttt{reference} node indicates the unique name of its referred element.

Because of this data structure, data access can be conducted on demand. When a graph-reading tool accesses an OHG instance, it first reads the \texttt{origin} with the designated ID. Then, the tool reads each node by tracing the outbound edges (or inbound edges for tracing in reverse) using their IDs.

4. Tools

This section explains our implemented tools: \texttt{OHG Builder} and \texttt{OHG Reader}. Figure 2 shows the system configuration including our proposed tools. OperationRecorder[3] records developers’ textual edits and other operations on Eclipse and outputs operation history files. \texttt{OHG Builder} reads operation history files and converts them into an OHG instance. \texttt{OHG Reader} reads the graph and outputs a history summary according to the user’s commands. The summary shows the history of change events (addition, deletion, rename, and move) on program elements. By using OHG, such a summary can be easily generated since the rename and move are directly presented by traceability links. The time of an addition and deletion can be also obtained by the \texttt{timeFrom} and \texttt{timeTo} attributes of each node. The output summary can be used for the user’s quick understanding of
the change history.

4.1 OHG Builder

OHG Builder builds a graph instance from an edit history. OHG Builder reads the operation history files specified by a user and automatically generates a graph. It executes the following nine steps:

Step 1 Input. First, the tool reads the input operation history files. Operations are sorted in the order the developer actually performed.

Step 2 Graph initialization. The origin node and the null node are created.

Step 3 File list generation. The file list contains all source file names in the operation history.

Step 4 Package information generation. Package information is a HashMap object that associates a package name with a list of class names within the package.

Step 5 Traceability data generation. The traceability data contain information about a rename and move of a program element that was conducted by automated refactoring functions Eclipse provides. Renaming and moving are detected based on operation patterns within the change history. Details are explained in 4.1.1.

Note that the traceability data do not contain a rename or move conducted by only normal edits in the source code text (e.g., renaming by deleting a part of the old name and inserting a part of the new name). They are detected in Step 9e.

Step 6 Cut-and-paste data generation. This step generates data about cut-and-paste operations that may include a move of program elements. In this step, a pair of cut and paste operations which treat the same string (except for white space characters) is identified.

Step 7 File history generation. Change histories for each file are generated. First, the operation history is separated into sequences for each file. Then, the two parts which correspond to before and after a file rename are concatenated. Here, the rename is identified based on the traceability data in Step 5. Moreover, a file history is separated if it contains a file deletion. Thus, when a file was deleted and another file with the same name is created later, the histories of those files are distinct from each other. In this step, compound operations are disassembled since they may include changes to multiple files.

Step 8 History simplification. The following three processes are performed in this order. (1) Operations that do not affect source code contents are removed. (2) Successive edit operations which are performed in a neighbor text region are merged. This process reduces needless operations (e.g., typos) in the history. (3) Operations recorded with the same timestamp are grouped into a single compound operation since each code snapshot in such operation sequences is clearly redundant.

Step 9 SAST comparison and graph generation. By processing each operation in the history chronologically, graph elements are generated as follows:

a. A code snapshot of the time when the current operation was performed is restored.

b. The code snapshot is parsed to build an AST.

c. From the AST, declaration nodes in the corresponding SAST are generated. To determine the unique name for each node, package information generated in Step 4 is used. The detailed process of resolving a type name within the unique name is explained in 4.1.2.

d. Type reference nodes are appended to the SAST.

e. Successive SASTs are compared to generate SAST differences. Details are explained in 4.1.3. In this process, each graph element is output (or updated) into XML files. The file format is mentioned in 3.3. Here, new nodes and edges are appended into the graph instance, and the timeFrom and timeTo attributes of existing nodes are updated, if needed.

4.1.1 Detecting Renames and Moves

This section explains details of detecting a rename and move of a program element in Step 5. An operation history
recorded by OperationRecorder can present an activation of an automatic refactoring function as a **MenuOperation** element. However, details of renaming and moving do not always appear in an intuitive or straightforward form within the recorded history. Therefore, we used a heuristic approach; we conducted rename and move operations, and subsequently identified operation patterns for them.

1) Detection of a move

The following two operation patterns are used to detect a move of a program element.

**Pattern 1:** When a developer moves a source file in a package into another package with the Eclipse's move function, the following operations can be observed.

1. **MenuOperation** (with a label containing "move.element") for the old file.
2. **Empty NormalOperation** (without inserted or deleted strings) for the new file.

If this pattern is detected, **OHG Builder** recognizes the old file was moved to the location of the new file.

**Pattern 2:** When a developer moves a program element by text-editing in the editor, the following operation can be observed.

1. **CompoundOperation** whose label contains "Move".

   The compound operation contains the actual string replacement corresponding to the move. Therefore, the names before and after the move can be restored.

