Semantic Zooming of Code Change History

YoungSeok Yoon
Institute for Software Research
Carnegie Mellon University
Pittsburgh, PA 15213, USA
youngseok@cs.cmu.edu

Brad A. Myers
Human-Computer Interaction Institute
Carnegie Mellon University
Pittsburgh, PA 15213, USA
bam@cs.cmu.edu

***Abstract*—Previously, we presented our technique for visualizing fine-grained code changes in a timeline view, designed to facilitate reviewing and interacting with the code change history. During user evaluations, it became evident that users often wanted to see the code changes at a higher level of abstraction. Therefore, we developed a novel approach to automatically summarize fine-grained code changes into more conceptual, higher-level changes in real time. Our system provides four collapse levels, which are integrated with the timeline via semantic zooming: *raw level* (no collapsing), *statement level*, *method level*, and *type level*. Compared to the raw level, the number of code changes shown in the timeline at each level is reduced by 55%, 77%, and 83%, respectively. This implies that the semantic zooming would help users better understand and interact with the history by minimizing the potential information overload.**

Keywords—semantic zooming; edit collapsing; program comprehension; software visualization; timeline visualization; Azurite

I. Introduction

In our previous work, we presented a *timeline visualization* of the fine-grained code change history, where all the changes are represented as color-coded rectangles in a two-dimensional space (see Fig. 1a) [1]. Not only can users see the change history from the timeline, but they can also select one or more of the changes and invoke useful editor commands such as *selective undo*, which undoes only the selected changes without affecting the following changes in the history. After implementing these features in an Eclipse plug-in called AZURITE (**A**dding **Z**est to **U**ndoing and **R**estoring **I**mproves **T**extual **E**xploration), we conducted a controlled lab study which showed that the timeline is usable and programmers can perform their backtracking tasks twice as fast compared to when not using AZURITE [2].

The timeline can be zoomed in and out. However, in earlier versions of the timeline, it was difficult to see the bigger picture of the history from the timeline, even when it was significantly zoomed out, because the individual edits were still too fine-grained. During our evaluation studies, users mentioned that they often wanted to see the code changes at a higher-level of abstraction, such as the level of adding a field, editing an existing method, and so on. Therefore, we added a *semantic zooming* feature to the timeline, which is the main topic of this paper. The timeline dynamically adjusts the presented level of detail, depending on the current horizontal zoom scale. To provide this semantic zooming feature, we devised a real-time edit collapsing algorithm, which takes fine-grained code changes as input and produces more conceptual and abstract level of code changes.

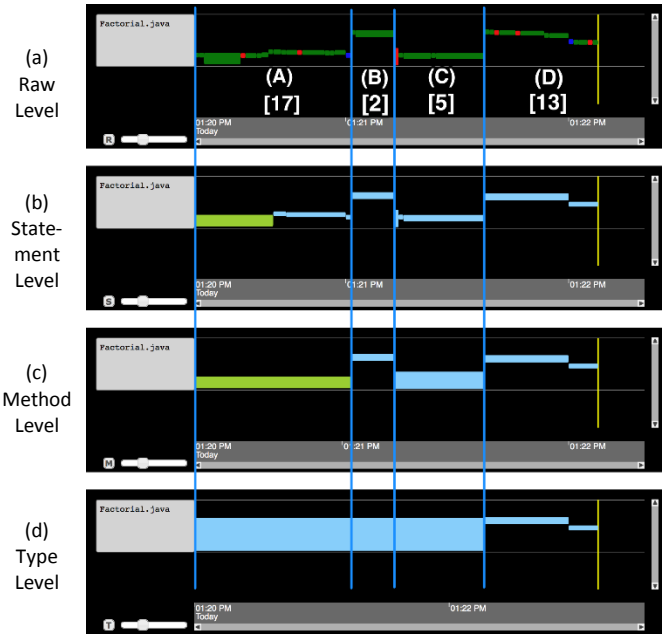

Fig. 1. The timeline view after completing the example coding task described in Fig. 2, shown at different levels of detail but the same zoom level. The blue vertical lines were added for the purpose of explanation, and are not shown in the actual timeline. The numbers in the square brackets indicate how many rectangles are in each section at the raw level.

