Reducing Feedback Delay of Software Development Tools via Continuous Analysis

Kivanç Muşlu, *Member, IEEE*, Yuriy Brun, *Member, IEEE*,
Michael D. Ernst, *Senior Member, IEEE*, and David Notkin, *Fellow, IEEE*

Abstract—During software development, the sooner a developer learns how code changes affect program analysis results, the more helpful that analysis is. Manually invoking an analysis may interrupt the developer's workflow or cause a delay before the developer learns the implications of the change. A better approach is *continuous analysis* tools that always provide up-to-date results. We present Codebase Replication, a technique that eases the implementation of continuous analysis tools by converting an existing offline analysis into an IDE-integrated, continuous tool with two desirable properties: isolation and currency. Codebase Replication creates and keeps in sync a copy of the developer's codebase. The analysis runs on the copy codebase without disturbing the developer and without being disturbed by the developer's changes. We developed Solstice, an open-source, publicly-available Eclipse plug-in that implements Codebase Replication. Solstice has less than 2.5 milliseconds overhead for most common developer actions. We used Solstice to implement four Eclipse-integrated continuous analysis tools based on the offline versions of FindBugs, PMD, data race detection, and unit testing. Each conversion required on average 710 LoC and 20 hours of implementation effort. Case studies indicate that Solstice-based continuous analysis tools are intuitive and easy-to-use.

Index Terms—Continuous analysis, Codebase Replication, Solstice

1 Introduction

When a developer edits source code, the sooner the developer learns the changes' effects on program analyses, the more helpful those analyses are. A delay can lead to wasted effort or confusion [6], [53], [65]. Ideally, the developer would learn the implications of a change as soon as the change is made.

A few analysis tools already provide immediate feedback. IDEs such as Eclipse and Visual Studio continuously compile the code to inform developers about a compilation error as soon as a code change causes one. Continuous testing informs the developer as soon as possible after a change breaks a test [65]. Speculative conflict detection informs the developer of a conflict soon after the developer commits conflicting changes locally [11]. Speculative quick fix informs the developer of the implications of making a compilation-error-fixing change even *before* the change takes place [58].

Unfortunately, most analysis tools are not designed to continuously provide up-to-date results. Instead, a developer may need to initiate the analysis manually.[1] To ease converting these offline analysis tools into continuous analysis tools [16] that run automatically and always provide up-to-date results, we introduce Codebase Replication.

After computing analysis results, a continuous analysis tool may indicate that information is available, display the analysis results, or prompt other relevant analyses or tools to run. Some analyses benefit more than others from continuous execution. For example, a fast continuous analysis provides more frequent and earlier feedback than a slow continuous analysis. Even for long-running analyses, continuous execution could provide results to the developer after an external interruption, such as receiving a phone call. This is sooner than the developer would otherwise learn the results, and, again, without requiring the developer to initiate the analysis.

Our goal is not to make analyses run faster nor incrementally, but to make it easy to run them more frequently and to simplify the developer's workflow—all without requiring a redesign of the analysis tool. Research that makes analyses run faster, including partial and incremental computation, is orthogonal to our work.

Making an analysis continuous is challenging. This explains why few continuous analyses exist, despite their benefits. Two major challenges are *isolation* and *currency*. Isolation requires that (1) the analysis should not prevent the developer from making new changes, and code changes made by an impure (side-effecting) analysis should not alter the code while the developer is working, and (2) developer edits should not make results of an ongoing analysis potentially stale. Currency requires that (1) analysis results are made available as soon as possible, and (2) results that are outdated by new developer edits are identified as stale.

Codebase Replication addresses these challenges by employing four principles:

1) *replication*—keeping a separate, in-sync copy of the developer's codebase on which the analysis executes,

1. Only 17 percent of Eclipse plug-ins listed on GitHub that implement analyses are continuous and reactive to the developer's latest changes; see Section 3.1.

- *K. Muşlu, M.D. Ernst, and D. Notkin are with the Department of Computer Science & Engineering, University of Washington, Seattle, WA 98195. E-mail: {kivanc, mernst, notkin}@cs.washington.edu.*
- *Y. Brun is with the School of Computer Science, University of Massachusetts, Amherst, MA 01003. E-mail: brun@cs.umass.edu.*

Manuscript received 24 Feb. 2014; revised 14 Feb. 2015; accepted 1 Mar. 2015. Date of publication 25 Mar. 2015; date of current version 26 Aug. 2015. Recommended for acceptance by L. Baresi.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2015.2417161

2) *buffer-level synchronization*—ensuring that the analysis has access to the latest developer edits,
3) *exclusive ownership*—allowing analyses to request exclusive write access to the copy of the developer's codebase, and
4) *staleness detection*—identifying results made stale by new developer changes.

The key idea underlying Codebase Replication is to create and maintain an in-sync copy of the developer's codebase. This allows the underlying offline analysis to run on the copy codebase without the developer's changes affecting analysis execution. Further, if the analysis needs to alter the copy codebase, it can do so without affecting the developer's codebase and without distracting the developer. The offline analysis does not need to make a fresh copy of the code before each execution (which would cause significant delay) because the copy codebase is always in sync.

Existing analysis automation techniques fail to achieve simultaneous isolation and memory-change currency. Build tools such as Ant [1] and Maven [2] can be automated to achieve file-change currency, but they do not achieve isolation since the developer and the analysis work on the same codebase. These build tools could create a separate copy of the codebase and achieve isolation in the same way as integration servers, such as Hudson [44] and SonarQube [73], but such an approach cannot achieve memory-change currency: build tools can only access the code saved to disk, whereas integration servers can only access the latest version control commit. Codebase Replication enables isolated memory-change-triggered continuous analyses without increasing implementation complexity.

If the developer edits the code while the analysis is running, Codebase Replication can choose from a variety of reactions: terminate and restart the offline analysis so that the produced results are always accurate; defer propagating the developer's edits to the copy codebase until the analysis is finished so that the analysis can complete, in case the results are useful even when a little stale; or complete the analysis and use analysis-specific logic to mark parts of the results as stale. Since computing analysis results early and often means that more accurate results are available sooner after the developer changes the code, the results could be shown to the developer immediately to reduce wasted time [6], [53], [65], or less frequently to avoid distracting and annoying the developer [7]. Codebase Replication supports both. An up-to-date analysis result might become stale after an edit to the program. In this case, a continuous analysis can either immediately remove the stale results and indicate that the offline analysis is re-executing on the updated program, or mark the results stale, but keep them until the new results are computed, so that the results are not removed from the developer's context. Codebase Replication supports both.

If a UI were poorly designed to interrupt the developer, then the continuous analysis results could be distracting to the developer. However, if the information is presented unobtrusively and the developer is permitted to act on it when he or she chooses, then developers find it useful. This has been confirmed experimentally by Saff and Ernst [66] and is reflected by the popularity of continuous analysis tools such as continuous compilation. Codebase Replication automatically updates the analysis results in a separate GUI element without disturbing the developer. The developer can make this GUI element (in)visible to ignore or have access to the analysis results.

This paper extends our previous work [57] that proposed Codebase Replication. First, by identifying the key continuous analysis design dimensions and the interactions between these dimensions and various offline analysis properties, this paper further simplifies the creation of continuous analyses (Section 3). Second, while the previous work considered offline analyses to be black boxes, this paper improves Codebase Replication's currency guarantee by proposing two designs that allow Codebase Replication to interrupt an ongoing offline analysis (Section 4.3). Third, this paper expands the earlier evaluation of the claim that Codebase Replication eases the implementation of IDE-integrated continuous analyses by converting an offline analysis into a new continuous race detection tool (Section 6.2.3); this conversion required a programmer to write 545 LoC during 24 hours of implementation work. Fourth, this paper evaluates the usability of continuous analysis tools implemented with Solstice in a case study on ten programmers (Section 6.3); this case study shows that programmers like using Codebase-Replication-based continuous analysis tools. Fifth, this paper analyzes the publications and publicly available implementations of existing continuous analyses to further clarify how they lack isolation, currency, or both (Fig. 13 in Section 7).

The main contributions of our work are:

- A discussion of the three major design dimensions of continuous analysis implementations (Section 3).
- A Codebase Replication design that addresses currency and isolation (Section 4), including two alternatives for adding external interruption support to a continuous analysis implementation to increase its input currency without violating the analysis isolation (Section 4.3).
- Solstice, an Eclipse-based realization of Codebase Replication that brings isolation and currency to offline analyses for easily converting them into continuous analysis Eclipse plug-ins (Section 5).
- An evaluation of Solstice's performance, in terms of overhead (isolation cost) and responsiveness to changes (currency) (Section 6.1).
- Four publicly-available, continuous-analysis Eclipse plug-ins with isolation and currency properties, and an evaluation of the ease of building such plug-ins with Solstice (Section 6.2).
- An evaluation of the Solstice continuous testing plug-in in two case studies, demonstrating that Solstice (and therefore Codebase Replication) continuous analysis tools are intuitive and easy to use, and are liked by the programmers (Section 6.3).

2 Definitions

In order to explain Codebase Replication, we first define several concepts, including what it means for an analysis to be continuous.

A *snapshot* is the state of the source code of a software program at a point in time. An *analysis* is a computation on a snapshot that produces a result. An *offline analysis* is an

analysis that requires no developer input. A *continuous analysis* is one that automatically computes an up-to-date result without the need for the developer to trigger it. Finally, a *pure analysis* is one that does not modify the snapshot on which it runs, while an *impure analysis* may. More formally:

Definition 1 (Snapshot). *A* snapshot *is a single developer's view of a program at a point in time, including the current contents of unsaved editor buffers. A unique snapshot is associated to each point in time.*

Each of a developer's changes creates a new snapshot.

In this paper, we limit ourselves to considering analyses that run on a single developer's codebase. Some analyses, such as conflict detection [11], [12], may need multiple developers' codebases for the same program. Our work is applicable to such analyses, although the definition of a snapshot would need to be extended.

Definition 2 (Analysis). *An* analysis *is a function $A : S \rightarrow R$ that maps a snapshot $s \in S$ to a result $r \in R$: $A(s) = r$.*

Definition 3 (Offline analysis). *An* offline analysis *is an analysis that requires no human input during execution.*

For example, a rename refactoring is not an offline analysis because each execution requires specifying a programming element (e.g., a variable) and a new name for this element. An offline analysis may require human input for one-time setup, such as setting configuration parameters or the location of a resource.

Definition 4 (Analysis implementation). *An offline* analysis implementation *A_o for an analysis A is a computer program that, on input snapshot s, produces $r = A(s)$.*

We denote as $T_{A_o}(s)$ the time it takes an analysis implementation A_o to compute $r = A(s)$ on a snapshot s.

It is our goal to convert an offline analysis implementation A_o into a continuous analysis implementation A_c that executes A_o internally.

Definition 5 (ε-continuous analysis implementation). *Let A be an offline analysis, and let t_s be the time at which snapshot s comes into existence. An* analysis implementation *A_c that uses A_o is* ε-continuous *if $\exists\, \varepsilon_a, \varepsilon_s \leq \varepsilon$ such that for all snapshots s, both of the following are true:*

1) *A_c makes $r = A(s)$ available no later than $t_s + T_{A_o}(s) + \varepsilon_a$ if no new snapshot is created before this time. ε_a is the result delay: the time it takes to interrupt an ongoing analysis (A_o) execution, apply any pending edits to the copy codebase, restart the analysis, and deliver the results (e.g., to a UI or a downstream analysis). ε_a is independent of the underlying offline analysis run time since it does not include $T_{A_o}(s)$.*
2) *For all times after $t_{s+1} + \varepsilon_s$, A_c indicates that all results for s are stale. ε_s is the staleness delay: the time it takes to mark the displayed results as stale after the moment they become stale.*

We often refer to ε-continuous analyses as simply *continuous*, implying that an appropriately small ε exists. For simplicity of presentation, our definition of a continuous analysis assumes the most eager policies for handling concurrent developer edits and displaying stale results. Sections 3.2 and 3.3 explore other policies.