2) Detection of a rename

The following pattern is used to detect a rename of a program element.

**Pattern 3:** When a developer uses the Eclipse's rename function, the following operations can be observed.

1. **MenuOperation** whose label contains "rename".
2. The first non-empty **CompoundOperation** within 10 seconds from the above operation.

   A time restriction is attached to the second operation. This is because the developer may cancel the rename after the first menu operation. If the rename is canceled, the second operation does not appear. This compound operation itself can be observed due to another kind of operation, such as code completion. To reduce false positives, a compound operation within 10 seconds after the menu operation is deemed as a rename operation. When the compound operation is a genuine rename, it contains both names before and after the rename.

4.1.2 Type Name Resolution

**OHG Builder** uses Eclipse JDT in syntax and semantic analyses. For correct type name resolution, it requires path information of source files and class files referred in the program. Unfortunately, operation histories recorded by OperationRecorder do not contain such information. Therefore, we implemented our own name resolution function. To resolve type names, we use import statements in the restored code snapshots. In addition, package information generated in Step 4 is required to correctly identify classes within the same package since they can be accessed without an import statement.

Details of the type name resolution processes are:

- If Eclipse JDT resolves the type binding and the binding originates in source code, the result is adopted. In this case, all the type names declared in the same source file are correctly resolved.
- Otherwise, import statements are used. Most class names are correctly resolved since import statements include fully qualified names.
- Otherwise, package information is used. The classes in the same package are correctly resolved.
- Otherwise, the result from Eclipse JDT is adopted. The class names in java.lang are resolved. If resolving fails, the simple type name (appearing in source code as-is) is used as a fully qualified type name.

4.1.3 SAST Comparison

The comparison starts from the SAST's root, and it is based on the unique name of each node. Any node dangling unmatched nodes is deemed unmatched. Let \( s_{f(n)} \) be the \( n \)-th SAST of source file \( f \). A node that exists in \( s_{f(n)} \) but not in \( s_{f(n+1)} \) is deemed to be deleted. In this case, an edge from the deleted element to the null node is generated. A node that does not exist in \( s_{f(n)} \) but exists in \( s_{f(n+1)} \) is deemed to be added. In this comparison, the order of child nodes is not considered. Hence, reordering of the program elements is ignored.

After that, the tool reviews the result for renaming and moving, including cut-and-paste operations. The tool detects a rename using the rename information generated in Step 5. The renamed element itself is deemed to be renamed (linked with a traceability edge). This procedure is also applied to move information. Cut-and-paste data generated in Step 6 are also used to detect a move of elements. When a rename or move is detected, an edge heading to the null node is removed and a traceability edge corresponding to the rename or move is generated.

Moreover, using edit operations performed between \( s_{f(n)} \) and \( s_{f(n+1)} \), element matching is conducted. If a part of the signature of a deleted element in \( s_{f(n)} \) remains in the added element in \( s_{f(n+1)} \), the tool deems it as a rename operation\(^1\), and generates the corresponding traceability edge.

\(^1\)A rename also means a change in the element's signature after the SAST comparison step.
Table 1 Commands of OHG Reader.

| Cmd. | Description |
|------|-------------|
| c    | Shows children of the focal node. |
| c i  | Moves to the i-th child node. |
| p    | Moves to the parent node. |
| g n  | Moves to the element with unique name n (first declaration). |
| g / n| Moves to the element with unique name n of the time t. |
| t    | Traces the outbound traceability edge if possible. |
| r    | Traces the inbound traceability edge if possible. |
| ev p | Shows the change events in the time period p. |
| ev t1t2 | Shows the change events between t1 and t2. |
| simple | Switches to the Simple Mode. |
| fqn | Switches to the FQN Mode. |

† Root edges are treated in the same way as child edges.

4.2 OHG Reader

We implemented a CUI-based graph reading tool called OHG Reader. OHG Reader reads an OHG instance. It has a single focal node, which is initially set to the root node. Similar to the tool (Graph Tracer) we proposed in [6], OHG Reader receives a user’s commands. For example, if a user hits the enter key (empty command), the tool shows the current focal node as shown below:

type=class from=2012/05/29 15:56:50 to=2012/05/29 20:41:24 name=reversi.Reversi

This means the focal node corresponds to class reversi.Reversi whose lifetime is between 2012/05/29 15:56:50 and 2012/05/29 20:41:24. Some other commands are shown in Table 1. If a user hits the enter key (empty command), the tool shows the current focal node as shown below:

