Integrating Systematic Exploration, Analysis, and Maintenance in Software Development

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 to software development. For example, with only one codebase, the developer must pause development while running tests, or a static analysis, as any edit could invalidate the ongoing computation. Were the IDEs supported a copy of developer's codebase, the analyses could have run on this copy, in parallel with the development process. In this paper, we propose techniques and tools that integrate multiple live codebases support to the software development process. Our hypothesis is that IDE support for multiple live codebases can provide a richer development process and aid developers.**

I. Introduction

Modern integrated development environments (IDEs) support one live codebase at a given moment, which imposes limitations to software development. Consider a developer who needs to fix a bug. It is common for the developer to explore the development history to find the bug. However, if the developer is not an avid version control system (VCS) user, this process can become very difficult, even impossible as the IDEs have little exploration support for the historical codebases. While implementing the fix, the developer might want to use static and dynamic analysis tools to guide her. However, with only one live codebase, most of these analyses cannot be run in parallel with the development as each developer edit might invalidate the ongoing computation. Non-trivial bug fixes require implementation and comparison of multiple candidate solutions (codebases). However, having access to one live codebase for any given moment makes it harder to maintain, analyze, and compare these solutions.

The current workflow for software development (Figure 1) supports only one live codebase (current code) and limited number of manually managed historical codebases. In this paper, we propose tools and techniques that would add IDE support for multiple codebases. Our thesis is that IDE support for multiple codebases can aid developers and provide a richer development experience with more powerful exploration of the development history, continuous analysis feedback, and easier maintenance and analysis of multiple codebases.

II. Development with Multiple Live Codebases

Figure 2 depicts the workflow we envision when IDEs will have support for multiple codebases. We believe that a development workflow with IDE support for exploration

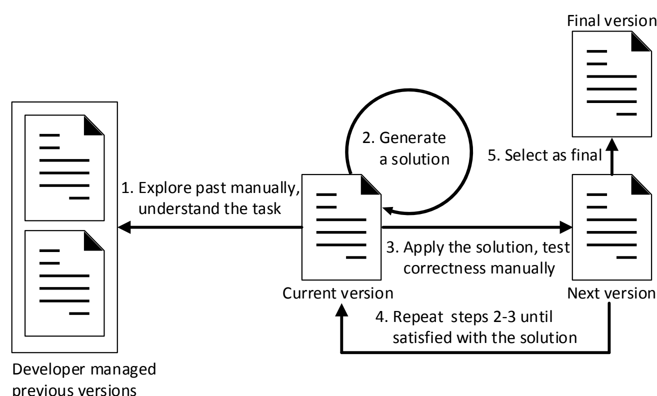

Fig. 1: Representation of current software development workflow. Modern IDEs support one live codebase.

of the development history, continuous analysis feedback for the current code, and maintenance and analysis of multiple codebases will make development easier. This section explains how we came up with this development process, and provides a roadmap for the rest of the paper.

First, we note that most of the static and dynamic analysis — even the pure ones that have IDE integration — are not continuous. These analyses, such as testing, help the developers to find bugs, generate solutions, and implement these solutions. However, the fact that the developer needs to manually invoke them, and wait until they terminate interrupts the development. Consequently, the developers use these analyses less often, and get reduced benefits. Section III identifies the obstacles for continuous analysis integration, and proposes a replication framework that removes these obstacles by creating and maintaining a duplicate of the developer's codebase. This replication framework adds support for the second live codebase, which is the core of our work.

Then, we note that there is limited support for exploring development history in IDEs. Exploring development history is important for understanding software evolution and regression bugs. We argue that our replication framework can be extended to keep a complete and fine-grained development history. The framework already keeps track of the changes to the source code, combining this with a background VCS would create a history where no information is lost. Section IV identifies the

978-1-4673-3076-3/13 © 2013 IEEE

ICSE 2013, San Francisco, CA, USA
Doctoral Symposium

Accepted for publication by IEEE. © 2013 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

