Enhancing Software Development with IDE-Managed Multiple Codebases

Kıvanç Muşlu
Computer Science & Engineering
University of Washington
Seattle, Washington, USA
kivanc@cs.washington.edu

Abstract—Modern integrated development environments (IDEs) support one live codebase at a given moment, which imposes limitations on software development. Consider a developer who needs to fix a bug. (1) The developer might search through the version control system (VCS) to find the code change that introduced the bug. Depending on the frequency of commits, this might yield a big changeset. A copy codebase could automatically maintain a finer-grained development history, which would more accurately localize the buggy change. (2) While implementing the fix, the developer might use program analysis tools to guide her. Without a copy of the developer’s codebase most of these analyses cannot be run in parallel with the development, as they assume that the codebase they are running is not modified concurrently. Consequently, the developer needs to manually invoke these analyses and wait until they terminate. (3) Non-trivial bug fixes might require implementation and comparison of multiple solution candidates (codebases). However, having access to only one live codebase at any given moment makes it harder to maintain, analyze, and compare these solutions.

The current workflow for software development (Figure 1) supports a limited number of codebases: one live codebase (current code) and manually- or VCS-managed historical codebases. In this paper, we propose tools and techniques that would add IDE support for multiple codebases. Our thesis is that an incrementally-maintained copy of the developer’s codebase can be used to enhance software development by adding support for continuous analysis execution, and making it easier to systematically explore the development history and maintain multiple development versions.

I. INTRODUCTION

Modern integrated development environments (IDEs) support one live codebase at a given moment, which imposes limitations on software development. Consider a developer who needs to fix a bug. (1) The developer might search through the version control system (VCS) to find the code change that introduced the bug. Depending on the frequency of commits, this might yield a big changeset. A copy codebase could automatically maintain a finer-grained development history, which would more accurately localize the buggy change. (2) While implementing the fix, the developer might use program analysis tools to guide her. Without a copy of the developer’s codebase most of these analyses cannot be run in parallel with the development, as they assume that the codebase they are running is not modified concurrently. Consequently, the developer needs to manually invoke these analyses and wait until they terminate. (3) Non-trivial bug fixes might require implementation and comparison of multiple solution candidates (codebases). However, having access to only one live codebase at any given moment makes it harder to maintain, analyze, and compare these solutions.

The current workflow for software development (Figure 1) supports a limited number of codebases: one live codebase (current code) and manually- or VCS-managed historical codebases. In this paper, we propose tools and techniques that would add IDE support for multiple codebases. Our thesis is that an incrementally-maintained copy of the developer’s codebase can be used to enhance software development by adding support for continuous analysis execution, and making it easier to systematically explore the development history and maintain multiple development versions.

II. DEVELOPMENT WITH MULTIPLE LIVE CODEBASES

Figure 2 depicts the workflow we envision when IDEs will have support for multiple codebases. In Section III, we describe a replication framework that creates and incrementally maintains a copy of the developer’s codebase. Figure 3 depicts how the replication framework, and other parts of our proposal build on top of each other. Then, we focus on three benefits of this replication framework: IDE support for (1) continuous analysis feedback for the current code, (2) exploration and analysis of the development history, and (3) maintenance and analysis of multiple codebases.

Most program analyses are not continuous — that is the analysis does not automatically update its results as the developer’s code changes. The developer needs to manually invoke these analyses and wait until they terminate, which interrupts development. So developers may use these analyses less frequently than ideal. Section IV identifies the obstacles for continuous analysis integration and explains how existing non-continuous analyses can be wrapped into continuous ones using our replication framework.

IDEs offer limited support for storing and exploring development history. A complete and fine-grained development history is important for understanding software evolution and regression bugs. For example, delta debugging [2] works better with finer-grained changes. Our replication framework keeps track of all changes to the source code, enabling creation of a history where no information is lost. This IDE-managed fine-grained development history can be explored by the developer...
or another tool in the granularity appropriate for the current task. Section V identifies the design decisions for such a fine-grained history framework, and discusses one visualization application that can be built on such framework.

A developer cannot maintain and analyze multiple development versions simultaneously in the IDE. Using a VCS, the developer can change the version that is being edited, make some modifications, and run an analysis on this version. However, it is not possible to make the same modification, or run an analysis, on multiple versions at the same time. Section VI introduces Layers, another extension to our replication framework, which adds maintenance and analysis support for multiple codebases in the IDE.

III. COPY CODEBASES WITH REPLICATION FRAMEWORK

The main goal of our replication framework is to maintain a buffer-level copy of the developer’s codebase where an arbitrary computation can be run without interference from concurrent developer edits. The framework keeps track of the developer’s changes to the source code, at the buffer level, and duplicates these changes to a copy codebase. Replication is incremental and the amortized cost of maintaining the copy codebase is negligible [3]. A program analysis that extends our framework can get the exclusive ownership of the copy codebase — which lets this analysis to run on a development snapshot and ignore any conflicting developer edits.

