Speculative Analysis: Exploring Future Development States of Software

Yuriy Brun*, Reid Holmes†, Michael D. Ernst*, David Notkin*
*Computer Science & Engineering
University of Washington
Seattle, WA, USA
†School of Computer Science
University of Waterloo
Waterloo, ON, Canada
{brun,mernst,notkin}@cs.washington.edu, rtholmes@cs.uwaterloo.ca

Abstract

Most software tools and environments help developers analyze the present and past development states of their software systems. Few approaches have investigated the potential consequences of *future* actions the developers may perform. The commoditization of hardware, multi-core architectures, and cloud computing provide new potential for delivering apparently-instantaneous feedback to developers, informing them of the effects of changes that they may be considering to the software.

For example, modern IDEs often provide "quick fix" suggestions for resolving compilation errors. Developers must scan this list and select the option they think will resolve the problem. Instead, we propose that the IDE should speculatively perform each of the suggestions in the background and provide information that helps developers select the best option for the given context. We believe the feedback enabled by speculative operations can improve developer productivity and software quality.

Categories and Subject Descriptors
D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.6 [**Software Engineering**]: Programming Environments

General Terms
Design

Keywords
Speculation, developer awareness, recommender system, version control, quick fix, IDE

1. Introduction

The world's ever-growing dependence on software and software-intensive systems drives the continued demand for increased software quality and software engineering productivity. Many approaches for improving software quality and productivity are based on analysis of the current development state, and sometimes past development states, of the software. A development state represents a snapshot of the software system's source code at an instant in time. (Development state could also include relevant configuration files, library versions, makefiles, and other data affecting the software under development.) For example, most bug detection approaches examine the current development state, and most analyses underlying regression testing also use a past development state of the software. However, few, if any, approaches systematically explore potential *future* development states of a software system under development.

We call the exploration of potential future development states *speculative analysis*, by analogy with speculative execution (e.g., branch prediction and cache pre-fetching). Speculative analysis expends CPU cycles to analyze a possible future development state. This offers two distinct advantages. If a developer edits the source code into that development state, then the analysis results are available immediately, which saves valuable human time. If the developer is choosing what code edit to perform, then the analysis results can inform the decision, guiding the developer to the best choice. This paper focuses primarily on the latter benefit.

We believe that speculative analysis can help developers make better decisions — leading to increased software quality and developer productivity — by providing them with information about the consequences of performing each of a set of likely operations *before* they execute any of those operations. As an example, modern interactive development environments (IDEs) offer "quick fix" suggestions for resolving compilation problems in source code. The IDE could also indicate the effects of each quick fix suggestion: whether the system compiles, whether its tests pass, or whether the code can be merged with other developers' changes.

Software tools have always used computational resources to improve the effectiveness of software developers. Increased processing power, in the form of multi-core architectures and Internet-based cloud computing, is dramatically decreasing the cost and increasing the availability of computing, making a new set of trade-offs worthwhile to consider. We believe the time is ripe for speculation-based analysis for software development.

Section 2 discusses the unexplored nature of analysis over future software development states. Section 3 then sketches two concrete examples of speculative analysis. Section 4 identifies some challenges to achieving speculative analysis. Section 5 places our research in the context of related work. Finally, Section 6 concludes.

2. Future: An Unexplored Analysis Dimension

Most existing developer tools and techniques focus on using computational resources to explore the *present* development state of the software. For example, (1) continuous

FoSER 2010, November 7–8, 2010, Santa Fe, New Mexico, USA.

testing executes tests in the background, while developers work, quickly informing them whether a change has broken a test [20], (2) development environments, such as Eclipse and VisualStudio, suggest automated or semi-automated *quick fixes* that are likely to remove specific compilation errors, and (3) temporal property mining and monitoring can automatically infer and enforce software models, helping developers discover model violations and bugs [10].

Two classes of techniques have also leveraged the *past* development states, often along with the present state. The first class involves comparison between the previous and current development state to identify unexpected differences. For example, regression testing identifies executions that have different outcomes than previous development states of the same software [1]. The second class examines past development states to gain insight that can be used during future software evolution tasks. For example, past development history can be used to recommend files that should change together [24, 25], identify potentially-reusable source code [6], or empirically validate actual development practice [16].

We propose a third class of software tools and environments, those that speculate and explore potential *future* development states of software to provide developers with useful information about those states. Figure 1 shows a rough (and incomplete) representation of some techniques used to improve software development process efficiency and software quality. The figure indicates an untapped space of development states that we claim can be used to help developers.