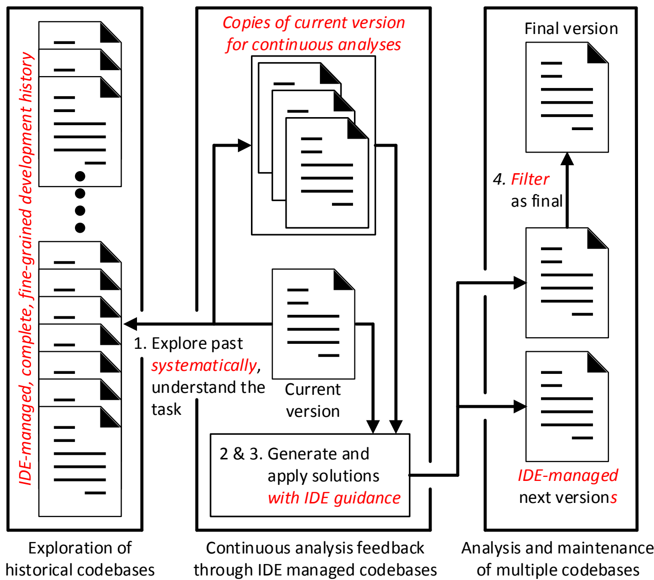

Fig. 2: Representation of the hypothetical software development when IDEs support multiple codebases. IDE manages multiple codebases to help the developer systematically explore the development history, get continuous analysis feedback for the current code, and analyze and maintain multiple codebases. The expected changes in developers actions are shown in red color and emphasized.

design decisions for such VCS, and discusses one visualization application that can be built on top of this VCS.

Finally, we note that IDEs have no native support for maintaining multiple development versions. The developers must manually use a VCS-like approach to maintain each codebase. Switching between codebases require the execution of VCS commands (or manual operations), which slows down the development. More importantly, there is no support to run an analysis on multiple development versions; the current practice can focus on one development copy at a time. To compare multiple codebases, the developer must go through all of them and apply the evaluation criteria to each, one by one. Section V introduces Layers, as another extension of our replication framework, which adds maintenance and analysis support for multiple codebases in the IDE.

III. Enabling Continuous Code Analysis by Replicating Developer's Codebase

Static code analyses, such as Findbugs [1] and PMD[1], and dynamic analyses, such as unit testing, are vastly used by the developers and became a major part of development process. Despite their popularity, it is not a simple task to take a conventional background analysis and transform it into a continuous one. Some of the most widely used continuous analyses, such as Eclipse incremental compiler[2], are implemented by the IDE developers after long design and optimization considerations.

[1]http://pmd.sourceforge.net/

[2]http://www.eclipse.org/jdt/core/index.php

The continuous integration of third party analyses have important limitations. Integration through Eclipse project builders requires the developer to save the edits in the buffer, as most analyses work on source or compiled code on the disk. Continuous testing Eclipse plug-in [2] suffers the same limitation. So, there is inconsistency between what the developer sees and what the analyses can work on. Additionally, analyses that work on developer's code must be very fast, as each save operation chances the code on the disk and might invalidate ongoing computations. For example, FindBugs only runs on the current file (instead of project) in the continuous version of its Eclipse plug-in.

The downside is worse for impure analyses — analyses that need to modify the source code before running. Continuous impure analyses cannot run on developer's code as the intermediate changes done by the analysis will be seen by the developers and confuse them. The only continuous impure analyses that we know of all use copy codebases [3]–[5]. On the other hand, the non-continuous impure analyses [6]–[9] block the development while they run.

To ease the integration of continuous analyses to IDEs, we have created a replication framework that keeps track of developer's changes to the source code, at buffer level, and duplicates these changes to a copy codebase. Replication is incremental and the amortized cost of maintaining the copy codebase is negligible [10]. Our framework also manages the ownership of this copy codebase. Through a simple API, the analyses can get the exclusive ownership of the copy codebase — which lets the analyses run on a static development snapshot and ignore any conflicting developer edit in the meanwhile. These concurrent edits are applied to the copy codebase after the analyses produce results. The analysis authors can choose to invalidate an ongoing analysis or show the results of a recent development snapshot with extra information. Note that results from a recent development snapshot — but not the current one — can still be very useful for most of the slow analyses, such as Findbugs and PMD.

Our framework simplifies the implementation of any continuous impure analysis, too. After getting the exclusive ownership, the impure analysis can modify the copy codebase as it sees fit. None of these changes will be seen by the developer. The only additional thing that the impure analysis needs to do is to revert these changes at the end of the computation. Most impure analyses already implement this logic for the copy codebase they manage manually. Moreover, our framework might take a snapshot of the copy codebase — through VCS operations — before giving the exclusive ownership to the analysis and revert to this state after the analysis is over.