4.2 OHG Reader

We implemented a CUI-based graph reading tool called OHG Reader. OHG Reader reads an OHG instance. It has a single focal node, which is initially set to the root node. Similar to the tool (Graph Tracer) we proposed in [6], OHG Reader receives a user’s commands. For example, if a user hits the enter key (empty command), the tool shows the current focal node as shown below:

type=class from=2012/05/29 15:56:50 to=2012/05/29 20:41:24 name=reversi.Reversi

This means the focal node corresponds to class reversi.Reversi whose lifetime is between 2012/05/29 15:56:50 and 2012/05/29 20:41:24. Some other commands are shown in Table 1. If a user hits "g reversi.Reversi" (g stands for go to), the focal node moves to the first declaration of reversi.Reversi. In the table, t, t1, and t2 are time. The time can be specified in the format YYYY/MM/DD hh:mm:ss. YYYY, MM, DD, hh, mm, and ss denote the year, month, day, hour, minute, and second, respectively. The hour, minute, and second can be omitted to present a specific time period for "ev p" command. For example, if a user hits "ev 2012/05/29 20", the tool shows the list of change events related to all nodes in the subtree of the focal node between 20:00:00 and 21:00:00 on the specified day. Figure 3 shows an example of the result of the ev command. It shows a change event list of 2012/05/29. Lines with INS and DEL indicate insertions and deletions of a program element, respectively. The result of a rename (REN) contains both names before and after the rename.

The tool shows the summary in two modes: (1) FQN Mode: every type name is shown with its FQN. (2) Simple Mode: every type name is shown with its simple name. In the FQN Mode, unique names shown in the command results are precise, but quite long. Users can use also the Simple Mode to view results with unqualified element names.

If a user wants to trace changes in a program element, he/she can select the element by navigation commands, such as g, and then he/she can trace its change by repeating the t command. The results show the transit of the element's name changes, which contain its move and rename. Figure 4 is an example (to attribute is omitted). It shows changes of a unique name of method PlayerPanel().

5. Experiments

This section presents experiments for evaluating the feasibility of graph generation and accuracy of traceability links on program elements compared to an existing tool. Research questions (RQs) are as follows:

RQ1 (Feasibility): Can OHG Builder reduce the required computer resource to build graph instances so that a prevailing personal computer (PC) can generate them?

RQ2 (Traceability): Can OHG Builder provide traceability links with sufficient accuracy as compared to the baseline approach using GT?

5.1 Feasibility Evaluation

To demonstrate the feasibility of OHG Builder in graph generation, we conducted a comparative study. In this paper, the feasibility means that a graph instance can actually be generated with prevailing PCs. As of 2020, we assumed the standard specifications of such PCs are 16GB of RAM and 500GB of secondary storage (HDD or SSD).

5.1.1 Experiment Procedure

The experiment to evaluate the feasibility was conducted in the following procedure.

(1) Data preparation

Since operation history data in software development still
do not prevail at this point, we used operation histories recorded in in-house software development projects in the experiment. Each project was completed by a single developer. The leftside of Table 2 shows the kind of application (App.), number of operations (NOP), development hours, and total lines of code (LOC) of the final snapshot of all files in each project.

(2) Graph generation

With the prepared histories, we built graph instances of OpG2 and OHG and measured the number of elements, heap size in graph generation, output file size, and required time for graph generation.

An OpG2 instance was generated by Graph Builder [6] that was slightly modified. The original OpG2 includes edit operations and code snapshots, whereas OHG omits them since they can be easily obtained from operation history files that OperationRecorder generates. Therefore, we modified Graph Builder not to generate those nodes to conform to the element kinds of OHG.

As mentioned in 2.4, memory resource was exhausted in generating an OpG2 instance in our former experiment. Therefore, we used a memory-abundant workstation (called WS) for OpG, and set the maximum heap size (option of Java virtual machine) to 50GB. On the other hand, OHG instances were generated on both WS and a notebook PC (called NB). We set the maximum heap size to 50GB for WS to compare the result with OpG2’s. As for NB, the maximum heap size was set to 4GB to show that OHG instances can be generated with the limited resource on a currently prevailing computer.

The specifications of WS and NB are shown below:

WS: Xeon E5-2630v4 2.2GHz, RAM 64GB, HDD 1TB, Ubuntu 18.04, OpenJDK 8.
NB: Core i7-7700HQ 2.8GHz, RAM 16GB, SSD 500GB.

5.1.2 Results

The overall results are shown in the rightside of Table 2. The upper line of each project shows the result of OpG2. The middle and lower lines show the results of OHG with WS and NB, respectively.

(1) Number of elements (the column of # Elem)

The result shows the number of elements drastically reduced in OHG. In total, the number of OHG nodes is approximately 2.0% of OpG2’s.

(2) Heap size (the column of Heap)

Since memory consumption in the Java virtual machine is mostly occupied by the heap, we measured the maximum heap size during graph generation with a profiler called Java VisualVM[27].