Future development states of a piece of software are unknown, making exploring and exploiting them difficult. Some such explorations are likely infeasible due to the expansive required resources. However, some others require only a reasonable amount of computation, as we suggest in our examples in Section 3. Choosing the right states to explore is a significant research challenge, as we discuss in Section 4.1.

Several genetic and evolutionary programming techniques attempt to automatically improve a program by searching through possible future development states using an objective function to drive the search [23]. One challenge for such techniques is that the objective function of a software developer is often implicit. For example, while it may be explicit that a developer is planning to fix a bug or add a feature, there is rarely an explicit and precise definition of the expected improved behavior. A developer fixing a memory leak is unlikely to have, for example, a formal specification of how the program should behave without the leak. We argue that speculative analysis can aid developers in situations with implicit or poorly defined objective functions by speculating about the immediate future, largely characterized by operations the developer is likely to apply, and analyzing the results of executing those operations (see Section 3 for examples). Such speculative analysis may yield useful information regarding which of the operations the developer should pursue.

Thus, we suggest that some of the additional processing power available to developers today should be used to explore the future. Our proposed speculative analysis techniques consider the decisions developers are likely to make and provide the developers with information about the consequences of making those decisions. Speculative analysis will provide concrete, precise data to developers who today

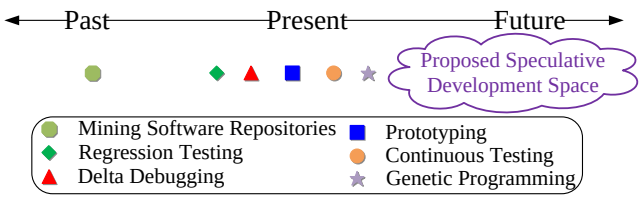

Figure 1: Today's techniques use the present and past development states of the software to improve developer productivity and software quality. We propose speculative analysis techniques that explore *future* development states of the software.

make important decisions based predominantly on experience and intuition.

In a very general sense, in addition to using cycles to learn more about the current development state of the software, we propose a complementary approach in which cycles are used over a theoretically unbounded set of possible future development states of the software. Running some existing analyses, and, potentially, new appropriate analyses on those future states will augment the developers' understanding of their software, provide additional information regarding the decisions they must make, and potentially improve the quality of those decisions to make the development processes more efficient and result in higher-quality software.

One way to express the benefits of speculative analysis is that, to the first order, today's techniques help developers by using cycles to analyze a specific version of a program to learn more and more about that one version. Speculative analysis leverages the fact that, from the point of view of a given development state, there are other places to invest computational cycles. Specifically, most useful software systems will have a large number of subsequent versions — fixes, enhancements, adaptations, etc. We see potential in using additional computational cycles to compute possible new variations of a program and to analyze properties of those versions (though perhaps not as deep a set of properties as for the current version), as a way to aid developers in evolving the software more effectively. One shortcoming of performing deeper and deeper analysis of a single development state is that the costs rise and the benefits often diminish as the analysis deepens. While speculative analysis would likely apply shallower analyses to multiple development states, such analyses across a set of possible future development states is likely to produce significant information at a lesser cost.

3. Speculative Analysis Examples

We now suggest two concrete examples of speculative analysis in development environments. These examples are feasible to implement today, and, if done correctly, can improve the developers' experience right away. The idea of speculative analysis is not limited to these concrete examples. We envision that one impact of speculative analysis will be exploring substantial and far-reaching changes to the software, such as automated code generation or bug detection and removal [17, 23].

Modern programming environments provide a vast number of options to developers, such as code completion, refactorings, and quick fixes. Developers choose from a set of alternative potential actions using their intuition; they then

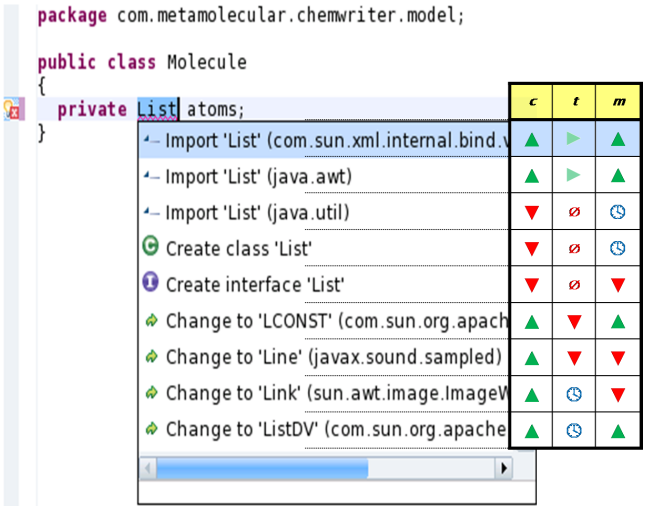

Figure 2: A speculative analysis technique can augment Eclipse's quick fix menu with an indication of the consequences of each fix. For each possible quick fix, the box on the right indicates whether compilation, testing, and version-control merging would succeed or fail. The clock icon indicates that the IDE is still computing the result and the ø symbol means "not applicable" — testing cannot proceed if compilation fails.