We have shown that using our framework it is easy to take a conventional analysis and transform it into a proof-of-concept IDE-integrated continuous analysis. The author of the framework implemented three proof-of-concept analyses in less than 18 hours and 700 lines of code for each analysis. We believe that using a replication framework and keeping track of the developer changes to the source code removes some of the most important barriers for continuous analyses implementa-

tion and integration with IDEs. Consequently, adding support for an extra codebase, will bring constant analysis feedback to development process.

Work & evaluation plan: Our framework is implemented as an Eclipse plug-in, Solstice[3], and is under submission. Using Solstice, we want to re-implement Quick Fix Scout[4] to better evaluate the amount of effort that our framework reduces in continuous impure analysis implementations.

IV. Systematic Exploration through a Complete Development History

Software development is almost never linear. While performing a development task, the developers might interleave the task at hand with other operations. For example, a developer might start implementing module `m1`, abandon this implementation, and implement another module `m2`. Assume that at this point, the developer realizes that she actually needs module `m1`. Most of the IDEs do not have undo capabilities that can bring back module `m1` without removing `m2`. The process becomes more complex if there are other operations between the implementation of these modules. Most IDEs only store the development history for one session, so the action might be irreversible if the developer restarted the IDE in the middle. 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 the same module from scratch.

To solve this problem, we propose an IDE-managed VCS, which keeps the complete development history as fine-grained commits, automatically in the background. Using the framework we have introduced in Section III, the implementation of such VCS is obvious; the repository can be stored in the copy codebase and the VCS can be implemented as an analysis that commits the contents of the copy codebase periodically.

StoryTeller VCS [11] showed that the development history can be replayed exactly in the future by committing every keystroke. For systematic exploration of the development history, commit granularity can be larger. Most modern IDEs merge all keystrokes between developer pauses into a large edit, which is a good candidate for the commit granularity. The development history created with this granularity will still include all relevant information for further exploration and will have less overhead.

For a rich exploration experience, we propose a Dependency Aware [12] selective Tree-based Undo Model (DATUM) built on top of our VCS. 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 of the IDEs. For the modules example, were the developer used DATUM, she could have undone the removal of module `m1` (redo it). As long as there are no dependencies between modules `m1` and `m2`, DATUM would retrieve module `m1` without losing `m2`.

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 [13], which adds support for multiple undo operations to the linear undo model, and selective undo models [12], [14], [15] that permit the developer to undo an arbitrary action in the history. Cass et al. showed that dependency-aware selective undo model correlates the best with what the developers think about undo [16]. On the other hand Vim[5] and Emacs[6] 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. [17] showed that developer-managed repositories should be mined carefully since the developers might rewrite the development history at any point, which misleads the mined results. The development history recorded by our VCS does not have this disadvantage, as the developers never interact with the VCS directly. Therefore, our system can also be used to improve research that mines development history, such as finding code fragments that are prone to bugs [18].

With an IDE-managed, complete and fine-grained development history, we take the burden of keeping track of the development history from the developer. Furthermore, DATUM aims to help the developers to better explore and modify the non-linear development history.

Work & Evaluation Plan: After the implementation of IDE-managed, fine-grained VCS, we need to formalize and implement DATUM. Once completed, DATUM will be evaluated through case studies, where developer use DATUM and provide their user experience. Improving the precision of data mining research through our VCS is out the of scope of this proposal, however lays out a nice example on other usages of the complete and fine-grained development history.

V. Maintaining Multiple Development Versions

Software development frequently requires the maintenance of multiple codebases. For example, a company that gives support to multiple versions of a program must maintain these versions separately. When a bug is detected in the common code, the fix must be applied to all affected versions. Some versions might use different libraries or operating systems, and could require different fixes. Even within the same version, a developer might want to maintain an experimental feature or bug fix separately if it is not ready for the 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

[3]https://bitbucket.org/kivancmuslu/solstice

[4]https://quick-fix-scout.googlecode.com/

[5]http://vimdoc.sourceforge.net/htmldoc/undo.html

[6]http://www.emacswiki.org/emacs/UndoTree

branches at the same time. Moreover, the branches can quickly diverge from each other and might conflict with each other. Brun et al. showed that conflicts are frequent in open source software and can be missed for a long time [4].

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' and other layers can be created as more code is written. For example, the developer can create an 'optimization' layer where some code is replaced with its optimized version and a 'debug' layer, which contains extra debugging statements. Any unnecessary layers can be deleted, two layers can be merged, and a layer can be split into two. Layers have parent-child relationship and 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 all versions are children of the 'core' layer. To fix the bug affecting all versions, the developer just modifies the 'core' layer; the fix automatically propagates to all versions.