Heap consumption for OpG2 in Project G exceeded the NB’s RAM size. In a real-use situation, other software also consumes memory resource, so that the memory consumption of several other projects (e.g., C and F) may also cause trouble in practice. In contrast, OHG limits the maximum heap consumption. It ranges between 3.3 and 4.9 when the heap size is set to 50GB (on WS). The reason why OHG consumes more heap in Projects A and B is that it generates many XPath instances to access each graph element. It is noteworthy that the heap size can be limited to much lower values when the heap size is set to 4GB (on NB). OHG instances were successfully generated with such a little memory consumption.

Figure 5 shows the heap size transition in graph generation for Project F which has the most textual operations and longest development hour. As the figure shows, the size
of used heap increased to near 1,000 MB and was kept under 500 MB throughout the time period without increasing over time. Judging from the result, the length of the input change history does not seem to seriously affect the maximum heap consumption. This tendency was also observed in the all other projects.

(3) Disk space (the column of File size)

Both OpG2 and OHG instances are output as XML files. To show that graph instances are actually treated in a realistic document size, the sum of output file size was calculated.

Disk space for OHG instances is approximately 12% of OpG2’s in total (for both WS and NB). Note that OHG Builder determines the output filepath randomly. Thus, the total file size varies for each time. The result shows the file size of a graph instance is not an issue for prevailing PCs.

(4) Graph generation time (the column of Time)

OHG instances can be separately read, as different from OpG2. This design decision of OHG negatively affects the graph generation time since it incurs frequent hard disk access. Hence, we compared the required time to generate graph instances.

When using WS, the graph generation time for OHG is 398.6% of OpG2’s on average. Like this, OHG is disadvantageous to OpG2 with respect to the generation time. However, OHG Builder can still process 53 normal operations (5.2 minutes of development time) in a second on average. This performance is considered to be enough practical since it can be applied for real-time graph generation (in future). Moreover, when using NB, the performance is 1.5 times higher than using WS due to its disk access speed.

Answer to RQ1

Yes, OHG Builder can reduce the required computer resource to build graph instances as compared to OpG2. A PC with standard specifications actually generated graph instances in a realistic time, although a slightly longer time than that for OpG2 was needed.

5.2 Evaluation for Accuracy of Traceability Links

Regarding program change understanding, we evaluated whether OHG contributes to the accuracy of traceability links.

 Ideally, we should apply OHG to program change understanding tasks in real development to clarify its contribution. However, our experimental dataset is quite old (recorded between 2012 and 2016), and the original developers already forgot most of development situations. Therefore, we focused on the accuracy of traceability links on program elements as a significant part of program change understanding.

We used GT [17] as a comparison target of this experiment. The reasons we selected GT as the baseline are:

- It is widely used in many studies and we can obtain executable files from its website.
- Our preliminary study showed it can generate more accurate traceability links than OpG2.

GT detects a code change as an insertion, deletion, update, or move of an AST node by comparing two successive code snapshots. Although the snapshots are supposed to be obtained from a VCS in its normal use, we used snapshots which are restored by using the edit operation dataset in our experiment. This is because a former study [24] already showed that using coarse-grained change data (i.e., revisions) incurs serious loss of the traceability. To our knowledge, no study has applied GT to operation-based code snapshots for a traceability analysis.

5.2.1 Experiment Procedure

Here we explain the experiment procedure. First, we created code snapshot data with our dataset mentioned in 5.1.1. The code snapshot data contain all states (before and after every edit operation) without syntax errors of each source file. Next, we applied GT to the data, so that successive snapshots of the same source file were compared. As the result, we obtained code change information.

(1) Traceability evaluation 1

From the GT’s output, 50 cases of updating or moving a program element were randomly selected†. For each case, we manually checked whether it was really an update (rename or signature change) or move by replaying corresponding edit operations. Each result was classified as true (TP: true positive) or not (FP: false positive). Here, a TP means the developer actually conducted a deletion and insertion of text strings (including automated edits) for changing the element’s unique name. A FP is opposite.

Then, we checked whether OHG Builder generated a correct traceability link in each case. If OHG Builder generated a correct link, the case is determined as a TP. If it does not generate a link, the case basically falls into a FN (false negative). However, when a developer clearly did not intend

†The original outputs of GT contain changes in non-target elements (e.g., literal and local variable). Therefore, non-target cases were excluded from selected cases when we checked each case by replaying. The number of GT’s outputs which are targets of this study is estimated to be 378.
Table 3 Result overview in the traceability evaluation 1.

|       | GT       | OHG     |
|-------|----------|---------|
| TP    | 44       | 36      |
| TP    | 35       | 35      |
| TN    | 6        | 10      |
| FN    | 6        | 5       |

(2) Traceability evaluation 2
Since the traceability evaluation 1 cannot detect FNs of GT, we then randomly selected 50 cases that OHG Builder detected as a rename or move of a program element. Similar to the traceability evaluation 1, we manually checked whether it is a real rename or move by replaying the recorded operations. If the result is associated with a real rename, it is categorized as TP.