Our framework also supports impure program analyses, which modify the source code. After getting exclusive ownership, an impure analysis can modify the copy codebase as it sees fit. None of these changes will be seen by the developer. Currently, an impure analysis needs to revert its changes before it gives the ownership back to our framework.

Our framework is implemented as an Eclipse plug-in, Solstice¹. The remainder of the paper discusses applications of our framework that focus on continuous analysis execution, systematic exploration of the development history, and maintenance and analysis of multiple development versions.

IV. IDE-INTEGRATED CONTINUOUS PROGRAM ANALYSES

Most program analyses, such as Findbugs [4], PMD², and unit testing, assume that the source code does not change while the analysis is running. A continuous analysis does not make this assumption; it automatically updates its results as the source code changes. Some of the most widely used continuous analyses, such as Eclipse’s incremental compiler, are implemented by the IDE developers after long design and optimization considerations.

It is difficult to convert an arbitrary analysis into an IDE-integrated continuous one. Integration through Eclipse project builders requires the developer to save the edits in the buffer, as most analyses work on the source or compiled code on the disk. The continuous testing Eclipse plug-in [5] suffers the same limitation. So, there is inconsistency between what the developer sees and what the analyses work on.

The problems are worse for impure analyses. A continuous impure analysis cannot run on the developer’s code as the modifications done by the analysis will confuse the developer. All continuous impure analyses that we know of use copy codebases [1], [6], [7]. On the other hand, non-continuous impure analyses [8]–[11] block development while they run.

Wrapping a non-continuous analysis into an IDE-integrated continuous one is trivial using our replication framework. The analysis author implements a wrapper program analysis that extends our framework. After each developer edit, the wrapper automatically gets exclusive ownership of the copy codebase, runs the non-continuous analysis on this copy, and shows the results to the developer. If there is a concurrent developer edit while the analysis is running, the wrapper can restart the analysis after the edit is reconciled, or continue running the analysis and annotate analysis results as stale. Stale results from a recent development snapshot may still be useful.

¹https://bitbucket.org/kivancmuslu/solstice
²http://pmd.sourceforge.net
We have shown [3] that it is easy to write continuous analysis wrappers. We implemented three wrappers, where each wrapper took less than 2 weeks and 500 LoC on average.

Work & evaluation plan: Using Solstice, we want to re-implement Quick Fix Scout\(^3\) to compare implementation effort without and with Solstice.

V. Systematic Exploration through a Fine-Grained Development History

Software development is rarely linear. Developers might interleave multiple tasks. For example, a developer might start implementing module \(m_1\), abandon this implementation, and implement another module \(m_2\). Assume that at this point, the developer realizes that she actually needs \(m_1\). Most IDEs do not have undo capabilities that can bring back \(m_1\) without removing \(m_2\). The process becomes more complex if there are other operations between the implementation of these modules. The action might even be irreversible if the developer restarted the IDE in the middle, as most IDEs only store the development history for one session. Without manual management of development history, the developer might need to go through hoops to achieve this goal, or even worse, might need to re-implement \(m_1\) from scratch.

To solve this problem, we propose a fine-grained history framework (FGHF), which stores the complete development history as fine-grained commits, automatically in the background. Implementation of FGHF on our replication framework is straightforward; FGHF would be an extension that commits the contents of the copy codebase periodically.

The StoryTeller VCS [12] showed that the development history can be replayed exactly in the future by committing every keystroke. A larger commit granularity might be desirable for systematic exploration of the development history. Most IDEs merge all keystrokes between developer pauses into a large edit, which is a good candidate for the commit granularity.

For a rich exploration experience, we propose a Dependency Aware [13] selective Tree-basedUndo Model (DATUM) built on our FGHF. In DATUM, the development history starts as an empty root node, each edit creates a child, each undo creates an empty sibling, and each redo goes back to the previous child of the current node. The tree-based representation permits exploration of arbitrary siblings (e.g., multiple undoes, redoes, and edits) easily. This model supersedes the current undo model implemented in most IDEs. For the modules example, if the developer used DATUM, she could undo the removal of \(m_1\) (redo it), without losing \(m_2\).

Modern IDEs still implement the linear undo model that can only undo the last operation. Researchers provided lots of alternatives, including the script based undo model [14], which adds support for multiple undo operations to the linear undo model, and selective undo models [13], [15], [16] that permit the developer to undo an arbitrary action in the history. Cass et al. showed that the dependency-aware selective undo model correlates the best with what developers think about undo [17]. Vim\(^4\) and Emacs\(^5\) store the development history as a tree of edits. However, their undo implementation is neither dependency aware nor selective; it only supports multiple undo/redo invocations. We believe that by combining the tree-based development representation with dependency-aware selective undo model, DATUM will be more precise and flexible at the same time.