It is particularly challenging to convert an *impure* offline analysis to a continuous analysis. Our approach handles both pure and impure analyses.

Definition 6 (Pure/impure analysis implementation). *An analysis implementation A_o is* pure *iff its computation on a snapshot s does not alter s. An* impure *analysis implementation may alter s.*

Running a test suite is an example of a pure analysis because it does not alter the source code. Mutation analysis—applying a mutation operation to the source code and running tests on this mutant—is an impure analysis. An analysis that performs source code instrumentation is a special case of an impure analysis. An analysis that performs run-time instrumentation of a loaded binary is pure. Note that an impure analysis only alters the source code temporarily, while computing the results. Once the results are computed, the impure analysis or Codebase Replication must revert the source code to its initial state.

3 Key Design Dimensions for a Continuous Analysis Tool

Our approach to implementing a continuous analysis involves executing an offline analysis internally. This section discusses three key design dimensions for continuous analysis tools that are converted from their offline analyses. Section 3.1 investigates what can trigger a continuous analysis tool to internally run the offline analysis, Section 3.2 investigates when a continuous analysis tool can abort an offline analysis execution, and Section 3.3 investigates how stale results are displayed to the developer. For each dimension, we discuss the advantages and disadvantages and provide examples from existing analysis implementations.

3.1 Triggering Analysis Execution

A continuous analysis may use four categories of triggers to start internal offline analysis executions: (1) whenever the snapshot changes in memory, (2) whenever the snapshot changes on disk, (3) periodically, and (4) other triggers. Additionally, analyses from each of the categories may be *delayed* and/or *overlapping*.

Memory-change triggers. The continuous analysis runs the internal offline analysis each time the program snapshot changes in memory, such as when the developer makes an edit in the IDE. A memory-change-triggered analysis provides feedback without requiring the developer to save the file. The Solstice Continuous Testing plug-in (described in Section 6.2.4) and Eclipse reconciler compiler[2] are examples of this category of analysis.

File-change triggers. The continuous analysis runs the internal offline analysis each time the program snapshot changes in the file system. A file-change-triggered analysis is motivated by the hypothesis that changes a developer saves to disk are more likely to be permanent than those merely made

2. Eclipse contains two different compilers. The reconciler compiler operates on unsaved buffer content in order to give quick feedback; Eclipse calls it the "Java reconciler". The incremental compiler operates on saved files and gives more complete and correct feedback about compiler errors whenever the user saves the document; Eclipse calls it the "incremental Java builder".

in memory. Finally, as file-system changes are less frequent than memory changes, these triggers can result in less resource use. However, waiting for changes to be saved to disk can delay computing pertinent analysis information. Eclipse provides a continuous analysis framework, called Incremental Project Builders [23]. Incremental Project Builders broadcasts the difference between two incremental builds on the file system, so that other analyses can access this difference and incrementally update results. Any continuous analysis Eclipse plug-in built using Incremental Project Builders, such as the FindBugs [35], Checkstyle [24], and Metrics [28] plug-ins, is a file-change-triggered continuous analysis.

Periodic and non-stop. The continuous analysis runs the offline analysis with a regular period. For example, the Crystal tool [10], [12] executes its analysis every 10 minutes. A variant of periodic analysis is a *non-stop* analysis, which runs every time the previous execution finishes. A periodic analysis is most suitable when it is difficult to determine which actions may affect the analysis result.

Other triggers. Quality-control analyses often run before or after each version control commit. Pre-commit analyses prevent developers from committing code that breaks the build or test suite. These quality-control analyses must be fast since they would otherwise discourage the developer from making frequent commits. Building components and running unit tests that are directly affected by the changes are examples of such quality-control analyses. Post-commit analyses such as continuous integration notify the developers soon after a bad commit. Continuous-integration analyses can be slower since they run separately from the development, typically on a dedicated integration server. Building the software completely and running all integration tests are examples of continuous-integration analyses.

Analyses from each of the above categories may be *delayed* after the trigger before running the offline analysis. Eclipse's reconciler compiler is delayed until the developer pauses typing to avoid running the analysis during a burst of developer edits. The delay avoids executing the analysis on intermediate snapshots for which the results are less likely to be of interest to developers and are likely to become stale quickly: the delay thus also reduces analysis overhead. Delays are most appropriate for a memory-change-triggered analysis. Although file-change- and other-triggered analyses rarely use delays since actions such as saving a file or committing code already suggest good opportunities to run the offline analysis, these analyses might introduce delays for taking common developer patterns into consideration. For example, developers commit in bursts and a delay may avoid running the analysis on a snapshot that is about to be overridden by a new commit. Jenkins [48] supports a "quite period" in which the builds are delayed after a commit to prevent an incomplete commit trigger a build failure.

Additionally, analyses from each of the above categories may be *overlapping*. Whenever a trigger fires while a previous offline analysis is still running, the continuous analysis has to decide whether to start a second, concurrent copy of the offline analysis. If the analyses are non-overlapping, the new offline analysis execution can be skipped, delayed until the current execution finishes, or started instead of finishing the previous execution (Section 3.2).

Surveying triggering in existing analyses. To determine how existing analysis tools, we surveyed Eclipse plug-ins on GitHub. We performed manual inspection of the documentation, automated analysis of the source code to find design patterns that indicate a continuous analysis, and manual inspection of the source code when necessary.

Of the 159 projects that match "Eclipse plug-in", 47 implemented analysis tools. Of those 47, 21 (45 percent) were continuous: one (2 percent) was triggered by VCS commands, 12 (26 percent) were file-change-triggered and only eight (17 percent) were memory-change-triggered. This suggests that despite the significant benefits of memory-change-triggered continuous analyses, they are difficult to implement, which motivates and justifies our work.

Of the 21 continuous analyses, 16 (8 file-change-triggered and 8 memory-change-triggered) extended Eclipse to handle languages other than Java, made possible in part by the relative simplicity of using the Incremental Project Builder [23], Xtext [80], and Reconciler [25] patterns. Similarly, we anticipate that our work will ease the creation and increase the number of memory-change-triggered continuous analysis tools for arbitrary analyses.

3.2 Abandoning the Ongoing Offline Analysis

When the developer edits the program while a continuous analysis is running the offline analysis, the continuous analysis can either (1) immediately interrupt the offline analysis execution without getting any results, potentially rerunning it on the latest snapshot, or (2) never interrupt the offline analysis, and finish running it on the snapshot, which is no longer the up-to-date development snapshot.

Immediately interrupt. The continuous analysis abandons the ongoing offline analysis immediately when a developer makes an edit. Such a continuous analysis is most suitable when the results of the analysis on an outdated snapshot have little or no value. An immediately-interrupting continuous analysis wastes no time executing on outdated snapshots. Quick Fix Scout [58] is an example of an immediately-interrupting continuous analysis.

Section 4.3 presents two designs for implementing immediately-interrupting continuous analyses by forcibly terminating the ongoing offline analysis.

Never interrupt. The continuous analysis continues to execute the offline analysis despite concurrent developer edits. To ignore the developer edits, the never-interrupting continuous analysis runs on a copy of a recent snapshot. An example is an analysis that runs after commits or nightly builds. A never-interrupting analysis is most suitable when the offline analysis takes a long time to run and when the results on a slightly outdated snapshot still has value to the developer. For example, a continuous integration server may complete running tests, while allowing the developers to continue editing the program and make new commits. The test results are useful for localizing bugs. The alternative of interrupting the test suite execution every time the developer makes a change could mean the test suite rarely finishes and the developer rarely sees any analysis results.

Codebase Replication enables never-interrupting continuous analyses by providing an isolated copy of the snapshot. More sophisticated interruption policies are possible. For example, when a conflicting developer edit takes place, an offline analysis could be permitted to complete its execution if that execution is estimated to be at least 50 percent done.

3.3 Handling Stale Results

A continuous analysis that does not interrupt its offline analysis may generate stale results. Furthermore, as the developer edits the code, displayed results may become stale.

We group continuous analyses by how they handle stale results into two categories: (1) immediately remove stale results, and (2) wait to remove stale results until new results are available. Analyses in either category can use cues to indicate that results are potentially stale and/or a new analysis is being run. It would also be possible to delay removing the analysis results or to create a separate (perhaps fast) analysis to check if stale results no longer apply, and remove them based on that analysis.

Immediately remove. The continuous analysis immediately removes the stale results and, potentially, indicates that the offline analysis is being rerun. This approach is appropriate if displaying stale results may lead to developer confusion. For example, showing a compilation error for code that the developer has already fixed may cause the developer to waste time re-examining the code. Examples include Quick Fix Scout [58] and most Eclipse analyses based on Incremental Project Builders, such as the Eclipse FindBugs and Checkstyle plug-ins. A never-interrupting, immediately-removing analysis may be a poor choice because if the developer edits the code while the analysis is running, the continuous analysis completes the offline analysis but never shows its results to the developer.

Display stale. The continuous analysis waits to remove the stale results until new results are available. Removing the old results may hinder a developer using them. For example, when fixing multiple compilation errors, the first keystroke makes all the results stale, but nonetheless the developer wants to see the error while fixing it and may want to move on to another error while the code is being recompiled. If the developer edits one part of the code, then all the analysis results technically become stale, but the developer may know that analysis about unaffected parts of the code remain correct. The Crystal tool [10], [12] is an example of a display-stale analysis, because it visually identifies results as potentially stale.

Unless otherwise noted, when we describe an ε-continuous analysis implementation, we mean a memory-change-triggered, non-overlapping, immediately-interrupting, immediately-removing analysis.

4 Codebase Replication

This section describes our Codebase Replication design, and how it addresses the challenges of isolation and currency.

4.1 Codebase Replication Architecture

Codebase Replication converts an offline analysis A_o into a continuous analysis A_c while addressing the two major challenges to creating continuous analysis tools: isolation and currency.

Isolation ensures that the developer's code changes and the execution of the offline analysis happen simultaneously without affecting each other. The developer should be isolated from A_c: A_c should neither block the developer nor change the code as the developer is editing it (even though an impure A_o may need to change the code). Additionally, A_c should be isolated from the developer: developer edits should not alter the snapshot in the middle of an A_o execution, potentially affecting the results.

Despite isolation between the developer and A_c, *currency* requires A_c to react quickly to developer edits and to A_o results. Whenever the developer makes an edit, A_c should be notified so that it can mark old results as stale, terminate and restart A_o, or take other actions. A_c should react to fine-grained changes in the developer's editor's buffer, without waiting until the developer saves the changes to the file system nor commits them to a repository. A_c should also react quickly to A_o results, making them promptly but unobtrusively available to the developer or to downstream analyses.

Eclipse provides the Jobs API [29], which allows the UI thread to spawn an asynchronous task, execute it in the background, and eventually join back to the UI thread. Eclipse's incremental compiler and continuous analysis plug-ins implemented on Incremental Project Builders use the Jobs API. Codebase Replication does not follow this approach because the Jobs API does not support the goals of isolation and currency. The Jobs API does not maintain a separate copy of the program, so any code changes by an impure analysis would interfere with the developer, thus failing to provide isolation. The Jobs API does not have direct access to an editor-buffer-level representation of the program, so to provide currency, the continuous analysis would have to combine active editor buffers with the file representation of the program.

Codebase Replication addresses the isolation challenge by creating and maintaining an in-sync copy of the developer's codebase. Codebase Replication addresses the currency challenge by providing notifications for events that occur in the developer's IDE and in A_c; these events can trigger terminating and restarting A_o and updating the UI.