evaluate the outcome of their selection to validate that their choice was effective. Problems may become apparent during compilation, testing, merging of changes from the version control system, or other development activities. A developer can perform some or all of these activities, but should not be forced to remember to do so nor to wait while the task executes. Even more importantly, a developer who unexpectedly encounters a difficulty or a dead-end must either return to the original decision point (e.g., via undo or version control revert) and redo work — or deal with the consequences of the poor decision and make suboptimal adjustments to the code simply because of the difficulty of returning to the original decision point and redoing work.

3.1 Speculative Quick Fix

Speculative analysis can augment Eclipse's quick fix mechanism. When encountering compilation errors, Eclipse can suggest a set of changes (quick fixes) to the code that may resolve those errors. A speculative tool can execute those quick fixes and report pertinent, precise information on the outcomes of those executions.

Figure 2 shows a mockup that augments a screenshot of Eclipse displaying quick fixes. To the right of the quick fix options, we have added indications of the consequences of each option on compilation, testing, and merging operations. By speculating on these options while the developer is working, the approach can deliver context-sensitive information in a proactive, online fashion; fields that have not yet been speculatively analyzed are represented by the clock icon and dynamically fill in as the speculative analysis tool computes them.

3.2 Speculative Version Control

Developers increasingly use distributed version control systems, such as Git or Mercurial, to manage the resources of their projects. One of the defining characteristics of these systems is that every developer can work on his or her own independent workspace. The main benefit of this independence is that developers can work in isolation; unfortunately, this can cause problems when developers merge their activities. These merge problems can arise in the form of explicit textual merging conflicts, new compilation errors in the merged version, or tests failing after a merge that previously executed successfully.

We wish to improve the developers' understanding of how their workspaces relate to the master version and to others' workspaces. We propose that a tool can speculatively merge each developer's workspace with other relevant workspaces in the background, deriving contextualized feedback about the potential effects of various version control operations. If the speculative analysis indicates that merging would not proceed cleanly, then developers can decide whether to address it quickly or to temporarily avoid merges. If the speculative analysis indicates clean merges, then developers can merge immediately, reducing the likelihood of future merge problems. We have already begun building and evaluating the potential benefits of such a speculative tool [4].

4. Speculative Analysis Research Challenges

Building an effective speculative analysis tool raises a number of challenges. In this section, we describe some of these challenges, categorized as research challenges, technical challenges, and driving questions.

4.1 Research Challenges

A key research challenge is identifying the breadth of the speculation: a set of possible future actions a developer might take. The set of possible future development states is unbounded, but a tractable speculative analysis can only consider a limited set of them. The two tasks described in Section 3 achieved their reductions by considering only specific, likely actions. Quick fix speculation considered only those actions (a small set) that can be recommended by the quick fix mechanism. Version control speculation considered only one common action, merge, applied against all pairs of development workspaces. Other ways to reduce the size of the set could include Bayesian predictions of likely user actions, as just one example. Our work will rely on, or extend if necessary, previous work on predicting user actions [3, 5, 7, 8, 14, 15]. We do not suggest that there is a *correct* set of states that should be considered; indeed different sets of development states would likely be investigated by different analyses, as in our two included examples.

In addition to properly limiting the breadth of the speculation, our mechanisms must limit the depth. Our two tasks considered only a single action (then validated it, for example, via compilation and testing), as opposed to sequences of actions. It would be possible to go deeper. For example, speculative quick fix could consider what other quick fix suggestions would become available after executing a given quick fix action, and speculate those suggestions' executions as well. It is likely that, in some speculative analysis cases, the utility of some action is not immediately visible until more actions are performed. Therefore, speculative analysis must strike a proper balance between the breadth and depth of its search.