There are existing tools that summarize the code changes when two snapshots of code – often from the version control system – are given (e.g., [3, 4]). However, these techniques use a top-down approach: they extract conceptual edits from the complete snapshots. In contrast, AZURITE uses a bottom-up approach: it collapses and summarizes multiple, fine-grained edit operations into a more meaningful, conceptual edit in real time, as the user edits the code. Our algorithm is capable of handling the *stream of edits* in an efficient, incremental manner.

II. Example

Imagine that a programmer wants to write a factorial number calculating program. For example, she might implement the factorial calculator by taking the following steps in order:

(a) Write `factorial` method using a `for` loop (Fig. 2a)
(b) Test the function with a constant value (Fig. 2b)
(c) Change the `factorial` method to use recursion (Fig. 2c)
(d) Modify the `main` method to get user input (Fig. 2d)

978-1-4673-7457-6/15/$31.00 ©2015 IEEE

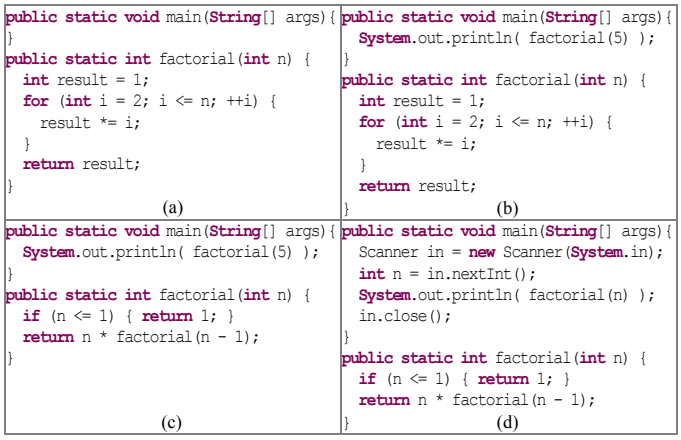

Fig. 2. The code changes for the factorial example.

Fig. 1a shows the state of the timeline after completing all of these steps. The timeline is partitioned into four sections, each corresponding to a programming step described above. The blue vertical lines are not actually shown in the timeline but were added on the screenshot for the purpose of explanation.

It can be seen that there are relatively many rectangles shown in the timeline, even for the seemingly simple programming steps. With the real-time edit collapsing algorithm, the same code edit history can be displayed more abstractly. Fig. 1 shows how the example script would be displayed at different collapse levels but at the same zoom level. AZURITE's timeline supports a total of four collapse levels: *raw level*, *statement level*, *method level*, and *type level*, listed from the lowest to the most abstract.

III. THE FOUR COLLAPSE LEVELS

A. Raw Level (No Collapsing)

The raw level (Fig. 1a) shows the fine-grained edits as they arrive, without any collapsing. This level was used during the evaluation studies of AZURITE [2].

B. Statement Level

The statement level (Fig. 1b) is now used as the default in the timeline. The goal of constructing the statement level is to collapse consecutive edit operations that belong to the same statement. This is achieved by an empirically developed approach. Whenever an edit introduces syntax errors to the code, the collapse logic waits until those errors are removed by the incoming edits and collapses them together, which would typically happen when a semicolon is typed at the end of a statement. Using this rule, the individual collapsed edits become much more comprehensible, and they typically represent a single statement change, variable addition, empty method stub addition, etc.

C. Method Level

The main idea of the method level (Fig. 1c) is to collapse all the consecutive edits made in the same method into a single edit. The method level is useful for discriminating the conceptual units of code edits, assuming that programmers divide the code logic into relatively small methods. For example, in Fig. 1c, each step of the example factorial programming matches a single rectangle in the timeline, because the programmer was alternating between the **`factorial`** method and the **`main`** method. At the method level, she could easily backtrack by selecting the changes that modified the **`factorial`** method to use recursion with a single mouse click, and then invoking selective undo.

