Temporal Code Completion and Navigation

Yun Young Lee[1], Sam Harwell[2], Sarfraz Khurshid[2], and Darko Marinov[1]

[1] University of Illinois at Urbana-Champaign, USA
{lee467,marinov}@illinois.edu

[2] The University of Texas at Austin, USA
{samharwell,khurshid}@utexas.edu

***Abstract*—Modern IDEs make many software engineering tasks easier by automating functionality such as code completion and navigation. However, this functionality operates on *one version* of the code at a time. We envision a new approach that makes code completion and navigation aware of code evolution and enables them to operate on *multiple versions* at a time, without having to manually switch across these versions. We illustrate our approach on several example scenarios. We also describe a prototype Eclipse plugin that embodies our approach for code completion and navigation for Java code. We believe our approach opens a new line of research that adds a novel, *temporal dimension* for treating code in IDEs in the context of tasks that previously required manual switching across different code versions.**

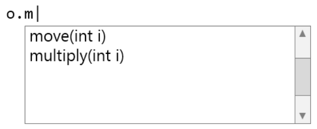
I. Introduction

Modern IDEs make many software development tasks easier by automating functionality such as code completion and navigation. For completion, many IDEs have context-specific assists for completing identifiers, such as type, method, or field names, from a given prefix of the element names. More recent advancements in code completion, such as Code Recommenders [1] in Eclipse, can even suggest identifier completion from a given partial name match, rank suggestions based on usage patterns, and suggest completion for longer code snippets. For navigation, IDEs offer functionality such as finding a declaration of a given element (e.g., navigating to a method declaration from a call site) or searching all references to a given element (e.g., showing all calls to a method). Some research prototypes additionally use dynamic program information to improve navigation [2], [3].

While much progress has been made on code completion and navigation, existing approaches operate on *one version* of the code at a time. However, as projects evolve, developers are constantly required to understand and seamlessly work with code continually changing across multiple versions. IDEs which restrict code completion and navigation to operate only on the current version inherently restrict the ability of developers to remain productive as code evolves, because developers looking for information from earlier versions are required to search through version history in a version control system (VCS) or to manually switch to a different version and search through its code.

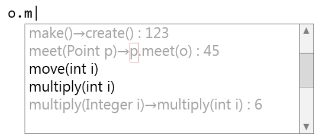
For example, consider a developer, Alice, who types `o.m` and invokes code completion, expecting to find a method `make()` that she used a few days ago. If she does not find the expected method name in the suggested list, she may suspect that someone on her team has renamed or moved the method. To find the new method name or location, she would need to cancel the current completion and search through the version history to find the specific change that altered the `make()` method. After reviewing the past changes to determine new name/location of the method, she would then need to come back to the current version before typing the updated reference. Not only has this search disrupted her development effort, but the process is also tedious and progressively slower as the size, duration, and number of people involved in a project increase.

We envision a new approach that provides a *temporal dimension* to the familiar code completion and navigation features of IDEs, allowing them to work with *multiple versions* at a time without resorting to searches through the code history in the VCS or manual version switching. Our approach will locate program elements from past versions that are relevant to the current scope and present them through code completion. In addition, our approach will allow developers to navigate to and within past versions of the code. This would help Alice quickly find the intended `make()` method.



```
o.m|
move(int i)
multiply(int i)
```

Fig. 1. Traditional code completion has only current matches



```
o.m|
make()→create() : 123
meet(Point p)→p.meet(o) : 45
move(int i)
multiply(int i)
multiply(Integer i)→multiply(int i) : 6
```

Fig. 2. Proposed code completion with historical matches (gray)

Figures 1 and 2 show two mockup completion proposal lists for this scenario: the traditional list has only matches from the current code version, and the proposed list includes historical matches from older versions. Each historical completion would show the old name, inferred new name/location in the current version, and the version ID. For example, the `make` that Alice was looking for was renamed to `create` in version 123; selecting this proposal, Alice could choose to replace her typed `o.m` with `o.create()`, or choose to inspect the diff between versions 123 and 122 or between 123 and current version. The method `meet` was moved (from the class of `o`) to the `Point` class; selecting this proposal, Alice could choose to replace `o.m` with `[p].m(o)`, where `[p]` indicates that she would need to provide an expression of type `Point`. If Alice is unable to understand the changes that occurred just from the completion proposal list, she could choose to navigate to the past version of the object `o`'s class and examine the changes made to the `make` method.