Additionally, developers may be more likely to perform some actions, while other actions may be more informative than others. Since computation is limited, a speculative

analysis must not only take into account the breadth and depth of its search, but also the potential utility of exploring the actions. Inevitably, some computation will be expended on actions that the user never performs. This computation is not necessarily wasted: the information it computes helps the developer to avoid a bad development state. Further, the computation is not wasted if, overall, the cheap cycles cost less than the savings in expensive developer time.

Finally, we have described only two example uses of speculative analysis. The space of potential speculations and speculative analyses is quite large and identifying the right ones to most effectively and positively influence developers is a deep and exciting challenge.

4.2 Technical Challenges

Speculative analysis requires significant computation to explore a large number of states. Thus, it is important to use the available cycles efficiently. Speculative analysis computational costs can be reduced by leveraging the inherent similarity of different future development states. It is feasible, for example, that an analysis can be performed on several similar development states far more efficiently than performing it independently on each state. This challenge involves understanding how certain analyses (such as compilation and testing) can be parallelized over different, but similar, development states on multi-core architectures and Internet-based cloud computing resources. As another example, some of the operations that led to ignored speculated development states may be reapplied to new states, and the precomputed analysis may be reapplied without being recomputed.

Two other technical challenges for speculative analysis are rapid cloning of development states of a project and preventing undesired interactions among these clones via shared resources, such as a lock, the file system, etc.

4.3 Driving Questions

Three questions drive the research agenda for speculative analysis:

First, will developers benefit from knowing information about the potential future development states of their software? If developers make decisions faster (or make better decisions) about which operations to apply to their projects, the end result could be improved developer productivity or higher-quality software than is achieved today.

Second, with abundant information available about a large number of speculated development states, are there effective ways to reduce and present the information to developers? A poor presentation can easily mask the advantages of speculative analysis and may even impede developer productivity by overwhelming the developer with redundant or uninteresting information.

Third, in what situations, if any, will speculative analysis effectively complement existing deep analyses of the current development state? These approaches will necessarily compete for computational cycles: does software productivity and quality improve more by making a marginal investment in single-state analysis or by speculative analysis? Single-program analysis will, at some point, provide sufficiently small returns to make the investment in speculative analysis worthwhile. We may be approaching that situation now. This suggests that speculative analysis may already have an opportunity to be more beneficial at the margins.

5. Related Work

A significant body of work exists on ways to increase developer awareness and assist developers in making better decisions, particularly about collaborative development. Mailing lists and chat systems can increase collaborative team member awareness [13]. FASTDash is an interactive visualization that helps people understand what other team members are doing by producing a spatial representation of the shared code base [2]. CollabVS can identify conflicts by analyzing dependencies among program elements in developers' workspaces [9]. Palantír shows who is changing which artifacts, and by how much [22]. Sarma provides a comprehensive classification of collaborative tools for software development [21].

While the work on analyzing past and current states of development is too numerous to mention, we outline some of the research on exploring present states the instant they appear, as it is most closely related to ours. Modern programming environments perform continuous compilation, providing the developer with rapid (and usually unobtrusive) feedback about compilation errors. Continuous testing reduces the time and energy required to keep code well-tested and prevent regression errors from persisting uncaught for long periods of time by continuously running regression tests in the background [18, 19, 20, 11]. Real-time integration may decrease developers' hesitation in committing changes [12]. Finally, recent research into automated generation of bug fixes pushes ever closer to exploration of future development states [17, 23].

6. Conclusion

Speculative analysis offers a new way to improve developer productivity and software quality by providing a new source of information that is complementary to existing approaches. Creating tools that perform speculative analysis is feasible, given the emergence of cheap computation via multi-core architectures and cloud computing.

Our vision of speculative analysis is quite broad. As long as developers make poorly-informed decisions, speculative analysis has the potential to provide pertinent information to help them make better decisions. As the field advances, the potential for applying speculative analysis will only increase — more knowledge and more analyses will provide opportunities for speculation over different development states.

The quick fix and version control examples provide a taste of what speculative analysis techniques may look like. However, they are only two points in the very large space of possible speculative analyses. There are an unbounded number of possible future systems, which provides a rich research space limited primarily by our imaginations, computational resources, and developers' cognitive constraints.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant #0937060 to the Computing Research Association for the CIFellows Project, by the National Science and Engineering Research Council Postdoctoral Fellowship, by Microsoft Research through the Software Engineering Innovation Foundation grant, and by IBM through a John Backus Award.

References

[1] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY,

USA, 2008.

[2] Jacob T. Biehl, Mary Czerwinski, Greg Smith, and George G. Robertson. FASTDash: a visual dashboard for fostering awareness in software teams. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI07)*, pages 1313–1322, San Jose, CA, USA, April 28–May 3, 2007.