D. Type Level

The highest collapse level is the type level (Fig. 1d). Similar to the method level, the main idea is to collapse all the consecutive edits in the same type into a single edit. The reason for providing type level is to make it easier to review or interact with the code edit history when the programmer is working with nested types, or multiple types simultaneously. A great example of such situations is when the programmer is using the State or Strategy design patterns [5], which are often implemented as nested classes in Java.

IV. COLLAPSING ALGORITHM

A. Overall Collapse Mechanism

The collapsing algorithm works in each of the collapse levels separately. The key idea of this collapsing algorithm is to keep a list of pending edits (*pending list*, hereafter) for each level. The edit operations in the pending list have already been determined to be collapsed together at that level, but are still pending in that the next incoming edit(s) may also be collapsed with them.

Once a new edit operation is added to the history buffer, the edit operation is first considered by the statement level collapser. There can be three different outcomes. (1) If the current pending list is empty, then the incoming edit is added to the pending list. When there are existing pending changes, the statement level collapser runs the collapse test with the rules described in Section III.B. (2) If the edit should be collapsed, then it is added to the end of the pending list. (3) If the edit should *not* be collapsed, then all the currently pending edits are finally marked as collapsed, the pending list is emptied, and the new incoming edit is added to the now empty pending list. When this happens, the edits that were just collapsed are considered by the next level (the method level, in this case) collapser as the new incoming edits. The same process is followed by the method level collapser, with its own collapse test, and the edits collapsed at the method level are then considered by the type level.

The collapsing algorithm had to remain compatible with the selective undo presented in [2], which required the following rules. First, collapsing is not allowed to reorder edits. Second, edits that are collapsed at one level cannot be split at a higher level. Finally, edits are never collapsed across user-defined tags or run/save events, since prior work shows that these events serve as explicit or implicit checkpoints for programmers [2].

B. Collapse Test for the Method Level and Type Level

At the method (or type) level, the basic collapse rule is to collapse the consecutive edits made in the same method or field (or type). In order to run this collapse test successfully, the collapser must first extract some *change details* of the edits. Most importantly, the *change kind* is determined, among the list of 12 kinds summarized in Table I. The change details also denote whether the edit range is bound to a certain method or class.

The change detail extraction process is illustrated in Fig. 3. To extract change details of a set of code edits, the system takes the two snapshots, before and after the edits, and the interim edit

TABLE I. KINDS OF CODE EDITS

Kind of Edit	Abbr.	Description
Add Field	AF	Adding a new field to a class
Change Field	CF	Modifying an existing field
Delete Field	DF	Deleting an existing field
Add Method	AM	Adding a new method
Change Method	CM	Changing an existing method
Delete Method	DM	Deleting an existing method
Add Type	AT	Adding a new type declaration
Change Type	CT	Changing an existing type declaration
Delete Type	DT	Deleting an existing type declaration
Change Import Statement	CIS	Adding, changing, or deleting one or more import statements
Non-code Change	NCC	Editing without altering the AST structure
Unknown	UNK	All the other changes

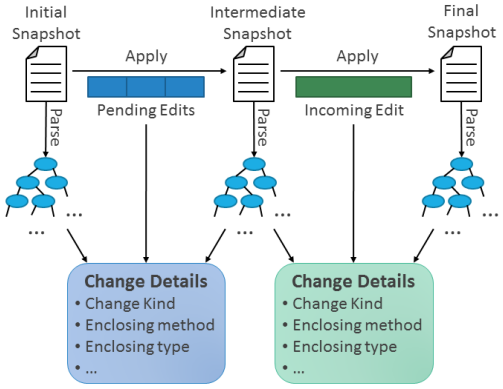

Fig. 3. Illustration of the change detail extraction process.

operations as input. Because there are two sets of edits in consideration, pending edits and the incoming edit, the collapser needs to analyze three versions of code snapshots. Each of these snapshots are first parsed into an abstract syntax tree. Then, the pending change details and the incoming change details are extracted from the three snapshots and the edits themselves.