This paper makes the following contributions:

- **Idea:** We introduce the idea of a *temporal dimension* to code completion and navigation.
- **Examples:** We discuss several motivating examples where our approach would help developers by making

it easier to perform completion and navigation tasks that operate on multiple versions of code.

- **Prototype:** We describe a prototype Eclipse plugin that embodies our approach for temporal completion and navigation. Our initial experience using this prototype on a medium-scale project is positive but also points out important challenges to be addressed in future work.
- **Discussion:** We discuss the key challenges to be addressed to turn our vision into a tool which supports large projects, such as Eclipse itself.

II. Motivating Examples

We expand on the example discussed in the introduction to illustrate some common development tasks that motivate our work. These scenarios naturally arise in large teams working on source code where publicly-exposed members are likely to change over time, e.g., agile development teams and developers working on projects with incomplete API specifications. These scenarios also arise on very large projects, where it is simply impossible to keep track of all changes made throughout the history of the full project.

Consider Alice again, searching for the method `make()` that she used before but can no longer find. During her search, if the class definition of the object `o` is complex or desired functionality was moved to another class, she would have to search the change history of the class in the VCS. If the file containing the class definition has been modified many times, or if the commit messages are not clear, or if the class itself went through some renames/moves, Alice may be forced to look through a long series of commits until she finds the one containing the relevant change.

Alice's task could in fact be even more complicated, e.g., if the `make()` method was deleted altogether (e.g., inlined), or if the class of the object `o`'s class was in a third-party library class contained in a `*.jar` file and Alice's project upgraded its dependency library which involved API changes. Such changes will only exacerbate her quest to find the root cause and a solution.

Consider next a related scenario, involving navigation. Alice finds out that the `make()` method was indeed deleted and its functionality decomposed into many separate methods. Upon closer inspection, she discovers that `make` was incorrectly implemented. She would greatly benefit from being able to quickly find out what other methods `make()` used to invoke and/or what methods invoked `make()`. However, with the `make()` method gone, she can only see it in a plain text file (in the current IDEs such as Eclipse or NetBeans, without checking out the entire old code version) that she retrieved from the VCS. She is forced to search for the definitions of other methods by manually searching instead of being able to automatically navigate directly to them.

III. Approach

The main goal of our proposed temporal code completion and navigation is to make development tasks easier by providing the developers with evolution information of code in a highly interactive way.

A. Code Completion

As described previously, code completion in modern IDEs only considers the current version of the code although developers can benefit from knowing what code completions were possible in other versions. Our temporal code completion will have three automated steps:

Search Versions: When a developer invokes temporal code completion in a specific context (e.g., on a variable of a certain type), we search all the previous versions for that context (i.e., for that type definition). We can query the VCS (or its precomputed index) to retrieve all the relevant versions and invoke the code completion on each version to find a match.

Infer Changes: Finding *where* old versions have matches with the current version already helps developers, but showing *how* the matched code changed can help even more. We plan to integrate with tools that can infer changes, e.g., Git can infer some file/class renames, and research tools such as RefFinder [4] and RefactorCrawler [5] can infer refactorings that happened between two versions of code. While those techniques provide a solid foundation, our temporal code completion and navigation build on top to show *in context* the changes related to historical matches for completion.

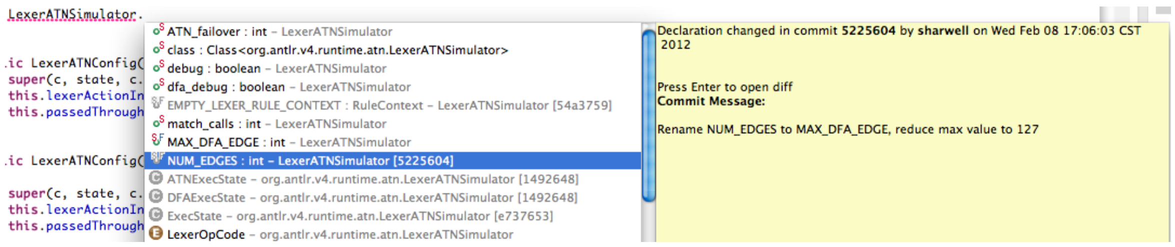
Complete New Code: When a tool finds both historical completions in old versions and how these old completions could be translated in the new version, the tool can even offer to automatically complete the old completions but in the new version. For example, the tool can offer to complete an old name that matches in an old version with the new name.