Fig. 1 shows the architecture of Codebase Replication. The IDE API generates events for all developer actions, including changes to the code. Codebase Replication keeps a queue of these events, and applies them to the copy codebase. Meanwhile, A_c can pause the queue, run A_o on the copy snapshot, collect A_o results, resume the queue, and update the results shown to the developer. Codebase Replication also notifies A_c about new developer actions, so that A_c may decide to interrupt, alter, or continue A_o's execution.

Codebase Replication supports multiple A_c using the same copy codebase, via the readers-writers lock protocol. Codebase Replication runs one impure A_c or multiple pure A_c in parallel. For example, a developer might run continuous testing and FindBugs in parallel to obtain both dynamic information—test results—and static information—FindBugs warnings. Running multiple pure A_c in parallel amortizes the already-low overhead (see Section 6.1). Codebase Replication guarantees that, even with multiple A_c, at any given time, the copy codebase can be modified by at most one analysis, and the copy codebase never changes during a pure analysis execution.

For exposition purposes, this paper introduces the Codebase Replication design with one copy codebase. A Codebase Replication implementation can maintain multiple copy codebases and run multiple impure A_c in parallel by running each impure A_c on a separate copy codebase.

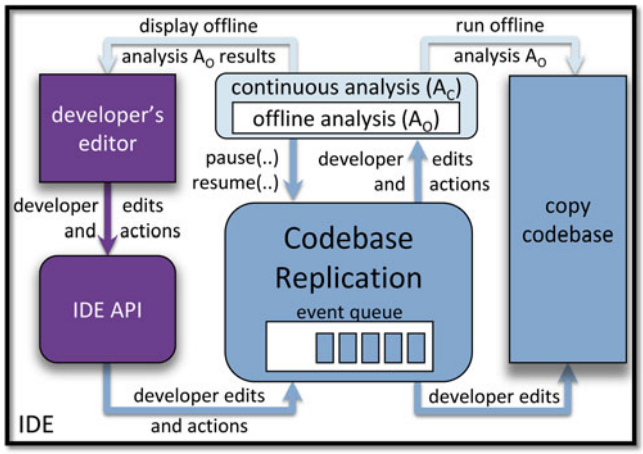


Fig. 1. Codebase Replication architecture. Codebase Replication (blue) facilitates communication between A_c (light blue) and a developer's IDE (dark purple) via asynchronous events.

4.2 Ensuring Isolation and Currency

Codebase Replication employs four principles to overcome the challenges of isolation and currency: replication, buffer-level synchronization, exclusive ownership, and staleness detection.

Replication. Codebase Replication runs A_o on the copy codebase to ensure that the developer's codebase is never affected by A_o. Replication is unidirectional: Codebase Replication copies the developer's changes on the developer's codebase to the copy codebase, not the other way around. Similar to sandboxing, if A_o modifies the copy codebase or crashes in the middle of the execution, the effects are confined to the copy, and the developer may continue to edit unaffected by these modifications and crashes. Further, this approach separates the UI logic from the analysis logic, which is a good design principle that improves maintainability.

Buffer-level synchronization. A_c aims to provide feedback on the developer's view of the code, which often resides in the IDE buffer rather than on disk. However, most existing A_o execute on files or even on binaries. Codebase Replication synchronizes the on-disk copy snapshot with the IDE's unsaved buffers.

Exclusive ownership. Changing the snapshot in the middle of an execution may cause A_o to produce incorrect results or to crash. The situation also arises if multiple A_o run on the same code at once, and at least one of them is impure. Codebase Replication allows one impure A_c at a time to claim exclusive access to a copy snapshot while its A_o executes, excluding all other analyses and pausing synchronization updates with the developer's buffer.

Staleness detection. A_o results from an old snapshot might become stale as a result of developer's changes. If a change occurs while A_o executes, the result may already become stale by the time A_o completes. When a change occurs, Codebase Replication notifies A_c and allows it to choose to finish executing or to terminate A_o.

Section 6 will revisit how well our design and implementation satisfy these requirements.

4.3 Improving Currency with Analysis Termination

To improve analysis input currency, when there is an edit that conflicts with an ongoing analysis execution, Codebase Replication has the ability to terminate the ongoing analysis execution, apply the conflicting edits to the copy codebase, and rerun the analysis on the updated codebase. Proper termination of an ongoing analysis requires additional support from either A_o or A_c. This section discusses two designs: one in which the support is provided by A_o and one in which the support is provided by A_c.

A_o-provided termination support. The first way to provide external support for termination is to require external interruption support from the offline analysis A_o. When Codebase Replication interrupts an ongoing analysis execution, A_o is expected to abandon its execution and do any cleanup that is needed in a timely manner, such as reverting modifications to the source code, databases, file pointers, and class loaders. If A_o provides external interruption support, A_c requires no changes. It just interrupts A_o when a conflicting developer edit is detected.

A_c-provided termination support. The second way to provide external support for termination is to design A_c to never interrupt the offline analysis A_o, instead executing A_o multiple times on different chunks of the codebase (e.g., one execution per file). A_c then needs to compose the individual results into a single analysis result for the whole program. Each time A_o finishes executing on a chunk, A_c checks for interrupts. If there is an interrupt, A_c abandons computing the result for the whole program, cleans up, and returns ownership to Codebase Replication. This approach does not require any modifications to A_o. However, it is only applicable when A_o is modular (can be split up to work on program chunks) and executing it on individual chunks is much faster than on the whole program.

To enable A_c-provided termination support, Codebase Replication supports a *step-based execution model* that handles interruption and termination, making it easy to write an interruptible A_c for a modular A_o. The continuous analysis A_c creates a sequence of *steps*, each one an atomic unit of work that executes in a reasonable amount of time, such as several seconds. There are two kinds of steps: (1) RUN steps that represent normal analysis execution, and (2) CLEANUP steps that clean up side effects. Codebase Replication maintains a worklist of steps, executes them in order, and checks for interruptions after each step execution. When it detects an interruption, such as a developer edit, Codebase Replication ignores the remaining RUN steps and executes the remaining CLEANUP steps.

Consider a continuous testing analysis tool (Section 6.2.4) that runs all tests in the project and displays the results to the developer. The tool would not be responsive to the developer's changes if it finishes executing the entire test suite before checking for new developer changes. The step-based execution model improves the tool's responsiveness by checking for changes more often, for example, after each class's tests finish.

For a modular offline analysis, executing it on a chunk of code corresponds to adding one RUN step to the worklist. For example, continuous testing adds the following steps to the worklist: one RUN step that creates a new class loader and identifies the test classes that JUnit can run (concrete classes with at least one test), one RUN step per test class that runs JUnit on that class, and one CLEANUP step, which releases the resources used by the new class loader. For the

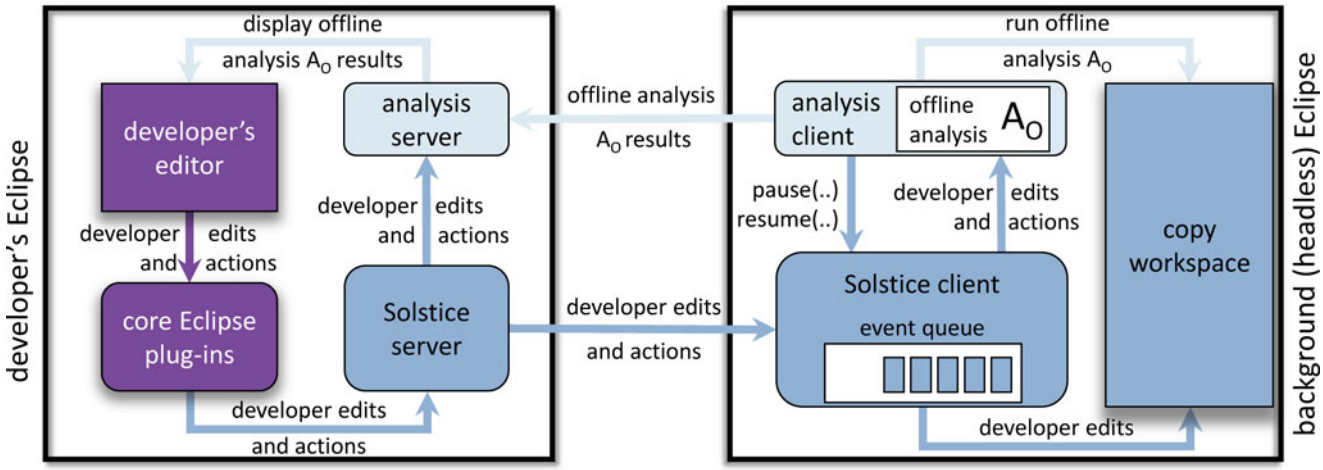


Fig. 2. Solstice architecture as an instantiation of the Codebase Replication architecture (Fig. 1) for Eclipse. Solstice observes the workspace in the developer's Eclipse and creates a new Eclipse process to manage the copy workspace. Solstice and the continuous analysis (A_c) each consist of two components, a server (`Solstice server` for Solstice and `analysis server` for A_c) that interacts with the developer's Eclipse, and a client (`Solstice client` for Solstice and `analysis client` for A_c) that interacts with the copy Eclipse. The developer's Eclipse (dark purple) generates events for developer actions, including edits. `Solstice server` (blue) sends these events to `Solstice client` (blue) and notifies A_c (light blue) of the actions. `Solstice client` stores these actions temporarily in the event queue, applies the edits to the copy workspace, notifies A_c of these actions, and provides a pause-resume API for managing exclusive ownership of the copy workspace. A_c (light blue) interacts with the developer's editor, Solstice, and the copy Eclipse. `Analysis client` runs A_o on the copy workspace and sends the results to `analysis server`. `Analysis server` modifies the developer's editor accordingly and implements staleness logic.

above-mentioned continuous testing implementation, Codebase Replication checks for conflicting edits after each step. If there is an interruption, Codebase Replication ignores remaining RUN steps, but executes the CLEANUP step, which ensures that the new class loader does not leak memory.

Assuming that executing each step is bounded by τ time, the step-based analysis execution approach guarantees that the offline analysis execution can be terminated safely in $(c+1)\cdot\tau$ time, where c is the number of CLEANUP steps added to the worklist before the analysis is interrupted. We anticipate that for most analyses, c will be small.

Our Eclipse-based Codebase Replication prototype Solstice, which we describe next in Section 5, supports the step-based execution model, and all the Solstice-based plug-in analyses from Section 6.2 use it. Support for the step-based execution model required, on average, only an extra 100 LoC.

5 SOLSTICE: CODEBASE REPLICATION FOR THE ECLIPSE IDE

To evaluate Codebase Replication, we built Solstice, an Eclipse-based, open-source Codebase Replication prototype. Solstice is publicly available at https://bitbucket.org/kivancmuslu/solstice/. This section describes Solstice (Section 5.1), explains how to implement continuous analysis tools using Solstice (Section 5.2), and describes one such implementation (Section 5.3). Later, Section 6.2 describes our experience using Solstice to develop four continuous analysis tools.

To the best of our knowledge; Solstice is the first framework that aids implementing memory-change-triggered analysis tools for arbitrary source and binary code analyses, which would otherwise be considerably more difficult to build.

5.1 Solstice Implementation

This section explains the Solstice implementation and refines Codebase Replication with Eclipse-specific concerns.

Fig. 2 illustrates Solstice's architecture. Solstice consists of two parts. `Solstice server` runs on the developer's Eclipse and is responsible for listening to the developer's actions. `Solstice client` runs on a background Eclipse (which we describe next) and is responsible for keeping the copy codebase in sync and managing the ownership of the copy codebase. Solstice-based continuous analysis tools use `Solstice client` for their computation logic and `Solstice server` for their visualization logic and to interact with the developer.