Once the change kinds are determined, the collapser uses the collapse test matrix to see if their change kinds are compatible and can be combined together (Table II & III). The content of each cell indicates the resulting change kind after collapsing the pending changes and the incoming change. For example, at the method level (Table II), an AM change followed by a CM change on the same method results in a collapsed AM change. The same logic applies to the fields. At the type level, more kinds of changes can be collapsed than at the method level. For example, an AM change followed by another AM change can be collapsed at the type level, provided that they were added to the same type. The gray cells indicate that they are not collapsible.

V. INTEGRATION WITH THE TIMELINE VISUALIZATION

The collapsed edits can be displayed in the timeline, as shown in Fig. 1. The group rectangles are color-coded according to their change kind. All the Adds are colored as yellow-green, all the Changes as sky-blue, and all the Deletes as pink, which corresponds to the colors of the member edits at the raw level. The other kinds of changes (NCC, UNK) are shown as grey. One or more group rectangles can be selected by mouse as they could at the raw level, and the context menu items work exactly the same way as they would when all the member rectangles are selected at the raw level.

TABLE II. COLLAPSE TEST MATRIX – METHOD LEVEL

Pending Changes \ Incoming Change	AF	CF	DF	AM	CM	DM	AT	CT	DT	CIS	NCC	UNK
AF		AF	NCC									
CF		CF	DF									
DF												
AM					AM	NCC						
CM					CM	DM						
DM												
AT												
CT												
DT												
CIS										CIS		
NCC											NCC	
UNK												

TABLE III. COLLAPSE TEST MATRIX – TYPE LEVEL

Pending Changes \ Incoming Change	AF	CF	DF	AM	CM	DM	AT	CT	DT	CIS	NCC	UNK
AF	CT	CT*	CT*	CT	CT	CT		CT	DT			
CF	CT	CT*	CT*	CT	CT	CT		CT	DT			
DF	CT	CT	CT	CT	CT	CT		CT	DT			
AM	CT	CT	CT	CT	CT*	CT*		CT	DT			
CM	CT	CT	CT	CT	CT*	CT*		CT	DT			
DM	CT	CT	CT	CT	CT	CT		CT	DT			
AT	AT	AT	AT	AT	AT	AT		AT	NCC			
CT	CT	CT	CT	CT	CT	CT		CT	DT			
DT												
CIS										CIS		
NCC											NCC	
UNK												

* or the value specified in the method level matrix, if the changes are made on the same code element

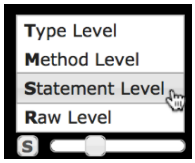

Fig. 4. The horizontal zoom slider, the collapse level controller button (S), and the popup menu.

A. Semantic Zooming

The timeline supports semantic zooming using the edit collapsing mechanism (Fig. 4). By default, the statement level is used, and the level is automatically adjusted as the zoom scale changes. Users can also manually change the level by clicking the collapse level controller (the "S" in Fig. 4) and then selecting the desired level, without changing the zoom scale. The first letter of the current level is always displayed as the button label.

B. Width and Height of a Group Rectangle

The width of a group rectangle is calculated by taking the sum of all the widths of the member edits that they would have at the raw level. Thus, the width of a group rectangle indicates the actual time the user spent to make the group edit, excluding all the idle times in-between. The height of a group rectangle indicates the size of all the member edits combined relative to the entire file in terms of number of characters, and the "Y" position represents the relative location of the edit in the file.

C. Summarizing the Edits

In the tooltip of a group rectangle, a one-line summary of the edit is displayed at the top, using the change details obtained during the collapse test. The summary is followed by the actual code changes. In Fig. 5, the summary is the part saying "Changed method '**factorial**'". As shown in this example, the name of the relevant code element (e.g., '**factorial**') is also displayed.

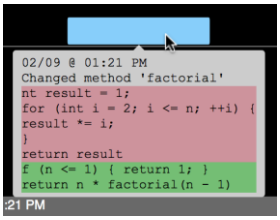

Fig. 5. A tooltip for a group rectangle showing the human-readable summary of the change.