To seamlessly integrate our approach into IDEs, we will augment the existing code completion with the historical proposals. Current and historical proposals will be visually discernible so as not to confuse the developers. In addition to completing new code, we will allow the developers to *explore the past* when they select a historical proposal by showing a diff view comparing the version where a definition was changed with the previous version where the selected definition was last seen. If the developers want a deeper exploration, we will allow them to navigate to the past version of the code as described next.

B. Navigation

Seeing a diff view between two versions of code is helpful when identifying the immediate changes. However, the diff views in current IDEs primarily visualize the changes in a non-interactive manner, with at best minimal interactive features that only allow changes to be moved between versions. Our temporal navigation will treat the past versions of source code as a first-class entity like the current version of source code, offering several navigation options:

Navigate in Past: Most IDEs, such as Eclipse and NetBeans, treat past versions of code as only plain text and thus prohibit developers from easily navigating across the past versions. We argue that such restriction is unnecessary and undesirable. If the developers are made aware that a version of code is from the past, and if the IDE enforces read-only policy, the developers can take advantage of code navigation



```
LexerATNSimulator.

.ic LexerATNConfig(
super(c, state, c.
this.lexerActionIn
this.passedThrough

.ic LexerATNConfig(

super(c, state, c.
this.lexerActionIn
this.passedThrough
```

```
ATN_failover : int – LexerATNSimulator
class : Class<org.antlr.v4.runtime.atn.LexerATNSimulator>
debug : boolean – LexerATNSimulator
dfa_debug : boolean – LexerATNSimulator
EMPTY_LEXER_RULE_CONTEXT : RuleContext – LexerATNSimulator [54a3759]
match_calls : int – LexerATNSimulator
MAX_DFA_EDGE : int – LexerATNSimulator
NUM_EDGES : int – LexerATNSimulator [5225604]
ATNExecState – org.antlr.v4.runtime.atn.LexerATNSimulator [1492648]
DFAExecState – org.antlr.v4.runtime.atn.LexerATNSimulator [1492648]
ExecState – org.antlr.v4.runtime.atn.LexerATNSimulator [e737653]
LexerOpCode – org.antlr.v4.runtime.atn.LexerATNSimulator
```

Declaration changed in commit **5225604** by **sharwell** on Wed Feb 08 17:06:03 CST 2012

Press Enter to open diff
Commit Message:

Rename NUM_EDGES to MAX_DFA_EDGE, reduce max value to 127

Fig. 3. Code completion proposal list including the historically available proposals in gray. The tooltip on the right shows the commit message for the change corresponding to the selected proposal.

in the past versions to more closely learn about changes that have happened. Therefore, we aim to provide navigation functionality in the past versions of code.

Open Past Types: Just as temporal completion will allow developers to find changed or deleted methods and fields of a type, temporal navigation will allow developers to find and open old versions of types, including deleted ones. Many VCSs, such as Git, track a renamed file as deletion of a file with the old name and addition of a new file with a new name. In such cases, temporal navigation will also infer changes, just as with temporal completion.

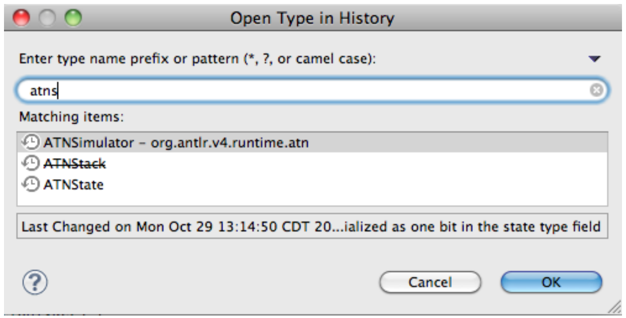
Support Past Call Graph: Most IDEs, such as Eclipse and NetBeans, provide call reference finding functionality that displays calls to and from a selected method, field, or constructor. Developers would greatly benefit from being able to view call graphs of past versions. While constructing call graphs can be expensive, an advantage for handling past versions of code is that code history does not change, so we conjecture precomputing indexes and/or caching results will allow for fast construction of precise call graphs.