The Eclipse API allows each Eclipse process to be associated with (and have access to) only one workspace. Solstice interacts with two Eclipse processes running at once: the developer's normal Eclipse, which manages the developer's workspace, and a second, background Eclipse, which runs `Solstice client` and maintains the copy workspace (and with it, the copy codebase). The background Eclipse is headless—it has no UI elements and the developer never sees it. The copy workspace resides in a hidden folder on disk. The Solstice implementation maintains one copy codebase. It runs all pure A_c in parallel, and each impure A_c in isolation.

Each time the developer starts Eclipse, Solstice executes an *initialization synchronization protocol* that briefly blocks the developer and ensures that the copy workspace is in sync with the developer's workspace. The first time the developer uses Solstice, the initialization synchronization protocol acts as a full synchronization and creates a complete copy of the developer's workspace on disk. Future executions verify the integrity of the files in the copy workspace through checksum and update the files that were added, removed, or changed in the developer's workspace outside of the IDE.

After the initialization synchronization protocol, `Solstice server` attaches listeners to the developer's Eclipse. The listeners track edits to the source code, changes to the current cursor location, changes to the currently selected file, changes to the currently selected Eclipse

project, invocations of Quick Fix, proposals offered for a Quick Fix invocation, and selections, completions, and cancellations of Quick Fix proposals. Developer actions that alter the code generate both a developer action event (e.g., to say that the developer clicked on a menu item) and an edit event that encodes the code changes. `Solstice server` sends the developer's events to the `Solstice client`, which makes the incoming events available to the continuous analysis tool through the observer pattern and applies all edits on the developer's workspace to the copy workspace.

5.2 Building Solstice-Based Tools

This section explains how to use Solstice to build a continuous analysis tool A_c based on an offline analysis implementation A_o. We refer to the *author* as the person developing A_c, and to the *developer* as the person later using A_c.

To implement A_c, the author specifies: (1) how A_c computes the results, (2) how A_c interacts with the developer, (3) the information that needs to be communicated between the server and the client components of A_c, and (4) how A_c handles stale results.

1) The author specifies A_c computation logic—how A_o runs and produces results. The computation logic is implemented as an Eclipse plug-in that interacts with `Solstice client`, represented as `analysis client` in Fig. 2. The computation logic always runs on the background Eclipse using the contents of the copy workspace.

Most analyses must verify some pre-conditions before running on a codebase. Solstice API contains analysis steps that simplify this verification process for common pre-conditions. For example, the author can use `ProjectCompilesStep` to ensure that the codebase has no compilation errors or `ResourceExistsStep` to ensure that a particular resource (e.g., test folder) exists. If a step's pre-conditions fails, Solstice abandons the analysis and shows a descriptive warning message to the developer.

As an additional contingency mechanism for infinite loops due to bugs in the analysis or dynamic execution of unknown code, Solstice lets the author specify a *timeout* for each step. If a step takes longer than its timeout, Solstice assumes that the analysis went into an infinite loop and terminates the analysis, including the remaining steps.

2) The author specifies the interaction logic—how A_c shows results and interacts with the developer. The interaction logic is implemented as an Eclipse plug-in that interacts with `Solstice server`, represented as `analysis server` in Fig. 2. The interaction logic runs on the developer's Eclipse using the developer's editor.

The same way Eclipse manages the life-cycle of its plug-ins, Solstice manages the life-cycle of A_c: each A_c starts after Solstice starts (when the developer opens Eclipse) and terminates before Solstice terminates (when the developer closes Eclipse). The author does not need to create and manage a thread for A_c, as Solstice takes care of these details.

For the rest of the section, we assume that A_c interacts with the developer. Continuous analysis tools that do not interact with the developer (e.g., an observational A_c that only logs developer actions) do not need an `analysis client` component: `Solstice client` duplicates all developer edits and A_c (`analysis client`) can access those events directly from `Solstice client` via listeners.

3) The author specifies the communication between `analysis client` and `analysis server`. The analysis results generated by `analysis client` need to be sent to `analysis server` to be displayed to the developer, as shown in Fig. 2. The communication does not have to be one-directional (although the example communication shown in Fig. 2 is). For example, the `analysis server` can allow the developer to modify A_c settings, which it would then send to the `analysis client`.

The Solstice API trivializes the inter-process communication between the `analysis client` and the `analysis server`. The author can invoke `sendMessage(...)` to communicate a `Serializable` Java object from the `analysis client` to the `analysis server` or vice versa. Solstice takes care of all low-level networking details, deserializes the object on the receiver, and executes a method that processes the object.

4) The author writes the logic for handling potentially-stale A_o results. Solstice timestamps every developer action and edit, A_o start, and A_o finish, to ensure that no event is lost and that Solstice knows to which snapshot an A_o result applies. Solstice supports all the policies discussed for abandoning A_o (Section 3.2) and handling stale results (Section 3.3). Solstice provides APIs for common scenarios, such as removing A_o results with each developer edit. To specify the staleness behavior of A_c, the author needs to set the value of one `boolean` argument. Solstice takes care of all low-level details, such as attaching multiple listeners to Eclipse to detect resource changes and updating the analysis visualization while handling potentially stale results.

5.3 An Example Solstice Continuous Analysis Plug-In

Suppose an author wants to use Solstice to build an A_c using an A_o. The author decides that A_c will be never-interrupting (Section 3.2) and display-stale (Section 3.3): when the developer makes a change while A_o executes, A_o might as well finish, and A_c will display potentially stale results to the developer, with an indicator.

The author would have to write the following interaction logic for A_c (`analysis server`):

```
class Server extends AnalysisServer {
  public Server() {
    super(true /*Display potentially stale
        results with a special indicator*/);
  }
  // Implement one of the following two methods:
  /** Convert results to human-readable text that
    * will be displayed on the analysis view. */
  String resultToText(Result result) {...}
  /** Convert results to Eclipse markers that
    * will be displayed on the analysis marker
    * view and in the source code. */
  IMarker[] resultToMarkers(Result result) {...}
}
```

`Server` passes `true` to the `Analysis Server` constructor, which makes Solstice display potentially stale results with a special indicator. Solstice calls `resultToText(...)` and `resultToMarkers(...)` with A_o results (`Result`) received from the `analysis client`. The author needs to implement at least one of these methods to transform `Result` into a human-readable text or Eclipse markers, which Solstice uses to automatically update the contents of the corresponding Eclipse view.

The author would also have to write the following computation logic for A_c (`analysis client`):

```
class Analysis extends ResourceBasedAnalysis {
  List<Step> getSteps() {
    List<Step> steps = new ArrayList<>();
    steps.add(new RunStep() {
      void run() {
        Result result = runOfflineAnalysis();
        generateResult(result);
    }});
    return steps;
  }
}
```

`Analysis` extends `ResourceBasedAnalysis`, which makes Solstice rerun A_o each time Solstice applies all developer edits to the copy workspace and the copy workspace is up to date. Under the step-based semantics, Solstice executes the analysis steps returned from `getSteps()`. Each step can invoke `generateResult(Result)`, which makes Solstice automatically send this `Result` to the `analysis server`.

6 Evaluation

To evaluate Solstice, we empirically measured its performance overhead (Section 6.1), determined the ease of using Solstice by implementing four proof-of-concept continuous analysis tools (Section 6.2), observed developers' interaction with continuous analysis tools in two case studies (Section 6.3), and compared Solstice to other methods of implementing IDE-integrated continuous analyses (Section 6.4).

6.1 Solstice Performance Evaluation

An effective continuous analysis should meet the following requirements:

Low initialization overhead. The developer should not be blocked too long during startup (Section 6.1.1).
Low synchronization overhead. While using the IDE, the developer should experience negligible overhead (Section 6.1.2).
High analysis input currency. The delay after an edit before an analysis can access an up-to-date program in the copy codebase should be small (Section 6.1.3).

This section presents the results of performance experiments addressing these three requirements. The experiments were executed on a MacBook Pro laptop (Mac OS X 10.9, i7 2.3 GHz quad core, 16 GB RAM, SSD hard drive).

Program	Workspace Size (MB)	Code Size (KLoC)	Sync Time (seconds)			
			Empty Copy		In-sync Copy	
			μ	σ	μ	σ
CrosswordSage [18]	17	4	0.3	0.2	0.1	0.1
ASMX [4]	26	43	1.8	0.4	0.5	0.1
Voldemort [77]	55	160	6.4	0.8	1.3	0.2
JDT [27]	164	1,694	124.0	10.8	8.6	1.1

Fig. 3. Solstice initial synchronization protocol performance. Each cell is the mean of 20 executions.

Solstice ran with a 512 MB RAM limitation for each of the server and client components.

6.1.1 Initial Synchronization Protocol Cost

Every time the developer runs Eclipse, Solstice executes a blocking initial synchronization protocol (recall Section 5) to ensure that the copy workspace is in sync with the developer's workspace. This is required because Solstice does not track changes to the developer's workspace when Eclipse is not running.

We have tested Solstice's initial synchronization protocol using four different workspace contents (Fig. 3). For each setting, we created a workspace with one program and invoked the initial synchronization protocol for two extreme cases: full synchronization and no synchronization. In the full case, the copy workspace is empty, which requires Solstice to copy the entire workspace. In the none case, the copy workspace is already in sync, which requires Solstice to only verify that the copies are in sync using checksums. Since developers make most of their code changes within an IDE, we expect most invocations of Solstice after the first one to resemble the none case.

Fig. 3 shows that Solstice has low synchronization overhead. This could be further reduced by a lazy initial synchronization protocol that only processes the active Eclipse project and its dependencies, not all projects in the program (for example, JDT consists of 29 Eclipse projects from eclipse.jdt, eclipse.jdt.core, eclipse.jdt.debug, and eclipe.jdt.ui).

Solstice would have been easier to implement if it always built a brand new copy of the workspace on Eclipse startup. There would be two main sources of overhead:

1) Copying the files. For the JDT workspace, containing 16,408 files, this takes 30 seconds ($\sigma = 1.1$ sec.).
2) Creating an Eclipse project and importing it into the workspace. Eclipse creates metadata for the project and indexes project files. For the JDT workspace, this takes 74 seconds ($\sigma = 2.5$ sec.).

This is 12 times slower than Solstice's incremental synchronization, which takes only 8.6 seconds for the JDT workspace.

6.1.2 IDE Synchronization Overhead

Solstice tracks all developer changes at the editor buffer level. The "IDE overhead" column of Fig. 4 shows, for the most common developer actions, the IDE overhead that Solstice introduces when the action is initiated programmatically. The overhead is independent of the edit size and is no more than 2.5 milliseconds.

Even adding or removing 1,000 files incurs modest overhead that is similar to the 1.085 seconds that Eclipse takes to import similar-sized project (org.eclipse.jdt.core, 1,205 files).

Operation Name	Size	Initial File Size (chars)	IDE Overhead (s)	Sync Delay (s)
Text Insert	1	1	0.001	0.002
		100	0.001	0.002
		1,000	0.001	0.002
		10,000	0.002	0.002
	100	1	0.001	0.002
		100	0.001	0.002
		1,000	0.001	0.002
		10,000	0.002	0.003
Text Delete	1	1	0.001	0.002
		100	0.001	0.002
		1,000	0.001	0.002
		10,000	0.003	0.002
	100	1	0.001	0.002
		100	0.001	0.002
		1,000	0.001	0.002
		10,000	0.002	0.002
Text Edit Summary			≤ 0.003	≤ 0.003
File Add	1	1,000	0.001	0.001
	100		0.102	0.157
	1,000		1.464	1.305
File Remove	1	1,000	0.001	0.001
	100		0.056	0.106
	1,000		0.566	2.491
File Edit Summary			grows linearly with size	

Fig. 4. The Solstice-induced overhead on developer edits for keeping the copy workspace in sync. Text operations of size 1 are single keystrokes, and larger text operations add, replace, or remove 100 consecutive characters at once to represent cut, paste, and tool applications, such as applying a refactoring or an auto-complete. File operations of size 1 are manual file generation, copy, and removal, and larger file operations represent copying, removing, or importing a directory or an entire Eclipse project. "IDE Overhead" measures the overhead imposed on the responsiveness of the IDE, and "Sync Delay" measures the delay before the copy workspace is up to date. For each text operation experiment, we executed the operation 100 times and took the average to reduce external bias, such as JVM warmup.