TABLE IV. NUMBER OF EDIT OPERATIONS AT EACH LEVEL

	Raw	Statement	Method	Type
Total #	282,195	127,683	63,984	47,384
Avg. #/hr	193/hr	88/hr	44/hr	32/hr
% RPL[a]		**55%**	**50%**	**26%**
% RRL[b]		**55%**	**77%**	**83%**

[a.] % Reduction from the Previous Level
[b.] % Reduction from the Raw Level

TABLE V. RUNNING TIME OF THE COLLAPSE LOGIC

Collapse Level	Running Time
Statement Level	7.08 ms
Method Level	11.29 ms
Type Level	11.80 ms

VI. EVALUATION

A. Log Analysis

To evaluate the performance of the collapsing mechanism, it was tested with the entire code editing transcripts obtained from the longitudinal study of programmers' backtracking [6], which contains 1,460 hours of detailed coding events including all the editor commands and fine-grained code changes collected from 21 programmers. The edit collapsing component processed the edits in the transcripts as if they were made in the code editor.

Table IV shows the number of edits in each collapse level. There were a total of 282,195 edits at the raw level. On average, the statement level collapser reduces the number from the raw level by 55%. In turn, the number is reduced by 50% more at the method level, and reduced by 26% more at the type level. This implies that the collapsing mechanism would be useful in minimizing the potential information overload by dramatically reducing the number of rectangles displayed in the timeline.

B. Performance Analysis

We also measured the performance impact of the collapse logic, using the same data set as in Table IV, using a PC running Windows 8 with a 2.60 GHz CPU. Table V summarizes the mean time it takes to run the collapse logic at each collapse level. The statement level logic, which mainly tests if the current code is parseable, takes about 7ms on average. The method and type level logic, which requires more sophisticated change detail extraction process as in Fig. 3, takes about 11~12ms.

From the data presented in Table IV, we can obtain the invocation rates of the collapse logic at each level. The statement level collapse logic is called every time when a new edit is made. The method level logic is called 0.45 times, and the type level logic is called 0.23 times per edit on average. Combining the invocation rates with the measured time, the average time it takes to run the collapse level per edit operation can be calculated as:

$$7.08\text{ms} \times 1 + 11.29\text{ms} \times 0.45 + 11.80\text{ms} \times 0.23 = 14.87\text{ms/op}$$

This means that whenever a new edit is made, the collapsing logic runs for 15ms on average, which we consider acceptable.

VII. LIMITATIONS AND FUTURE WORK

The edit collapsing mechanism described in this paper has a number of limitations. First, the mechanism never reorders the edits to make it compatible with our selective undo mechanism [2], so the collapsing mechanism may not work well when the programmer is jumping around multiple locations in code. In the future, a history refactoring mechanism as in Historef [7, 8] could be implemented to support edit collapsing more flexibly.

While the idea behind the collapsing mechanism is language independent, our implementation is tied to Java. Also, the collapsing mechanism only detects a few kinds of changes, and we could add more kinds as in [9] in the future. In addition, the visualization of the collapsed edits might be improved by visualizing more information about the changes in the timeline.

The source code parsing part of the collapse test logic could be improved by parsing only the changed area of code instead of the entire file, for example by using island grammars [10]. Alternatively, the AST parser could be configured to resolve the binding information to determine the connection between different parts of code, which could be useful for edit collapsing.

There could be other types of collapse levels which are orthogonal to the collapse levels that we presented. For instance, when the user is using a task tracking system such as Mylyn [11], all the edits made for the same task could be collapsed together.

VIII. RELATED WORK

The semantic zooming was inspired by previous work [12, 13], where multiple views are defined for different zoom levels, and the appropriate view is automatically chosen based on the current zoom level. The kinds of code changes in Table I are similar to the categories of atomic code changes presented in [3], with several differences. First, in their model, adding and deleting changes are always for an empty element. In contrast, in our model, an Add Method might represent an added method with its own body as well. Another difference is that our classification can represent Non-Code or Unknown changes. By representing these additional cases, they can also be selected and undone by the user. For example, an NCC code reformatting edit could later be selectively undone.