In this experiment, we found fake traceability links derived from modifications in import statements. When a developer first inserts a simple type name in a declaration of a class member, OHG Builder creates its corresponding node with the simple type name. If the developer later adds an import statement for it, OHG Builder detects it as a rename and generates a new node with a traceability link since the unique name is changed. This link may mislead the tool’s users. To solve this problem, OHG Reader parses a unique name to identify each qualified type name that the unique name contains. Based on the result of parsing, OHG Reader can correctly judge whether a traceability link is fake or not by comparing a non-qualified type name and qualified one. Thus, fake names are not problematic in real use. Therefore, we excluded the cases of fake names from this evaluation.

Then we checked whether GT’s result for each case is correct. When GT can correctly detect the update, the case is categorized as TP.

5.2.2 Results

(1) Result of the traceability evaluation 1
Here, we explain the result of the traceability evaluation 1. The overall result is shown in Table 3. GT detected 44 TPs and 6 FPs. As for the 44 cases that GT detected as TPs, OHG detected 35 TPs, 6 TNs, and 3 FNs. As for the 6 cases that GT detected as FPs, OHG detected 5 TNs and 1 FP. The accuracy of OHG is (46/50) = 0.92, while GT is (44/50) = 0.88. The accuracy is calculated as $\frac{TP+TN}{TP+TN+FP+FN}$. TN and FN values of GT are simply deemed as 0. OHG’s accuracy is slightly better than GT’s. Here, we do not claim whether the result is statistically significant or not since a larger dataset is required to perform a statistical test with an appropriate sample size.

Details of GT’s false positives
To deepen the consideration about the validity of traceability links, FP cases in GT’s result are shown below. Since GT’s algorithm compares the AST structures, FPs occur when similar statements were added or a bracket composing a block was modified.

1. When a developer fixed the correspondence of brackets, GT detected a move of a type reference in it.

2. When a developer replaced a large code fragment containing several empty methods by a paste operation, GT detected a move of a method. Several methods are common in code snapshots before and after the paste, but they are placed in different order. The authors considered this developer’s intention is to copy the implementation of those methods rather than move.

3. A field was deleted and another field with different type and name was inserted at the same place after a while. GT detected it as an update.

4. A developer pasted the whole code of a source file into another file and changed the class name in the pasted code. GT detected it as an update. In this case, the developer clearly intended to copy the class and the tool can know that the source code just after the paste contains a syntax error (since the source file name and class name are different). Therefore, we judged this case as a FP.

5. When a developer added a new declaration of a field whose type is the same as the existing variable in the file, GT detected a move of a type reference.

6. When a developer repeatedly added field declarations with the same modifiers (“public static final”), GT detected a move of a part of a field declaration.

Details of OHG’s failures

1. A developer deleted the whole declaration statement of a field and inserted a new field declaration by the same operation. OHG Builder detected it as a deletion and insertion. This occurred in three cases. It is controversial whether the developer really intended a rename. Although we cannot conclude with certainty because of our stale dataset, we adopted a conservative criterion to prevent deriving a biased conclusion.

2. When a developer pasted the whole code of a source file into another file and changed the class name in the pasted code, OHG Builder detected it as a rename.

††This syntax error is an exceptional case. This snapshot was used as an input since it does not have a syntax error when focusing on only source file contents.
Details of OHG’s true negatives

We deemed six cases of OHG’s results as TNs although they are deemed as TPs in GT’s results. In these cases, a developer copied a declaration statement of a program element and modified its name to append a new declaration similar to the existing one. GT detects such a case as an update of the new element, while OHG Builder detects as an insertion of the new element. Overall, it is clear that the developer’s intention is to append a new declaration. However, focusing only after copying the declaration statement, the developer needed to change the element’s name. Therefore, both results are reasonable, and we concluded such a case is not a fault for both tools.

Since GT detects changes in the AST structure, it may recognize edit operations inserting or deleting brackets as a move of elements surrounded by them. It may also make wrong matching results when a code fragment with a similar or the same structure is inserted. On the other hand, OHG Builder checks whether existing characters were deleted or not. Thus, it correctly judges the traceability of a program element when its AST path was not changed and the corresponding text string partially remained. This mechanism did not incur false positives in our experiment. Adversely, OHG Builder does not generate a traceability link when the entire text of a program element was temporarily deleted. In this case, the developer might intend to change the element, not delete and insert. We need to improve our method to handle such cases in future.