IV. Prototype Tool

We have implemented a prototype Eclipse plugin that embodies our approach for temporal code completion and navigation for Java development tools with Git VCS. Our current prototype (1) augments Eclipse's code completion with proposals that were possible in all the previous versions of the identifier for which the developer seeks completion and (2) allows developers to open and navigate to declarations from the past, including deleted types.

Temporal Completion: Figure 3 shows a screenshot taken from our prototype run of temporal code completion on the Antlr project for parser generators. The historical proposals are distinguished from the current ones as they appear grayed-out. Each historical proposal includes the author and version ID in which the change was made. If the developer selects the history proposal, the prototype opens a diff view (not shown in the figure) comparing that version and the version immediately following it, to clearly show the change at the element definition.

Temporal Navigation: Our prototype allows developers to search for and open any type (including deleted ones) from any version in the past (Figure 4). A historical type is opened in a read-only editor with a list of version IDs a developer can choose to view. Developers can navigate within the editor or to other historical types in the same version.



Open Type in History

Enter type name prefix or pattern (*, ?, or camel case):

atns

Matching items:

ATNSimulator – org.antlr.v4.runtime.atn
~~ATNStack~~
ATNState

Last Changed on Mon Oct 29 13:14:50 CDT 20...ialized as one bit in the state type field

Fig. 4. Open Type in History dialog allows developers to search for types from past revisions, including deleted ones which have a strike-through. Selecting a type displays details about the last revision of the type.

Our initial experience using this prototype on the Antlr Git repository, containing more than 1,500 versions, is highly positive. In particular, our current implementation of computing method and type information (to enable temporal completion and navigation, respectively) for all the versions in this repository takes under 20 seconds and 5 seconds, respectively, on a modern laptop. In the future, we plan to improve the performance and scalability of our tool, e.g., by precomputing a database of semantic information for each version.

V. Related Work

Much work has focused on improving code completion. For example, Omar et al. [6] developed a system architecture that allows library developers to introduce interactive interfaces, called palettes, for library users to use for code completion in the context of class instantiation. Developers can also implement palettes for their own code, but palettes are highly susceptible to changes. If the code for which palettes are implemented is modified, the palettes will also need to be modified. Muşlu et al. [7] introduced an Eclipse plugin called Quick Fix Scout, that computes on behalf of developers the consequences of Quick Fix recommendations. Quick Fix Scout allows developers to remove compilation errors faster, but it leverages the existing quick fix recommendations in Eclipse which does not take into account change histories. Perelman et al. [8] defined a language of partial expressions that makes type-directed predictions to help developers find method names based on the given arguments, arguments based on the method name, or to complete binary expressions such as assignment statements. Similarly, Duala-Ekoko and Robillard [9] developed a tool called API Explorer that help developers discover API methods or types that are inaccessible from a given API type, by leveraging the structural relationships between API elements. While such tools help developers use the *unknown* APIs, they do not help developers with the APIs they *used to know*. The tools would be useful for stable APIs, but the development process in general is inherently dynamic where the code and APIs change constantly.

Code navigation is also an active research topic. For example, Ko et al. [10] reported that developers engaged in software maintenance tasks spent up to 35% of their time navigating through the code, learning how the code works and how to modify it to complete their tasks. We believe that our temporal navigation will significantly reduce the time that developers spend in manually navigating to past versions of code. Singer et al. [11] introduced NavTracks, a code navigation tool that monitors and analyzes the navigation history of software developers as they perform their tasks, forming associations between related files. These associations are used to recommend potentially related files when, for example, a developer opens a file that she knows is relevant to a bug fix. Mäder and Egyed [12] implemented and evaluated a program editor tool with code navigation functionality augmented with requirements traceability, which allows developers to quickly identify where a requirement is implemented. While improving the speed and accuracy of development tasks, these tools still only work on *one version* of the code at a time.