6.1.3 Copy Codebase Synchronization Delay

To allow A_c to access the up-to-date version of the developer's code, Solstice must quickly synchronize the copy workspace. The "Sync delay" column of Fig. 4 shows the delay Solstice incurs during synchronization for the most common developer operations. Synchronizing text edits takes no more than 2.5 milliseconds. Thus, Solstice provides A_c access to the developer's code that is no more than 2.5 milliseconds old. Importing and deleting a 1,000-file project takes longer, up to 2.5 seconds, but since these operations are rare and already take several seconds for Eclipse to execute, the Solstice delay should be acceptable.

6.1.4 Summary

Our performance analysis demonstrates that Solstice introduces negligible overhead to the IDE, does not interrupt the development process (except during startup) and provides access to an up-to-date copy codebase with negligible delay.

6.2 Solstice Usability Evaluation

We have used Solstice to build four continuous analysis Eclipse plug-ins, each using an existing offline analysis implementation. This section describes these implementations and reports on the building experience.

Fig. 5 summarizes the continuous analysis tools built on Solstice. The epsilon values in Fig. 5 are computed by instrumenting Solstice to timestamp the moments when:

t_e: the developer makes an edit,
t_{as}: Solstice starts running A_o,
t_{ae}: Solstice finishes running A_o,
t_d: Solstice displays analysis results to the developer,
$t_{e'}$: the developer makes a new edit,
t_s: Solstice marks analysis results as stale (after the new edit)

After an analysis computed its initial results, we made 10 small edits in Eclipse (ranging a few lines to adding/removing one method) that produce different analysis results. For each analysis execution, we computed $\varepsilon_i = t_s - t_{e'}$ and $\varepsilon_a = (t_{as} - t_e) + (t_d - t_{ae})$ and Fig. 5 displays their maximum.

This section presents each continuous analysis as a separate Eclipse plug-in. Solstice supports running multiple pure A_c in parallel.

6.2.1 Continuous FindBugs

FindBugs is a static analysis tool that finds common developer mistakes and bad practices in Java code, such as incorrect bitwise operator handling and incorrect casts. FindBugs has found bugs in open-source software, is useful to developers, and is extensible with new bug patterns [43]. It is available as a command-line and a GUI tool, an Ant task extension, and an Eclipse plug-in [35].

The FindBugs Ant task extension and Eclipse plug-in can automate FindBugs invocations, but both fall short of being

Analysis	Lines of Code					Dev.	Evaluation Subject Program				A_o Run	ε_a	ε_i
	UI	IPC	Core	Other	Total	Time (h)	Name	Version	KLoC	KNCSL	Time (s)	(s)	(s)
FindBugs	240	18	239	22	519	25	Voldemort	1.6.4	160	115	74	0.033	0.002
PMD	265	18	224	22	529	4					25	0.102	0.004
Check Sync.	160	39	324	22	545	24	Java Grande	1.0	10.5	4.8	293	0.073	0.004
Testing	581	184	458	22	1245	25	Commons CLI	1.3	10.5	5.8	0.01	0.108	0.003

Fig. 5. Summary of four Solstice-based continuous analysis tools. Each tool consists of "UI" code for basic configuration and result visualization, "IPC" code for serialization, "Core" code for setting up and running the offline analysis A_o, and "Other" code for extension points. Code sizes are larger than reported in [57] because of new UI functionality and support for the step-based execution model. The continuous PMD analysis took little development time due to similarities to FindBugs, which was developed before it. Each tool is ε-continuous and the table reports the maximum observed ε values. The experiments used PMD's `java-basic` ruleset and all of Commons CLI's 361 tests. The Check Synchronization evaluation considers 11 (out of 31) of the Java Grande benchmark programs (main classes). 14 programs could not be executed by Check Synchronization and six programs took longer than 5 minutes and were excluded due to time considerations.

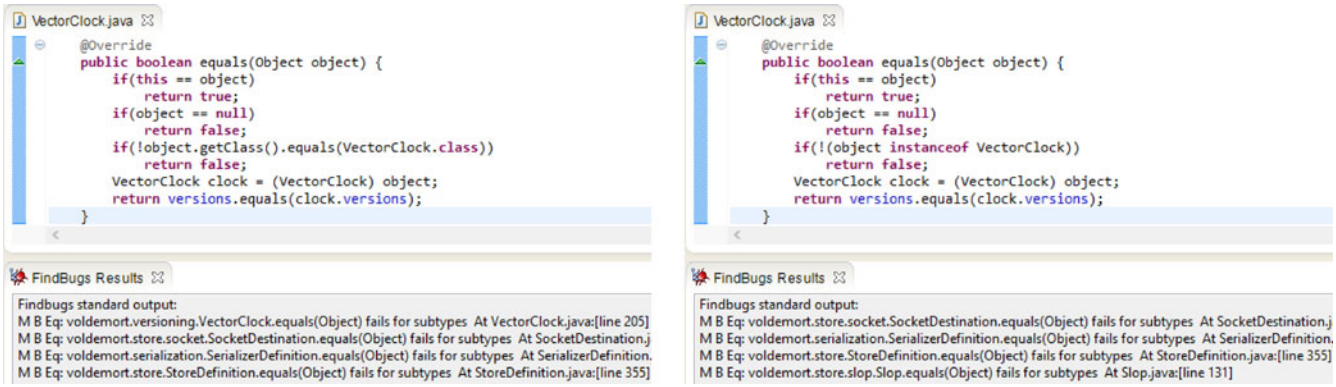


Fig. 6. Continuous FindBugs running on Voldemort. Both images show the top four warnings. The left screenshot shows the original Voldemort implementation; its first FindBugs warning suggests that the first use of `.equals(...)` is too restrictive. The developer changes `.equals(...)` to `instanceof` (right screenshot) and the top warning disappears without the developer saving the file or invoking FindBugs.

ε-continuous according to Definition 5 in Section 2. The Ant task extension executes only with each Ant build. The Eclipse plug-in has two FindBugs implementations. The developer has to manually invoke the complete FindBugs that analyzes the whole project. There is also a lighter Eclipse-Incremental-Project-Builders version that is disabled by default. This lightweight version automatically recomputes the FindBugs warning for the current editor file whenever the developer saves outstanding changes on the editor file. Both tools require the developer to perform an action to run, and neither reacts to changes made to the editor buffer. Further, since changes to one file may affect the analysis results of another, the lightweight mode of FindBugs plug-in may miss warnings.

We have used Solstice to build a proof-of-concept, open-source continuous FindBugs Eclipse plug-in, available at https://bitbucket.org/kivancmuslu/solstice-continuous-findbugs/. The plug-in uses the command-line FindBugs to analyze the `.class` files for all the classes in the currently active Eclipse project and all their dependent libraries. The plug-in's simple visualization displays the FindBugs warnings in an Eclipse view [30], which is a configurable window similar to the Eclipse Console. The plug-in immediately removes potentially stale warnings and recomputes warnings for the up-to-date codebase. Fig. 6 shows two continuous FindBugs plug-in screenshots.

6.2.2 Continuous PMD

PMD [63] is a static Java source code analysis that finds code smells and bad coding practices, such as unused variables and empty catch blocks. It is available for download as a standalone executable and as plug-ins for several IDEs, including Eclipse. Like FindBugs, it is popular and well-maintained. Unlike FindBugs, PMD works on source code. The existing Eclipse plug-in is not continuous; the developer must right-click on a project and run PMD manually.

We have used Solstice to build a proof-of-concept, open-source continuous PMD Eclipse plug-in, available at https://bitbucket.org/kivancmuslu/solstice-continuous-pmd/. The plug-in uses the command-line PMD to analyze the `.java` files for the currently active Eclipse project. The plug-in's visualization displays the PMD results in an Eclipse view. The plug-in immediately removes potentially stale results and recomputes results for the up-to-date codebase. Fig. 7 shows two continuous PMD plug-in screenshots.

6.2.3 Continuous Check Synchronization

The race detection tool Check Synchronization [15], based on technology first introduced in Eraser [68], detects potentially incorrect synchronization using dynamic checks. The tool has not yet been integrated into any IDEs.

We have used Solstice to build a proof-of-concept, open-source continuous Check Synchronization Eclipse plug-in, available at https://bitbucket.org/kivancmuslu/solstice-continuous-check-synchronization/. The plug-in searches for all classes with main methods inside the current project and runs the Check Synchronization tool on these classes. For each class with a main method, the plug-in shows the results to the developer through an Eclipse view. The plug-in immediately removes potentially stale results and recomputes new

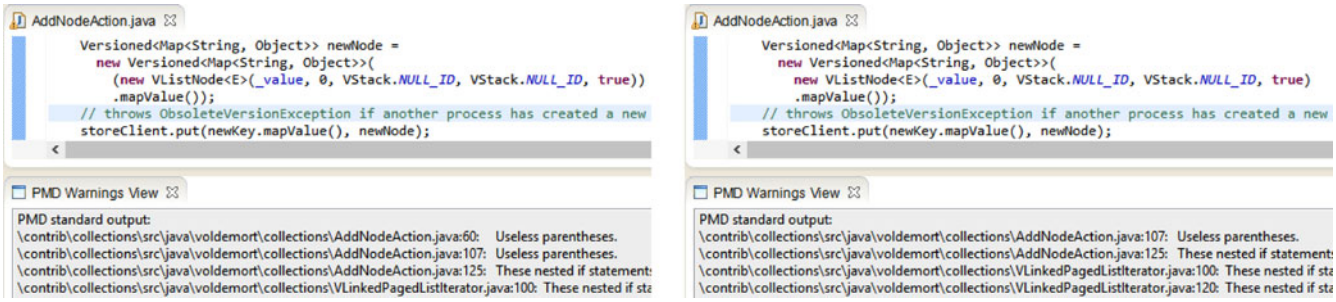


Fig. 7. Continuous PMD running on Voldemort. Both images show the top four warnings. The left screenshot shows the original Voldemort implementation; its first PMD warning suggests that the parentheses around `new` are unnecessary. The developer removes these parentheses (right screenshot) and the first warning disappears, without the developer saving the file or invoking PMD.