(2) Result of the traceability evaluation 2

Here, we explain the result of the traceability evaluation 2. Overall, both tools had 100% accuracy (detected 50 TPs) in this experiment. Note that GT did not detect a change within the signature of a program element (e.g., adding/deleting a method parameter, method return-type change) as an update. This is because GT generates subnodes of method and field declarations. Example cases are shown below:

- A change in a parameter type (from Normal Operation to ArrayList<NormalOperation>) is detected as a combination of an addition and move.
- A change in a return type (from int[][] to int) is detected as a deletion.

Since OHG treats a program element with its unique name, such case is simply detected as a rename (unique name change).

Answer to RQ2

Yes, OHG Builder can provide slightly more accurate traceability links as compared to the baseline approach using GT. Although both approaches cannot perfectly detect correct traceability links, both of them achieved a high accuracy.

6. Discussion

6.1 Performance

As shown in 5.1.2, OHG generation is slower than OpG2. If an edit history that needs to be treated is small enough, OpG2 may be a better option because of its graph generation time. However, as shown in 5.1, OpG2 can exhaust computer resources even with a history as short as a couple of weeks. As mentioned in Sect. 1, OpG2 has a monolithic data format. This means that an OpG2 instance cannot be separately loaded even if only a small part of the graph is needed. Thus, OpG2 can incur a serious performance problem in both reading and writing graph instances.

To the contrary, any part of an OHG instance can be loaded on demand due to its decentral data format. Particularly, a user can efficiently traceability links for a long-lived program element with minimal computer resource since the links are explicitly kept in the graph instance. In addition, OHG has a much better algorithm than OpG2 for correctly detecting traceability links.

The current implementation of OHG Builder maintains a cache of recently accessed graph elements to accelerate reading them. The default cache can accommodate 10,000 graph elements. If the cache size is expanded, OHG generation can be completed in a shorter time.

We used 52 kinds of characters for a graph element’s ID in this study. This will cause a problem on a case-insensitive file system since the ID characters are also used in the file name. If we reduce the number of kinds of characters, the maximum graph size will shrink, and performance will deteriorate since the probability of an ID conflict increases. However, when we tried 26 kinds of characters (a–z) for IDs, no significant degradation of performance was observed. We consider this is because the conflict probability is low enough for our dataset even with 2610 patterns of ID using a 10-length alphabetic string. In addition, the limitation of ID pattern limits the number of graph elements (i.e., the length and size of the project). However, the ID length can be extended even in the midst of development, if needed.

In our study, a single file composing an OHG instance contains two graph elements at maximum. If the threshold is a larger value, chance of file access conflict would increase and file access would be slower. We consider the threshold should be kept small unless the number of files is a disadvantage (e.g., slowing file copy).

The reason why OHG Builder can limit its memory consumption to a low level is that the tool reserves memory space for only the minimum required ASTs in Step 9 (mentioned in 4.1) and that the memory space is released immediately after the output of the partial graph. In our ex-

1Releasing memory is conducted based on the garbage collection mechanism of Java VM. OHG Builder immediately removes references to obsolete objects so that the garbage collector can work for them.
experiment, memory consumption results in each project are the almost same level although they have different development period and code size. In our consideration, the memory consumption can be affected by data generated between Step 5 and 8 (i.e., traceability data, cut-and-paste data, and simplified file history) since they are kept in memory during the later processes. Although the difference in the size of those data is not dominant within our dataset, those data may expand when the history is much longer. If the size of those data invalidates our current method, loading only recent parts of them will ameliorate the situation.

6.2 Traceability Links

We assume that there is not a universal method to determine whether a changed element should be treated as the same one or not. Hence, a flexible traceability analysis mechanism that can be customized by its user may improve the understandability of code changes. Actually, three cases of FN in our experiment of OHG (mentioned in 5.2.2) can be resolved by slightly modifying the SAST comparison algorithm. Moreover, the accuracy of detecting renames and moves can be improved by extending operation patterns. Of course, such a change in the detection algorithm may increase the number of false positives. However, we can still explore the better algorithm by flexible customization. Such customization is due to using operation history that contains developers’ various activity. In other words, our method has potential to more accurately detect traceability links than the current approach.

In the experimental results where OHG Builder detected as a rename, we found several cases considered redundant for change understanding. For example:

```java
-> View#textField:JTextField
-> View#textField:Label
```

In this case, the developer changed the type of the field and then he renamed it, while the role of the UI component was not changed. OHG Builder detected the change as two renames of the field since it reflects the developer’s edit procedure. To remove the intermediate state of the field for better understandability, it might be effective that an element with a short lifetime is omitted from the result.

In another case, when a developer manually renamed a class, OHG Builder separately detected the change in the class’s unique name and all other references to the class. In this case, changes in a type reference should be omitted from the result that OHG Reader shows.

Although the above cases were observed, the traceability links obtained by our proposed method have low risks of disturbing users’ change understanding. For example, a developer’s trials-and-errors in their editing process seldom appear on graph instances since they are generally performed in methods. Even if he/she repeats renaming a program element, those edit operations are combined into a single operation due to history simplification (Step 8(2) in 4.1).