Code completion and navigation, like many other development tasks, can benefit from the information about history of code changes. LaToza et al. [13] reported that 50% of developers find understanding the history of a piece of code to be a difficult problem. In fact, many researchers have extended code completion and navigation techniques with varying interpretations of historical information. Robbes and Lanza [14] used change-based information to improve code completion. They collected historical information such as the last modified or added date of a class/method, and used it to rank proposals in their tool. The modifications they consider, however, only pertain to the body of methods or classes, so deleted, moved, or renamed methods or classes are disregarded. Bruch et al. [15] introduced an intelligent code completion system that calculates each proposal's relevance in a given context, by using examples found in existing code repositories, and uses the information to filter and rank the proposals. While the system helps developers to focus only on relevant API elements, it disregards deleted elements (which can no longer be relevant in the current version). Such tools utilizing historical information can be more accurate and useful, but they effectively *compress* the entire history for the current (*single*) version of code. Our approach, in contrast, will allow developers to explore any version in the history.

VI. Conclusion and Future Work

We envision a novel approach for providing seamless code completion and navigation across multiple versions of a codebase. Realizing our vision requires addressing three key challenges. First, to support large projects with long histories, our approach requires the ability to efficiently search large code histories. This challenge can be addressed by precomputing the results for historical commits and/or storing these results on a centralized server shared by a development team. Second, the IDE should help developers understand *how* program elements have changed and should now be used, as opposed to simply presenting *which* elements have changed. For example, a method whose name they may remember could have been renamed, moved, inlined, or even replaced by a field. Many of these changes are refactorings, so we can leverage approaches that infer refactorings from history. Moreover, when an IDE supports recording the automatic refactorings initiated by developers, we can store this information in VCS and later use it to accurately associate historical elements with their current use forms. Third, we need an intuitive user interface to incorporate historical information into modern IDEs.

While our current emphasis is on temporal versions of projects stored in VCS, we believe the user interfaces we create naturally extend to other situations involving multiple versions of a codebase, such as parallel branches of a project or even live analysis of the ongoing simultaneous work on projects with multiple developers. We believe our approach *opens a new line of research* exploring ways to improve developer productivity while working on large, real-world, and always fluid codebases with multiple versions.

Acknowledgments

We thank Johnston Jiaa for his help with prototype implementation and discussions. This work is partially supported by the National Science Foundation under Grant Nos. CCF-1213091, CCF-0845628, and CCF-0746856. Sam Harwell is supported by Tunnel Vision Laboratories, LLC.

References

[1] "Eclipse Code Recommenders," http://eclipse.org/recommenders/.
[2] D. Rothlisberger, M. Harry, W. Binder, P. Moret, D. Ansaloni, A. Villazon, and O. Nierstrasz, "Exploiting Dynamic Information in IDEs Improves Speed and Correctness of Software Maintenance Tasks," *TSE*, 2012.
[3] M. Härry, "Augmenting Eclipse with Dynamic Information," Master's thesis, University of Bern, 2010.
[4] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based Reconstruction of Complex Refactorings," in *ICSM*, 2010.
[5] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated Detection of Refactorings in Evolving Components," in *ECOOP*, 2006.
[6] C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers, "Active Code Completion," in *ICSE*, 2012.
[7] K. Muşlu, Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Speculative Analysis of Integrated Development Environment Recommendations," in *OOPSLA*, 2012.
[8] D. Perelman, S. Gulwani, T. Ball, and D. Grossman, "Type-Directed Completion of Partial Expressions," in *PLDI*, 2012.
[9] E. Duala-Ekoko and M. P. Robillard, "Using Structure-based Recommendations to Facilitate Discoverability in APIs," in *ECOOP*, 2011.
[10] A. J. Ko, B. A. Myers, S. Member, M. J. Coblenz, and H. H. Aung, "An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks," *TSE*, 2006.
[11] J. Singer, R. Elves, and M. Storey, "NavTracks: Supporting Navigation in Software Maintenance," in *ICSM*, 2005.
[12] P. Mäder and A. Egyed, "Do Software Engineers Benefit from Source Code Navigation with Traceability? – An Experiment in Software Change Management," in *ASE*, 2011.
[13] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining Mental Models: A Study of Developer Work Habits," in *ICSE*, 2006.
[14] R. Robbes and M. Lanza, "Improving Code Completion with Program History," *ASE*, 2010.
[15] M. Bruch, M. Monperrus, and M. Mezini, "Learning from Examples to Improve Code Completion Systems," in *ESEC/FSE*, 2009.