Others have developed ways to extract high-level code changes from two versions of code, using AST tree differencing [14, 15]. One of the biggest challenges of AST differencing is the problem of matching code elements between the two versions, for example when renaming happens [16, 17]. We solved this problem by taking the edit operations as input as well.

IX. CONCLUSION

We described our approach of semantic zooming of code change history, which internally uses a real-time edit collapsing algorithm, implemented in our tool AZURITE. We hope that users would be able to review and/or interact with the code change history more effectively by working at the right level of abstraction. AZURITE is a publicly available plug-in for Eclipse (http://www.cs.cmu.edu/~azurite/), and we invite your feedback.

ACKNOWLEDGMENTS

Funding for this research coms in part from the Korea Foundation for Advanced Studies (KFAS) and in part from NSF grants IIS-1116724 and IIS-1314356. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of KFAS or the National Science Foundation.

REFERENCES

[1] Y. Yoon, B. A. Myers, and S. Koo, "Visualization of Fine-Grained Code Change History," Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'13), 2013, pp. 119-126.

[2] Y. Yoon and B. A. Myers, "Supporting Selective Undo in a Code Editor," Proc. International Conference on Software Engineering (ICSE'15), 2015.

[3] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, "Chianti: A Tool for Change Impact Analysis of Java Programs," Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), 2004, pp. 432-448.

[4] M. Kim and D. Notkin, "Discovering and Representing Systematic Code Changes," Proc. International Conference on Software Engineering (ICSE'09), 2009, pp. 309-319.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Pearson Education, 1994.

[6] Y. Yoon and B. A. Myers, "A Longitudinal Study of Programmers' Backtracking," Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'14), 2014, pp. 101-108.

[7] S. Hayashi, T. Omori, T. Zenmyo, K. Maruyama, and M. Saeki, "Refactoring Edit History of Source Code," Proc. IEEE International Conference on Software Maintenance (ICSM'12), 2012, pp. 617-620.

[8] S. Hayashi, D. Hoshino, J. Matsuda, M. Saeki, T. Omori, and K. Maruyama, "Historef: A Tool for Edit History Refactoring," Proc. IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'15), 2015.

[9] B. Fluri and H. C. Gall, "Classifying Change Types for Qualifying Change Couplings," Proc. IEEE International Conference on Program Comprehension (ICPC'06), 2006, pp. 35-45.

[10] L. Moonen, "Generating Robust Parsers Using Island Grammars," Proc. Working Conference on Reverse Engineering (WCRE'01), 2001, pp. 13-22.

[11] M. Kersten and G. C. Murphy, "Using Task Context to Improve Programmer Productivity," Proc. ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'06), 2006, pp. 1-11.

[12] K. Perlin and D. Fox, "Pad: An Alternative Approach to the Computer Interface," Proc. Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'93), 1993, pp. 57-64.

[13] J. I. Hong and J. A. Landay, "SATIN: A Toolkit for Informal Ink-Based Applications," Proc. Annual ACM Symposium on User Interface Software and Technology (UIST'00), 2000, pp. 63-72.

[14] B. Fluri, M. Wursch, M. Pinzger, and H. C. Gall, "Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction," IEEE Transactions on Software Engineering, vol. 33, 2007, pp. 725-743.

[15] I. Neamtiu, J. S. Foster, and M. Hicks, "Understanding Source Code Evolution Using Abstract Syntax Tree Matching," Proc. International Workshop on Mining Software Repositories (MSR'05), 2005, pp. 1-5.

[16] S. Kim, P. Kai, and E. J. Whitehead, "When Functions Change Their Names: Automatic Detection of Origin Relationships," Proc. Working Conference on Reverse Engineering (WCRE'05), 2005, p. 10 pp.

[17] M. Kim and D. Notkin, "Program Element Matching for Multi-Version Program Analyses," Proc. International Workshop on Mining Software Repositories (MSR'06), 2006, pp. 58-64.