Another advantage of Layers is the ability to detect any conflicting edit 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 framework can be implemented as an analysis running on the copy codebase of our replication framework. Layers operations would be translated to VCS operations, which would then be 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. When the IDE is idle, the framework can check for pairwise conflicts, and run other analyses on multiple layers. The results would be shown through views and markers.

The description and usage of Layers framework is very similar to the one described by Nita [19]. 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 [20] helps the companies to maintain multiple software based on a core product. Layers framework operates at a much finer granularity compared to SPL and has other use cases. Layers can be used to separate multiple implementation components, exploring implementation strategies, and visualizing impure analyses.

With the Layers framework, the developer will be able to compare multiple implementation strategies, and maintain multiple codebases, in the IDE, at the same time.

Work & Evaluation Plan: After the Layers framework is formalized, it will be implemented as an analysis on top of our replication framework. Once implemented, Layers framework will be evaluated through case studies, where the developers use the framework and provide their personal experience.

VI. Contributions

This paper proposes that IDE support for multiple codebases can aid various stages of software development. We first note that a replication framework can enable the implementation of IDE-integrated continuous analyses, which provides constant feedback to the developer for the task at hand. Then, we extend our framework to an IDE-managed VCS that helps the developer to explore the development history easier and adjust this history when needed. Finally, we propose another extension, Layers, which helps the developer to maintain and analyze multiple live codebases, at the same time.

References

[1] D. Hovemeyer and W. Pugh, "Finding bugs is easy," in *OOPSLA*, 2004, pp. 132–136.

[2] D. Saff and M. D. Ernst, "Continuous testing in Eclipse," in *ICSE*, 2005, pp. 668–669.

[3] K. Muşlu, Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Speculative analysis of integrated development environment recommendations," in *OOPSLA*, 2012, pp. 669–682.

[4] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Proactive detection of collaboration conflicts," in *ESEC/FSE*, Szeged, Hungary, 2011.

[5] "Beacon," http://blogs.msdn.com/b/msr_er/archive/2011/09/07/seif-project-crystal-receives-acm-sigsoft-distinguished-paper-award.aspx, 2011.

[6] K. Muşlu, B. Soran, and J. Wuttke, "Finding bugs by isolating unit tests," in *ESEC/FSE*, 2011, pp. 496–499.

[7] D. Schuler, V. Dallmeier, and A. Zeller, "Efficient mutation testing by checking invariant violations," in *ISSTA*, 2009, pp. 69–80.

[8] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid, "Regression mutation testing," in *ISSTA*, 2012, pp. 331–341.

[9] M. Gligoric, S. Badame, and R. Johnson, "Smutant: a tool for type-sensitive mutation testing in a dynamic language," in *ESEC/FSE*, 2011, pp. 424–427.

[10] K. Muşlu, Y. Brun, M. D. Ernst, and D. Notkin, "Making offline analyses continuous. **In Submission.**" in *ESEC/FSE*, 2013.

[11] M. Mahoney, "The storyteller version control system: Tackling version control, code comments, and team learning," in *SPLASH*, 2012, pp. 17–18.

[12] A. G. Cass and C. S. T. Fernandes, "Modeling dependencies for cascading selective undo," in *INTERACT*, 2005.

[13] J. E. Archer Jr., R. Conway, and F. B. Schneider, "User recovery and reversal in interactive systems," *TOPLAS*, vol. 6, no. 1, pp. 1–19, 1984.

[14] T. Berlage, "A selective undo mechanism for graphical user interfaces based on command objects," *TOCHI*, vol. 1, no. 3, pp. 269–294, 1994.

[15] B. A. Myers and D. S. Kosbie, "Reusable Hierarchical Command Objects," in *CHI*, 1996, pp. 260–267.

[16] A. G. Cass, C. S. T. Fernandes, and A. Polidore, "An empirical evaluation of undo mechanisms," in *NordiCHI*, 2006, pp. 19–27.

[17] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, "The promises and perils of mining git," in *MSR*, 2009, pp. 1–10.

[18] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy, "Change bursts as defect predictors," in *ISSRE*, 2010, pp. 309–318.

[19] M. Nita, "Editing text versions with layers," http://homes.cs.washington.edu/~marius/papers/layers/layers-draft.pdf, UW, Tech. Rep., 2010.

[20] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., 2005.