6.3 Problems in Lightweight Analysis

OHG Builder sometimes misunderstands a part of a program since it does not perform a full semantic analysis. For example, in the following incomplete code, `canvas.addPaint` was deemed as a type name although `canvas` and `textField` are actually a variable and `addPaint` is a part of a method name.

```java
canvas.addPaint
textField=new JTextField();
```

In addition, OHG Builder cannot correctly resolve type names with on-demand import. However, it does not always adversely affect change understanding. In our dataset, a developer used an on-demand import "org.eclipse.swt.widgets.*". However, we can estimate the unique name of each class from a simple name since the package is notably famous.

6.4 Dependency on the Recording Tool

Our method is based on OperationRecorder, which we previously created. OperationRecorder records code changes in a sequence of textual edit operations. It allows us to precisely restore code snapshots that enhance code change understanding. Note that a part of our proposed method does not depend on OperationRecorder (i.e., graph representation and lightweight parsing of code snapshots).

6.5 Threats to Validity

We consider there is no universal method to determine what traceability links are correct. The evaluation results with regard to the traceability links (e.g., which traceability link is a fault) may depend on the experimenter. Therefore, we adopted a conservative criterion to prevent overestimating our proposed method. Note that other experimenters may find different fault examples of OHG and GT.

In our study, we built graph instances from change histories of completed software development projects. As a real use case of OHG, a user may want to apply our method to a change history of an ongoing project. In such a case, the evaluation must be conducted under a situation where the user’s development environment occupies a certain amount of computer resources. We consider that graph generation can be conducted without issues with computer resources even concurrently with ordinary development activities (i.e., a graph is generated in the background) since the experimental results mentioned in 5.1.2 showed OHG Builder occupies little memory in graph generation.

The dataset used in this study is composed of in-house software development projects. Each project was conducted by a single developer. If we use more and various data, the experimental results may differ.

In addition, the evaluation for accuracy of traceability links in 5.2 is based on random selection. Hence, the result
changes any time the experiment is repeated.

This study is focused on the traceability on program elements as a crucial part of program change understanding. Of course, this premise has room for discussion. The study of Hora et al. [24] implies the significance of keeping track of program elements for precisely tracing API evolution. Moreover, many change detection methods mentioned in Sect. 2 can correctly detect deletions and insertions of program elements, while the ability and accuracy of detecting renames and moves vary among them. Therefore, we consider providing correct traceability links corresponding to renames and moves of program elements is particularly important in program change understanding.

7. Conclusion

In this paper, we proposed an operation history graph called OHG, a method to build it, and implemented tools. The evaluation showed that the performance of OHG instance creation and the accuracy of traceability links.

In this work, we omitted the methods’ inner-structure in ASTs since a lot of code snapshots must be treated. Due to our omission policy, the experimental result showed the graph size drastically decreased. The effect was better than we expected. On the other hand, the omission limits the variety of applications of OHG. To solve this, we will try to keep all the nodes in ASTs in the graph in future. Future works also include evaluation for program comprehension with various datasets in wild, devising a technique to extend an existing graph instance each time a new edit operation arrives, improvement of usability of OHG Reader, and its evaluation.

Acknowledgments

This work was partially sponsored by the Grant-in-Aid for Scientific Research (20K11762) from the JSPS.