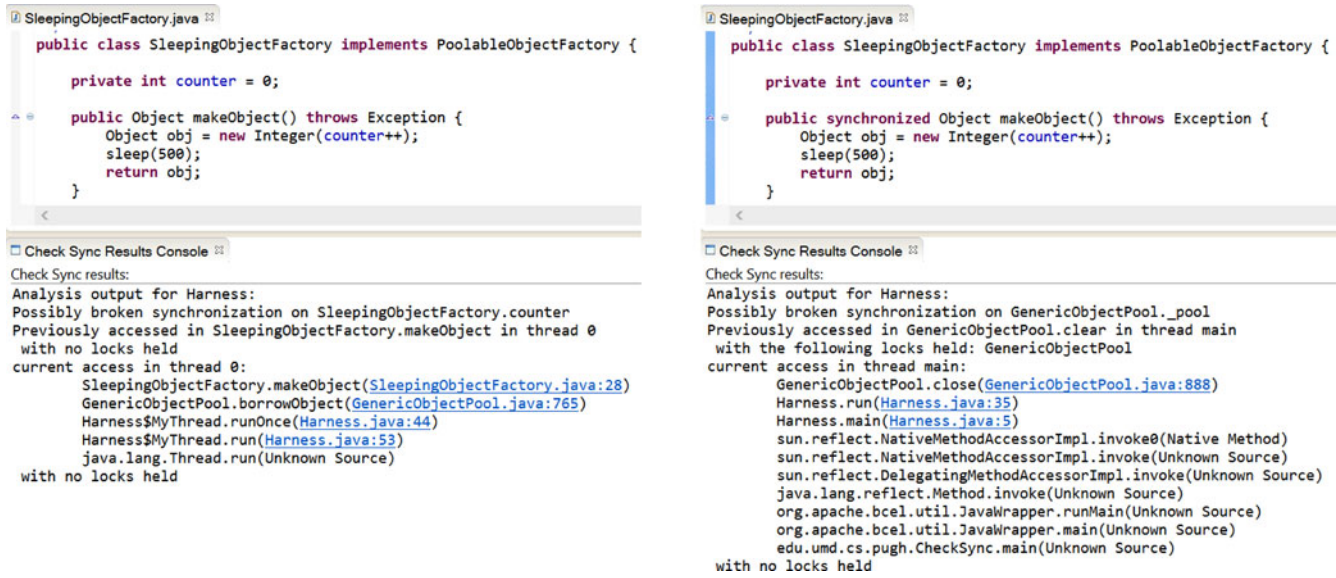


Fig. 8. Continuous data race detection running on Pool. Both images show the top warning. The left screenshot shows the original, buggy Pool implementation; its first warning suggests that the `SleepingObjectFactory.counter` field might have a data race. The developer adds `synchronized` to the method signature (right screenshot) and the top warning disappears without the developer saving the file or invoking data race detection.

results for up-to-date codebase. Fig. 8 shows two continuous Check Synchronization plug-in screenshots.

6.2.4 Continuous Testing

Continuous testing [67] uses otherwise idle CPU cycles to run tests to let the developer know as soon as possible when a change breaks a test. Continuous testing can reduce development time by up to 15 percent [65]. There are Eclipse [45], [67], Visual Studio [17], and Emacs [66] plug-ins for continuous testing. The original Eclipse plug-in [67] is ε-continuous, however it modifies Eclipse core plug-ins, making it difficult to update the implementation for new Eclipse releases; in fact, the plug-in does not support recent versions of Eclipse. By contrast, Solstice requires no modifications to the Eclipse core plug-ins and would apply across many Eclipse versions.

We have used Solstice to build a proof-of-concept, open-source continuous testing Eclipse plug-in, available at https://bitbucket.org/kivancmuslu/solstice-continuous-testing/. The plug-in runs the tests of the currently active Eclipse project. The plug-in immediately removes potentially stale test results and recomputes the test results for up-to-date codebase. The plug-in's simple visualization displays the test results in an Eclipse view. Fig. 9 shows two continuous testing plug-in screenshots.

6.3 Solstice Continuous Testing Usability Evaluation

We evaluated how continuous tools built with Solstice affect developer behavior in two ways. One of the authors used the Solstice continuous testing plug-in (Section 6.2.4) during routine debugging (Section 6.3.1), and we ran a case study (Section 6.3.2).

6.3.1 Debugging with Solstice Continuous Testing

The first author used the Solstice continuous testing plug-in ($CT_{Solstice}$) while debugging a BibTeX management project, consisting of 7 Java KLoC. The project was exhibiting `RuntimeException` crashes on a specific input. The author used $CT_{Solstice}$ while writing tests and fixing the bug. This took three days, and required an extension to the project's architecture and writing more than 100 LoC.

At the start of the debugging process, the subject program had no tests. The author wrote two tests: a regression test to validate that nothing was broken while fixing the bug, and another test for the failing input to observe the presence of the bug. The tests were 60 LoC on average and implemented the following algorithm: parse a bibliography from a hard-coded file, programmatically construct a bibliography that is expected to be equivalent to the parsed one, and assert that two bibliography representations are equivalent. The case study led to the following three observations:

$CT_{Solstice}$ can speed up discovering unknown bugs. When an input file did not exist, the program crashed with a `FileNotFoundException`. The author discovered this bug early, right after starting implementing the regression test: $CT_{Solstice}$ ran an incomplete test with an invalid path. The

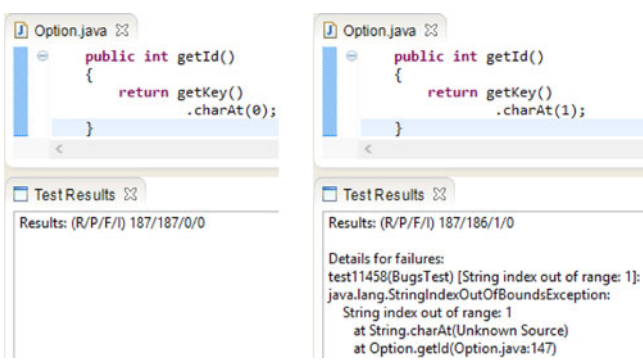


Fig. 9. Continuous testing running on Apache commons.cli. The left screenshot shows the original commons.cli implementation, for which all tests pass. The developer defines the id of an option to be its second character (right screenshot) and immediately sees that this change causes an existing test to fail, without saving the file or invoking JUnit.

author would not have thought to run this incomplete test and would have discovered the bug later, if at all.

$CT_{Solstice}$ makes debugging information available sooner. $CT_{Solstice}$ enables live programming [8], [13], [40], [78]. While debugging, developers often use print statements to view intermediate program state and assist in understanding behavior. $CT_{Solstice}$ makes the continuous testing console output and error streams available to the developer. With each edit, $CT_{Solstice}$ recomputed and redisplayed these logs, giving near-instant feedback on how changes to the code affected the print statements, even if the changes did not affect the test result. The author felt this information significantly simplified the debugging task.

$CT_{Solstice}$ is unobtrusive. During this debugging process, the author never experienced a noticeable slowdown in Eclipse's operation and never observed a stale or wrong test result.

6.3.2 Solstice Continuous Testing Case Study

To further investigate how developers interact with Solstice continuous analysis tools, we conducted a case study using the Solstice continuous testing plug-in. This case study investigates the following research questions:

RQ1: What is the perceived overhead for Solstice continuous analysis tools?

RQ2: Do developers like using Solstice continuous analysis tools?

The remainder of the section explains our case study methodology, presents the results, and discusses threats to validity.

Methodology. Each subject implemented a graph library using test-driven development (TDD). The subject was given skeleton `.java` files for the library, containing a complete Javadoc specification and a comprehensive test suite of 93 tests. The method bodies were all empty, other than throwing a `RuntimeException` to indicate that they have not been implemented. Accordingly, all tests failed initially. The subject's task was to implement the library according to the specification and to make all tests pass. The subjects were asked not to change the specification and not to change, add, or remove tests, but they could configure Eclipse as they wished and could use the Internet throughout the task.

For the case study, we recruited 10 graduate students at the University of Washington who were unfamiliar with our research.[3] Half of these subjects were randomly assigned to use JUnit (base treatment) and the remaining half were assigned to use the Solstice Continuous Testing plug-in. Subjects had varying Java (1 to 12 years), JUnit (0 to 4 years), and TDD (0 to 4 years) experience.

All sessions were conducted in a computer lab[4] at the University of Washington. After a 5-minute introduction that explained the purpose of the study, each subject completed a tutorial to learn the tool they would be using during the session. Then, each subject implemented as much of the graph library as possible within 60 minutes. We recorded the computer screen and snapshotted the subject's codebase each time it was compiled. Finally, we conducted a written exit survey, asking the subjects about their experience.

3. Subjects were recruited through a standard IRB-approved process. Participation in the study was compensated with a $20 gift card.

4. Computer specs: Intel i5-750: 2.67 GHz quad core CPU, network drive, 4 GB memory, connected to a 30-inch display.

	Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
Solstice continuous testing					
Test results were always up to date.	1	4	0	0	0
I liked using continuous testing.	2	2	1	0	0

Fig. 10. Exit survey summary for the user study subjects who used Solstice continuous testing.

Results. The test suite executed in under one second (unless the subject implemented methods that took unreasonably long). This short test suite execution time is appropriate for a small library and allowed us to answer *RQ1*; a long-running test suite would have masked the tool's overhead. All subjects agreed that the continuous testing results were always up to date (Fig. 10). In addition to the Likert-scale questions in Fig. 10, the exit survey also had a free-from question that asked the subjects to comment on their experience with the Solstice continuous testing plug-in. For example, one subject commented: "I really liked the fast feedback [from continuous testing]." Although the computers were running screen-recording software, two Eclipse instances, and a web browser, when asked during the exit survey if test results had been up to date, all five of the subjects agreed the results were up to date and none complained about lag nor any other evidence of overhead.

During the case study, we observed how developers interacted with Solstice continuous testing. After the tutorial, three developers (out of five) started using the tool as we expected, by repeating the following steps:

1) select a failure from the `Test Failures` view,
2) investigate the corresponding trace in the `Trace` view and navigate to the code locations using hyperlinks,
3) make the required code changes, and
4) verify that the failure is fixed (or discover that it is not) by looking at the updated results in the `Test Failures` view.

One subject had issues with using the two different views: she switched from the `Trace` view to the `Javadoc` view, forgot to switch back, and was confused by not being able to see the trace for the selected test failure. The last subject simply ignored the whole workflow as he was not used to using tools that provide continuous feedback. Fig. 10 shows that all but one of the subjects liked using Solstice continuous testing. One subject commented: "I really enjoyed [using Solstice continuous testing]! ...[Getting continuous feedback] in a real language like Java was pretty cool."

In addition to our qualitative results, we analyzed the recorded development history of each subject. 84 test failures were fixed by at least one developer from each treatment group. The Solstice continuous testing group fixed 52 of these failures faster, whereas the JUnit group fixed 38 faster. On average, JUnit subjects fixed 62 whereas Solstice continuous testing subjects fixed 49 test failures. As the size of our study was small, none of these results is statistically significant (all $p > 0.05$) according to the Mann-Whitney U test.

In answering *RQ2*, we conclude that Solstice continuous analyses tools are easy to use, intuitive, and unobtrusive. While our small-scale study has not shown directly the

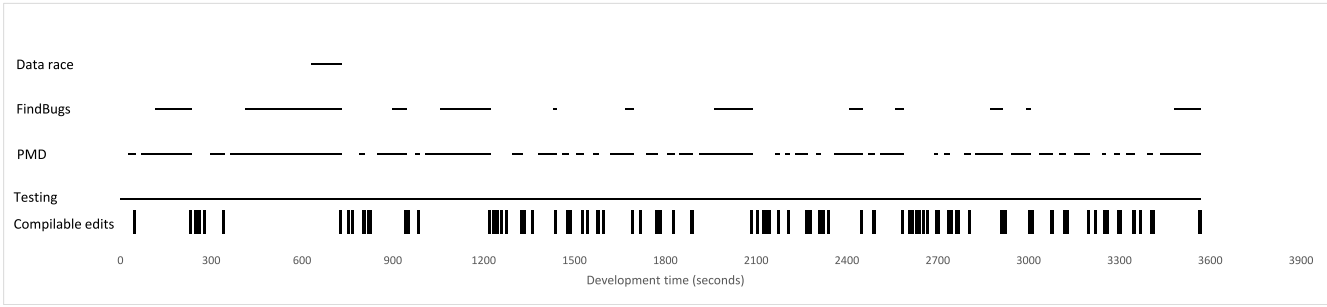


Fig. 11. Availability of Solstice analyses for one of the case study participants. The x-axis represents the development time in seconds. The vertical lines represent developer edits that yielded compilable code. Solid lines on analyses rows represent the times that the corresponding analysis would have shown up-to-date results during development. (Fig. 12 summarizes this data across all participants.)

benefits of continuous analysis tools, previous research has done so [37], [40], [52], [65], and reverifying this claim is outside the scope of this work.