Bird et al. [18] showed that developer-managed repositories should be mined carefully since the developers might have rewritten the development history, which misleads the mined results. The development history recorded by our FGHF does not have this disadvantage. Therefore, FGHF can also be used to improve research that mines development history, such as finding code fragments that are prone to bugs [19].

Using an incrementally-maintained copy of the developer’s codebase, it is possible to store a complete and fine-grained development history. Consequently, the developer is freed from the burden of keeping track of the development history manually. Furthermore, DATUM aims to help the developers to better explore and modify the non-linear development history.

Work & Evaluation Plan: FGHF and DATUM will be implemented on our replication framework. After the implementation, DATUM will be evaluated through case studies, where developers use DATUM and provide their experience. Improving the precision of data mining research through FGHF is not in the scope of this proposal, however it is a nice example of other uses of the complete development history.

VI. Maintaining Multiple Development Versions

Software development frequently requires the maintenance of multiple codebases. For example, a company that supports multiple versions of a program might maintain these versions separately. When a bug is detected in the common code, the fix must be applied to all affected versions. Even within the same version, a developer might want to maintain an experimental feature or bug fix separately, especially if the feature or the bug fix is not ready for release.

A popular way to maintain different development versions is to store each version in a VCS branch. However, branches are static; a developer cannot apply a common fix to multiple branches at the same time. Moreover, the branches can diverge from each other and might conflict with each other.

We propose a framework called Layers, which aims to help the developer to maintain multiple codebases in the IDE. The development starts with one layer; ‘default’. Other layers can be created as more code is written. For example, the developer can create a ‘debug’ layer, which contains extra debugging statements. A layer can be deleted or split into two, and two layers can be merged. Layers have a parent-child relationship, so a change done to a parent layer automatically propagates to the children. For our example of the company that gives support to multiple versions of a program, let us assume that the common code is implemented in the ‘core’ layer, and

\(^3\)https://quick-fix-scout.googlecode.com
\(^4\)http://vimdoc.sourceforge.net/htmldoc/undo.html
\(^5\)http://www.emacswiki.org/emacs/UndoTree
all versions are children of the ‘core’ layer. To fix the bug affecting all versions, the developer modifies the ‘core’ layer only; the fix automatically propagates to all versions.

Another advantage of Layers is the ability to detect conflicting edits in close to real time. A conflicting change between a child and a parent layer would be detected immediately during the propagation of the edit. Additionally, as the IDE is aware of all available layers, it can speculatively merge these layers and inform the developer when there is an incompatibility between two layers. The developer can resolve the conflict immediately, keep the warning for a future resolution, or ignore the warning if those layers will never merge.

Layers can be implemented on our replication framework. Layers operations would be translated to VCS operations and invoked on the copy codebase. For example, creation of a new layer corresponds to creation of a new branch and updating the internal parent-child relation so that whenever the parent layer is modified, the same modification is also applied to this new branch. After acquiring the ownership of the copy codebase, Layers can check for pairwise conflicts, and run other analyses on multiple layers, bringing continuous analysis execution to multiple development versions.

The description and usage of Layers is similar to the one described by Nita [20]. However, the framework we envision is different in terms of underlying implementation and representation. We think that the framework should be implemented on top of a VCS instead of as a tagged character model (TCM). This decision changes the way some operations are defined, especially the ‘merge’ operation. Textual conflicts are not possible under a TCM whereas our system would propagate the underlying VCS conflicts to the developer.

Software product line (SPL) engineering [21] helps the companies to maintain multiple software based on a core product. Layers operates at a much finer granularity compared to SPL and has other use cases. Layers can be used to separate multiple implementation components, explore implementation strategies, and visualize impure analyses.

With Layers, the developer will be able to maintain multiple development versions in the IDE. Implementing Layers on an incrementally-updated copy codebase lets us run arbitrary computation on these layers, which brings continuous analysis execution to multiple development versions.

Work & Evaluation Plan: Layers will be implemented on our replication framework. Then, we will conduct case studies, where developers use Layers and provide their experience.

VII. Contributions

This paper proposes that IDE support for multiple codebases can aid software development. We first note that a replication framework can efficiently add support for an incrementally-maintained copy codebase. Using the replication framework, it is easy to implement IDE-integrated continuous analyses, that run on the copy codebase, to provide constant feedback to the developer for the task at hand. Then, we extend the replication framework to an fine-grained history framework that helps the developer to explore the development history easier and adjust this history when needed. Storing a complete development history on the copy codebase extends continuous analysis execution to historical development versions. Finally, we propose another extension to the replication framework, Layers, which helps the developer to maintain and analyze multiple live codebases at the same time.

REFERENCES