References

[1] T. Omori, S. Hayashi, and K. Maruyama, “A Survey on Methods of Recording Fine-Grained Operations on Integrated Development Environments and Their Applications,” Computer Software, vol.32, no.1, pp.60–80, 2015. Translated version available at http://www.ritsumei.ac.jp/~tomori/publication/omori-survey16.pdf.
[2] Q.D. Soetens, R. Robbes, and S. Demeyer, “Changes as First-Class Citizens: A Research Perspective on Modern Software Tooling,” ACM Computing Surveys, vol.50, no.2, pp.18:1–18:38, 2017.
[3] T. Omori and K. Maruyama, “A Change-aware Development Environment by Recording Editing Operations of Source Code,” Proc. MSR ’08, pp.31–34, 2008.
[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?,” Proc. ECOOP ’12, vol.7313, pp.79–103, 2012.
[5] T. Omori, K. Maruyama, and A. Ohnishi, “Implementation of the Operation History Graph for Understanding Fine-Grained Software Evolution,” JSSST Symposium on Foundation of Software Engineering, pp.113–118, 2018 (in Japanese).
[6] T. Omori, K. Maruyama, and A. Ohnishi, “Summarizing Code Changes by Tracing an Operation History Graph,” Proc. MAINT ’19, pp.14–18, 2019.
[7] T. Omori and K. Maruyama, “An Editing-Operation er with Highlights Supporting Investigation of Program Modifications,” Proc. IWPESE-EVOL ’11, pp.101–105, 2011.
[8] R. Robbes and M. Lanza, “A Change-based Approach to Software Evolution,” Electronic Notes in Theoretical Computer Science, vol.166, pp.93–109, 2007.
[9] T. Omori and K. Maruyama, “Comparative Study between Two Approaches Using Edit Operations and Code Differences to Detect Past Refactoring,” IEICE Trans. Information and Systems, vol.E101-D, no.3, pp.644–658, 2018.
[10] L. Hattori and M. Lanza, “Syde: A Tool for Collaborative Software Development,” Proc. ICSE ’10, pp.235–238, 2010.
[11] Y. Yoon and B.A. Myers, “Capturing and Analyzing Low-Level Events from the Code Editor,” Proc. PLATEAU ’11, pp.25–30, 2011.
[12] K. Maruyama, S. Hayashi, and T. Omori, “ChangeMacroRecorder: Recording Fine-Grained Textual Changes of Source Code,” Proc. SANER ’18, pp.537–541, 2018.
[13] K. Maruyama, T. Omori, and S. Hayashi, “Slicing Fine-Grained Code Change History,” IEICE Transactions on Information and Systems, vol.E99-D, no.3, pp.671–687, 2016.
[14] M. Kim and D. Notkin, “Program Element Matching for Multi-Version Program Analyses,” Proc. MSR ’06, pp.58–64, 2006.
[15] H. Kagdi, M.L. Collard, and J.J. Maletic, “A survey and taxonomy of approaches for mining software repositories in the context of software evolution,” Journal of Software Maintenance and Evolution: Research and Practice, vol.19, no.2, pp.77–131, 2007.
[16] 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.
[17] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, “Fine-grained and Accurate Source Code Differencing,” Proc. ASE ’14, pp.313–322, 2014.
[18] G. Dotzlzer and M. Philippesen, “Move-Optimized Source Code Tree Differencing,” Proc. ASE ’16, pp.660–671, 2016.
[19] Y. Higo, A. Ohtani, and S. Kusumoto, “Generating Simpler AST Edit Scripts by Considering Copy-and-Paste,” Proc. ASE ’17, pp.532–542, 2017.
[20] V. Frick, T. Grassauer, F. Beck, and M. Pinzger, “Generating Accurate and Compact Edit Scripts using Tree Differencing,” Proc. ICSME ’18, pp.264–274, 2018.
[21] K. Huang, B. Chen, X. Peng, D. Zhou, Y. Wang, Y. Liu, and W. Zhao, “CIDiff: Generating Concise Linked Code Differences,” Proc. ASE ’18, pp.679–690, 2018.
[22] Y. Higo, S. Hayashi, and S. Kusumoto, “On Tracking Java Methods with Git Mechanisms,” Journal of Systems and Software, vol.165, article 110571, 2020.
[23] G. Torre, R. Robbes, and A. Bergel, “Imprecisions Diagnostic in Source Code Deltas,” Proc. MSR ’18, pp.492–502, 2018.
[24] A. Hora, D. Silva, M.T. Valente, and R. Robbes, “Assessing the Threat of Untracked Changes in Software evolution,” Proc. ICSE ’18, pp.1102–1113, 2018.
[25] M. Pawlik and N. Augsten, “RTED: A Robust Algorithm for the Tree Edit Distance,” Proc. VLDB Endowment, vol.5, no.4, pp.334–345, 2011.
[26] C.V. Alexandru, S. Panichella, S. Proksch, and H.C. Gall, “Redundancy-free analysis of multi-revision software artifacts,” Empir Software Eng, vol.24, pp.332–380, 2019.
[27] VisualVM, https://visualvm.github.io/ (Feb. 25, 2019 access).
Takayuki Omori is an associate professor at Department 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.

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 the Department of Computer Science, 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 Computer Society, ACM, IPSJ, and JSSST.

Atsushi Ohnishi received B. of Engineering, M. of Engineering, and Dr. of Engineering degrees from Kyoto University in 1979, 1981, and 1988, respectively. He was a Research Associate and an Associate Professor of Kyoto University from 1983 to 1994. Since 1994 he has been a Professor at Department of Computer Science (currently, Department of Information Science and Engineering), Ritsumeikan University. He was a visiting scientist at UC Santa Barbara from 1990 to 1991 and a visiting scientist at Georgia Institute of Technology in 2000. His current research interests include requirements engineering, object oriented analysis, and software design techniques. Dr. Ohnishi is a member of IEEE Computer Society, ACM, Information Processing Society (IPS) of Japan, the Institute of Electronics, Information and Communication Engineers (IEICE), and Japan Society for Software Science and Technology (JSSST).