Availability of results for long-running analyses. Our case study used a fast analysis: the time to execute the test suite was under a second. However, executing continuously is beneficial for all analyses, even long-running ones. Executing continuously reduces the cognitive load, since the developer neither has to decide when to run the analysis nor predict when there will be a long enough break in activity to complete the analysis. The continuous analysis eliminates or reduces wait time when the developer desires the analysis results. As discussed in Section 3.3, even potentially-stale analysis results have value. Thus, every analysis should be run continuously, under reasonable assumptions: the analysis process is run at low priority to avoid slowing down the developer's IDE, electrical power is less costly than the developer's time, and the UI that presents the results is non-obtrusive.

Given that any amount of increased availability of analysis results is beneficial, we ask how often would those analysis results be available. This section investigates our case study data in a quantitative experiment to estimate the availability of results from long-running Solstice continuous analyses. We focus on the following research question:

RQ3: How does the run time of an A_o affect the availability of its results to the corresponding A_c?

Using the development history (snapshots) of case study participants, we computed the percent availability and the average staleness of each of the four Solstice analyses of Section 6.2. Percent availability is the ratio of the total time the analysis results are up to date to the the total development time. Average staleness is the average value for how stale the currently-displayed results are (how long it has been since they were up to date), where up-to-date results are treated as 0 seconds stale.

We assume immediately-interrupting and display-stale (recall Section 3) implementations. The number of development snapshots is equal to the number of edits that yielded a compilable project. The analysis results become up to date if the developer pauses longer than $\varepsilon_a + T_{A_o}$, where:

ε_a : the continuous analysis result delay (Definition 5).
T_{A_o}: underlying A_o run time.

ε_a and T_{A_o} values are taken from Fig. 5. The analysis results become stale immediately at the beginning of the next snapshot.

Fig. 11 shows the developer edits and the availability of each analysis as a timeline, for one of the case study participants. We did this computation for each case study participant. We provide similar figures for the other participants, and the raw data at https://bitbucket.org/kivancmuslu/solstice/downloads/analysis_availability.zip.

Fig. 12 shows percent availability and average staleness of each Solstice analysis, averaged over all case study participants. Although the run time of an A_o has a negative effect on the availability of the corresponding A_c, long-running Solstice analyses would still be beneficial during development. Data race detection results are up to date 5 percent of the time.

Threats to Validity. We assess our evaluation activities in terms of simple characterizations of internal and external validity. Internal validity refers to the completeness and the correctness of the data collected through the case studies. External validity refers to the generalizability of our results to other settings.

As in other research, the possibility of a bug in the tools is a threat to internal validity. Seeing incorrect information could confuse and slow down the developers. However, we received no negative feedback about correctness.

The selection of the subject program, a simple graph library, poses a threat to external validity. Case study results for this data structure may not generalize to other software.

The selection of the offline analysis, testing, poses another threat to external validity. Case study results on how developers interact with continuous testing may not generalize to other continuous analysis tools. However, we believe the specific internal offline analysis does not affect the developer's interaction with the continuous analysis tool.

Finally, the fact that all our subjects were PhD students poses another threat to external validity. Case study results for a particular developer population may not generalize

Analysis	A_o Run time	Availability	Avg. Staleness
Testing	0.01 s.	99.7%	0.003 s.
PMD	25 s.	61.2%	10.9 s.
FindBugs	74 s.	33.7%	108.6 s.
Data race	293 s.	5.7%	1124 s.

Fig. 12. A_o run time, percent availability, and average staleness of each Solstice continuous analysis, averaged over all case study participant data. The results suggest that even a continuous long-running analysis can provide value during development. (Fig. 11 shows an in-depth look at a single participant.)

to other developer populations. However, none of the subjects knew Solstice continuous testing before the case study and their experiences with JUnit, Eclipse, and Java varied. Most subjects had professional experience through internships in industry.

6.4 Alternate Implementation Strategies

There are ways other than maintaining a copy codebase to convert offline analyses into continuous ones. Very fast offline analyses can run in the IDE's UI thread. While technically such an analysis would block the developer, the developer would never notice the blocking because of its speed. Most analyses are not fast enough for this approach to be feasible.

It is possible to reduce the running time of an offline analysis by making it incremental [64]. An incremental code analysis takes as input the analysis result on an earlier snapshot of the code and the edits made since that snapshot. Examples include differential static analyses [54], differential symbolic execution [62], and incremental checking of data structure invariants [71]. When the differences are small, incremental analyses can be significantly faster. With this speed increase, incremental analyses may be used continuously by blocking the developer whenever the analysis runs. Incremental code compilation [53] is one popular incremental, continuous analysis integrated into many IDEs. However, many analyses cannot be made incremental efficiently because small code changes may force these analyses to explore large, distant parts of the code. Further, making an analysis incremental can be challenging, requiring a complete analysis redesign. The process is similar to asking someone to write an efficient, greedy algorithm that solves a problem for which only an inefficient algorithm that requires global information is known.

While many analyses cannot be made incremental or efficient enough to run continuously while blocking the developer, those that can still benefit from being built using Codebase Replication. An impure analysis is freed from the burden of maintaining a copy codebase, as Codebase Replication maintains the copy codebase and lets the analyses own it exclusively. Codebase Replication allows long-running analyses to execute on a recent snapshot and produce results that may be slightly stale, whereas other approaches would not.

Codebase Replication uses a step-based execution model to execute A_o on the copy codebase. Codebase Replication could instead use a build tool, such as Apache Maven or Ant, letting an analysis author declare how A_o runs via build files. Although using a build tool might further simplify the A_c implementation, it would also limit A_c to the capabilities of the build tool. Step-based execution permits the analysis author to implement A_c using arbitrary Java code or to define A_c as a one-step analysis that executes a build tool.

It is possible to implement memory-change-triggered continuous analysis tools by combining a file-change continuous analysis framework such as Incremental Project Builders with tools that automatically save changes periodically such as the Smart Save plug-in [72]. However, running an analysis on a separate codebase has additional benefits. First, the developer never experiences any unwanted side-effects, such as crashes or code modifications due to impurity, of the analysis. Second, for longer-running analyses, when there is a conflicting developer edit, Codebase Replication can let the analysis finish its execution on the copy codebase and produce correct—albeit potentially stale—results.

7 Related Work

This section places our work in the context of related research. Section 7.1 discusses other approaches to building continuous analysis tools and Section 7.2 discusses existing continuous analysis tools and their benefits.

7.1 Building Continuous Analysis Tools

As we have described, an ε-continuous analysis exhibits both currency and isolation. Codebase Replication simplifies building such analyses. Alternatively, developers can build such tools by using IDEs' APIs to listen to source code edits. For example, Eclipse's `IResourceChangeListener` [22] and `IDocumentListener` [21] APIs broadcast file-level and memory-level changes, respectively. Eclipse's Java incremental compiler [27] and reconciler compiler [26] use these APIs; however, these analyses are written by the IDE developers, and building ε-continuous analyses using these primitive APIs is prohibitively difficult for third-party developers.

Some specialized development domains make building a limited set of continuous analyses simple. For example, a spreadsheet can be thought of as an IDE for data-intensive programs that reruns these programs on every code or data update. VisiProg [40], [52] proposed to extend this paradigm to general programming languages, but Codebase Replication is the first implementation that provides isolation and currency. As another example, live programming [8], [13], [78], which eases development by executing a fast-running program on a specific input as that program is being developed, is a special case of continuous analysis.

The rest of this section discusses alternate ways of creating continuous analysis tools and compares them to Codebase Replication. None of the existing approaches provides both isolation and currency, although some provide one or the other.

7.1.1 Methods that Yield Limited Currency but Lack Isolation

IDEs provide higher-level frameworks than the primitive listeners described earlier. For example, to simplify implementing build-triggered continuous analyses, Eclipse provides Incremental Project Builders [23]. This mechanism can be used to execute an analysis on the on-disk version of the program every time the incremental compiler runs. (Note that when auto-build is enabled in Eclipse, the code builds every time it is saved to disk, so build-trigger becomes equivalent to file-change-trigger.) This mechanism enables building analyses that have some currency, although Codebase Replication's memory-change access provides better currency by enabling the analyses to run on a more recent version of the program than one that has been saved to disk. Further, unlike Codebase Replication, this mechanism does not allow for analysis isolation. The analyses run on the developer's on-disk copy, meaning that an impure analysis's changes

directly alter the developer's code, and a developer's concurrent changes may affect the analysis.

IDEs also provide frameworks that simplify building a limited set of continuous analyses with memory-change currency. For example, Eclipse's **X**text [80] simplifies extending Eclipse to handle new languages. **X**text provides parsing, compilation, auto-complete, quick fix, and refactoring support, but is limited to building language extensions. Meanwhile Codebase Replication provides memory-change currency for arbitrary source or binary code analyses. Similarly to Incremental Project Builders, and unlike Codebase Replication, Xtext does not allow for analysis isolation as a developer's concurrent changes may affect the analysis. Further, Xtext does not support impure analyses.

7.1.2 Methods that Yield Limited Isolation but Lack Currency

Integration servers, such as Jenkins [48], can enable certain kinds of continuous analyses. An integration server maintains an isolated copy of the program under development and automatically fetches new changes, builds the program, runs static and dynamic analyses, and generates summaries for developers and project managers. However, integration servers lack currency, as they cannot be memory-change- or file-change-triggered; typically they are triggered periodically or by events such as a commit. Modern collaboration portals, such as github.com, bitbucket.org, and googlecode.com, integrate awareness analyses and create interfaces for developers to get feedback on the state of their programs. This is also a step toward making analyses continuous, as the portals can automate the running of analyses and can analyze multiple developers' codebases and notify the developers of analysis results. However, this mechanism also lacks currency as the analyses cannot be triggered by memory changes, file changes, or even most version control operations.

IDEs provide APIs that serialize accesses to the codebase, which can ensure partial isolation. For example, Eclipse provides a Jobs API [29] that enables third-party developers to schedule jobs that access the codebase. There is no isolation: these jobs either block each other and the developer edits, or they occur concurrently on the same codebase. In contrast, Codebase Replication can run an analysis on the copy codebase while letting the developer work, achieving true isolation, and providing native support for impure continuous analyses.

7.2 Existing Continuous Analysis Tools

Continuous analysis tools help developers by reducing the notification delay of code changes' effects on analysis results. For example, continuous testing [65], [66], [67] executes a program's test suite as the program is being developed. In a study, continuous testing made developers three times as likely to finish programming tasks by a deadline [66] and reduced the time needed to finish a task by 10-15 percent [65]. Similarly, continuous data testing greatly reduced data entry errors [59], and continuous compilation made developers twice as likely to finish programming tasks by a deadline [66]. Some continuous analyses [11], [12], [39], [58] can be speculative [9] by predicting developers' likely future actions and executing them in the background to inform the developers' decision making. Such tools have the potential to further increase the benefits of continuous analyses.