[3] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE09)*, pages 213–222, Amsterdam, The Netherlands, August 24–28, 2009.

[4] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Speculative identification of merge conflicts and non-conflicts. Technical Report UW-CSE-10-03-01, University of Washington, 2010.

[5] Andrea Bunt and Cristina Conati. Probabilistic student modelling to improve exploratory behaviour. *User Modeling and User-Adapted Interaction*, 13(3):269–309, 2003.

[6] Gianluigi Caldiera and Victor R. Basili. Identifying and qualifying reusable software components. *IEEE Computer*, 24(2):61–70, 1991.

[7] Eugene Charniak and Robert P. Goldman. A bayesian model of plan recognition. *Artificial Intelligence*, 64(1):53–79, 1993.

[8] Laura A. Dabbish, Robert E. Kraut, Susan Fussell, and Sara Kiesler. Understanding email use: predicting action on a message. In *Proceedings of the SIGCHI conference on Human factors in computing systems (CHI05)*, pages 691–700, Portland, Oregon, USA, 2005.

[9] Prasun Dewan and Rajesh Hegde. Semi-synchronous conflict detection and resolution in asynchronous software development. In *Proceedings of the 10th European Conference on Computer Supported Cooperative Work (ECSCW07)*, pages 159–178, Limerick, Ireland, September 24–28, 2007.

[10] Mark Gabel and Zhendong Su. Online inference and enforcement of temporal properties. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE10)*, pages 76–85, Cape Town, South Africa, May 2–8, 2010.

[11] David Samuel Glasser. Test factoring with `amock`: Generating readable unit tests from system tests. Master's thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, USA, August 21, 2007.

[12] Mário Luís Guimarães and António Rito-Silva. Towards real-time integration. In *Proceedings of the 3rd International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE10)*, pages 56–63, Cape Town, South Africa, May 2, 2010.

[13] Carl Gutwin, Reagan Penner, and Kevin Schneider. Group awareness in distributed software development. In *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work (CSCW04)*, pages 72–81, Chicago, IL, USA, November 6–10, 2004.

[14] Reid Holmes, Rylan Cottrell, Robert J. Walker, and Jörg Denzinger. The end-to-end use of source code examples: An exploratory study. In *The 25th IEEE International Conference on Software Maintenance (ICSM09)*, pages 555–558, Edmonton, AB, Canada, September 20–26, 2009.

[15] Mik Kersten and Gail C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE06)*, pages 1–11, Portland, Oregon, USA, November 5–11, 2006.

[16] Audris Mockus, Roy T. Fielding, and James Herbsleb. A case study of open source software development: the apache server. In *Proceedings of the 22nd ACM/IEEE International Conference on Software Engineering (ICSE00)*, pages 263–272, Limerick, Ireland, June 4–11, 2000.

[17] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP09)*, pages 87–102, Big Sky, MT, USA, October 12–14, 2009.

[18] David Saff and Michael D. Ernst. Reducing wasted development time via continuous testing. In *Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE03)*, pages 281–292, Denver, CO, USA, November 17–20, 2003.

[19] David Saff and Michael D. Ernst. Continuous testing in Eclipse. In *Proceedings of the 2nd Eclipse Technology Exchange Workshop (eTX04)*, Barcelona, Spain, March 30, 2004.

[20] David Saff and Michael D. Ernst. An experimental evaluation of continuous testing during development. In *Proceedings of the 2004 International Symposium on Software Testing and Analysis (ISSTA04)*, pages 76–85, Boston, MA, USA, July 12–14, 2004.

[21] Anita Sarma. A survey of collaborative tools in software development. Technical Report UCI-ISR-05-3, University of California, Irvine, Institute for Software Research, 2005.

[22] Anita Sarma, Zahra Noroozi, and André van der Hoek. Palantír: raising awareness among configuration management workspaces. In *Proceedings of the 25th ACM/IEEE International Conference on Software Engineering (ICSE03)*, pages 444–454, Portland, OR, USA, May 6–8, 2003.

[23] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st ACM/IEEE International Conference on Software Engineering (ICSE09)*, pages 364–374, Vancouver, BC, Canada, May 20-22, 2009.

[24] Annie T.T. Ying, Gail C. Murphy, Raymond Ng, and Mark C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering (TSE)*, 30:574–586, 2004.

[25] Thomas Zimmermann, Peter Weissgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering (TSE)*, 31(6):429–445, 2005.