Fig. 13 lists previous continuous analysis tools of which we are aware. Although IDEs provide frameworks and APIs to simplify the creation of continuous analyses, Fig. 13 shows that most existing third-party IDE-integrated continuous analysis tools are not ε-continuous, lacking either in isolation or currency. From the 16 file-change-triggered and build-triggered tools in Fig. 13, we selected the 7 with evidence of development or maintenance within the last year and contacted their developers to ask if they had considered making their analyses run whenever the in-memory code changes or compiles. We received responses from the developers of 4 of the 7 tools, GoClipse, InPlace Activator, TSLint, and TypeScript (TSLint and TypeScript are developed by an overlapping set of developers). All the developers thought making analyses continuous was a good idea, with one remarking that this would be hard to do, another that he didn't have enough time to implement this feature, and the third pointing out that part of the tool already has this continuous behavior, although not all of the tool's analyses are continuous. We conclude that developers prefer to build ε-continuous tools for at least some analyses, but that the effort required to build such tools prevents their development.

Building an ε-continuous analysis without Codebase Replication is prohibitively difficult and results in poor designs. As an example, an earlier Eclipse continuous testing plug-in [67] is ε-continuous, but making it ε-continuous required hacking into the core Eclipse plug-ins, so it does not work with subsequent versions of Eclipse. As another example, to achieve isolation, Quick Fix Scout [58] embeds and maintains its own copy codebase in the developer's workspace, significantly complicating its design and implementation. Further, embedding replication logic inside the analysis makes it difficult to debug the replication logic, as bugs that break the synchronization between the copy codebase and the developer's codebase are difficult to isolate. In contrast, as Section 6.2 has argued, Solstice makes it easier to write Eclipse-integrated analyses that maintain isolation and currency.

8 Contributions

While useful to developers, continuous analyses are rare because building them is difficult. We classified the major design decisions in building continuous analysis tools, and identified the major challenges of building continuous analyses as *isolation* and *currency*. We designed Codebase Replication, which solves these challenges by maintaining an in-sync copy of the developer's code and giving continuous analyses exclusive access to this copy codebase. We further introduced a step-based execution model that improves Codebase Replication's currency. We have built Solstice, a Codebase Replication prototype for Eclipse, and used it to build four open-source, publicly-available continuous analysis Eclipse plug-ins. We have used these plug-ins to evaluate Codebase Replication's effectiveness and usability.

We have evaluated Codebase Replication (1) on performance benchmarks, showing that Solstice-based tools have

Tool	Currency			Isolation
	Trigger	Interruption	Staleness	
Quick Fix Scout [58] consequences of Eclipse quick fixes	Memory-change*	Immediately interrupt	Immediately remove	Full
Eclipse reconciler compiler [26]	Memory-change*	Immediately interrupt	Immediately remove	Developer
JKind [49] Eclipse plug-in for JKind language	Memory-change	Immediately interrupt	Immediately remove	Developer
eVHDL [34] Eclipse plug-in for VHDL language	Memory-change	Immediately interrupt	Immediately remove	Developer
Scribble [70] Eclipse plug-in for Scribble language	Memory-change	Immediately interrupt	Immediately remove	Developer
wNesC [79] Eclipse plug-in for NesC language	Memory-change	Immediately interrupt	Immediately remove	Developer
OcaIDE [61] Eclipse plug-in for OCaml language	Memory-change	Immediately interrupt	Never remove	Developer
WitchDoctor [36] auto-completes manual refactorings	Memory-change	Never interrupt	Immediately remove	Developer
NCrunch [60] runs tests and computes coverage	Memory-change	Unknown†	Immediately remove	Full
DocMLET [20] Eclipse plug-in for LaTeX language	Memory-change	Unknown†	Immediately remove	Developer
dLabPro [19] Eclipse plug-in for dLabPro language	Memory-change	Unknown†	Immediately remove	Developer
Embedded CAL [32] embeds CAL language into Java	Memory-change	Unknown†	Immediately remove	Developer
Eclipse continuous testing [67]	Memory-change	Unknown†	Immediately remove	Developer
Sureassert UC [74] runs tests and computes coverage	Memory-change	Unknown†	Immediately remove	Developer
Blueprint [8] searches code examples from the Internet	Memory-change	Unknown†	Unknown†	Unknown†
Forms/3 [78] evaluates and visualizes the source code	Memory-change	Unknown†	Unknown†	Unknown†
Lighthouse [55] summarizes overall development effort	File-change	Unknown†	Immediately remove	Developer
WeCode [39] detects collaboration conflicts	File-change	Unknown†	Unknown†	Full
Eclipse incremental compiler [27]	Other (build)	Immediately interrupt	Immediately remove	Developer
FindBugs [35] bug detector	Other (build)	Immediately interrupt	Immediately remove	Developer
Checkstyle [24] detects code style violations	Other (build)	Never interrupt	Immediately remove	Developer
Infinitest [45] runs tests	Other (build)	Never interrupt	Never remove	Developer
TypeScript [76] Eclipse plug-in for TypeScript language	Other (build)◊	Never interrupt	Immediately remove	Developer
TSLint [75] lints type script code	Other (build)	Never interrupt	Immediately remove	Developer
SConsolidator [69] Eclipse plug-in for SCons build system	Other (build)	Unknown†	Immediately remove	Developer
JUnitLoop [51] runs test	Other (build)	Unknown†	Immediately remove	Developer
InPlace Activator [46] activates and updates source plug-ins	Other (build)	Unknown†	Immediately remove	Developer
GoClipse [38] Eclipse plug-in for Go language	Other (build)	Unknown†	Immediately remove	Developer
EclipseFP [31] Eclipse plug-in for Haskell language	Other (build)	Unknown†	Immediately remove	Developer
JDE [47] Eclipse plug-in for BlackBerry Java language	Other (build)	Unknown†	Immediately remove	Developer
CAL [14] Eclipse plug-in for CAL language	Other (build)	Unknown†	Immediately remove	Developer
JSON Schema Validation [50] validates JSON files	Other (build)	Unknown†	Immediately remove	Developer
Metrics [28] such as cyclomatic complexity	Other (build)	Unknown†	Unknown†	Unknown†
Visual Studio continuous testing [17]	Other (build)	Unknown†	Unknown†	Unknown†
CDT [59] detects likely database update errors	Other (database)	Never interrupt	Never remove	Developer
Crystal [11] detects collaboration conflicts	Periodic (10 m)	Never interrupt	Never remove	Full
APE [3] Eclipse plug-in for AnsProg programming environment	Manual◊	N/A	N/A	N/A
ML-Dev [56] Eclipse plug-in for Standard ML language	Manual◊	N/A	N/A	N/A
EMFText [33] Eclipse plug-in for Ecore metamodel	Manual◊	N/A	N/A	N/A
Bio-PEPA [5] Eclipse plug-in for Bio-PEPA language	Manual▽	N/A	N/A	N/A
Hibernate Synchronizer [41] Eclipse plug-in for Hibernate persistence framework	Manual▽	N/A	N/A	N/A
n-gram-based code completion [42]	Unknown†	Unknown†	Unknown†	Unknown†

* The analysis is 0.5 seconds delayed. For other analyses, the documentation does not describe a delay.

† The publications and tool documentation do not give adequate information to categorize the tool.

◊ In addition to the main analysis listed in the table, the plug-in includes secondary memory-change-triggered continuous analyses.

▽ In addition to the main analysis listed in the table, the plug-in includes secondary file-change-triggered continuous analyses.

Fig. 13. Previous continuous analysis tools, categorized according to the design dimensions of Section 3. The first six tools are ε-continuous. The continuous IDE plug-ins for language extensions provide parsing, compilation, auto-complete, quick fix, and/or refactoring support. "Developer" isolation means that the developer is isolated from the changes made by an impure analysis, but the analysis is not isolated from the developer's changes; this is adequate for building only *pure* ε-continuous analysis tools. For IDEs that support auto-build, build-triggered analyses are equivalent to file-change-triggered analyses.

negligible overhead and have access to the up-to-date code with less than 2.5 milliseconds delay, (2) by building continuous analysis tools, demonstrating that Codebase Replication and Solstice can be used for a variety of continuous tools including testing, heuristic bug finding, and data race detection and that the effort necessary to build new continuous analysis tools is low (each tool required on average 710 LoC and 20 hours of implementation effort), and (3) with case studies with developers that show that Solstice-based tools are intuitive and easy-to-use.

Codebase Replication provides a simple alternative to redesigning offline analysis logic to work continuously. Overall, the cost of converting an offline analysis to a continuous one with Codebase Replication is low. Further, the benefits of continuous analysis tools greatly outweigh the cost of building them with Codebase Replication. We believe that Codebase Replication, and our implementation, will enable developers to quickly and easily build continuous tools, and will greatly increase the availability of such tools to developers. These tools will reduce the interruptions developers face and the delay before developers learn the effects of their changes, and consequently will positively impact software quality and the developer experience.

Acknowledgments

This work was funded by the US National Science Foundation (NSF) grants CCF-0963757, CCF-1016701, and CCF-1446683. Luke Swart worked on the marker visualization that was used in the case study. Ezgi Mercan worked on an early stage of the Solstice prototype implementation. The authors thank Deepak Azad, Daniel Megert, and Stephan Herrmann for their help with the Eclipse internals throughout Solstice development.

REFERENCES

[1] Apache Ant. (2015, Jan. 29) [Online]. Available: http://ant.apache.org/

[2] Apache Maven. (2015, Jan. 29) [Online]. Available: http://maven.apache.org/

[3] AnsProlog programming environment. (2015, Feb. 23) [Online]. Available: https://github.com/robibbotson/APE/

[4] Extended ASM, a byte code manipulator. Distributed as part of Annotation File Utilities. (2014, Sep. 21) [Online]. Available: http://types.cs.washington.edu/annotation-file-utilities/

[5] An Eclipse plug-in supporting the Bio-PEPA domain specific language. (2015, Feb. 13) [Online]. Available: https://github.com/Bio-PEPA/Bio-PEPA/

[6] B. W. Boehm, *Software Engineering Economics*. Upper Saddle River, NJ, USA: Prentice-Hall, 1981.

[7] C. Boekhoudt, "The big bang theory of IDEs," *Queue*, vol. 1, no. 7, pp. 74–82, Oct. 2003.

[8] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer, "Example-centric programming: Integrating web search into the development environment," in *Proc. 28th Conf. Human Factors Comput. Syst.*, Atlanta, GA, USA, Apr. 2010, pp. 513–522.

[9] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Speculative analysis: Exploring future states of software," in *Proc. Workshop Future Softw. Eng. Res.*, Santa Fe, NM, USA, Nov. 2010, pp. 59–63.

[10] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Crystal: Precise and unobtrusive conflict warnings," in *Proc. 8th Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. Tool Demonstration Track*, Szeged, Hungary, Sep. 2011, pp. 444–447.

[11] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Proactive detection of collaboration conflicts," in *Proc. 8th Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Szeged, Hungary, Sep. 2011, pp. 168–178.

[12] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Early detection of collaboration conflicts and risks ," *IEEE Trans. Softw. Eng.*, vol. 39, no. 10, pp. 1358–1375, Oct. 2013.

[13] M. M. Burnett, J. W. Atwood Jr., and Z. T. Welch, "Implementing level 4 liveness in declarative visual programming languages," in *Proc. Symp. Visual Lang.*, Sep. 1998, pp. 126–133.

[14] An Eclipse plug-in supporting the CAL domain specific language. (2015, Feb. 13) [Online]. Available: https://github.com/levans/Embedded-CAL/

[15] Check synchronization. (2014, Sep.) [Online]. Available: http://www.cs.umd.edu/class/fall2004/cmsc433/checkSync.html

[16] Continuous analysis. (2015, Jan. 26) [Online]. Available: http://www.klocwork.com/products/documentation/current/Continuous_analysis

[17] Continuous testing for Visual Studio. (2014, Sep. 21) [Online]. Available: http://ox.no/software/continuoustesting/

[18] Crossword Sage. (2014, Sep. 21) [Online]. Available: http://sourceforge.net/projects/crosswordsage/

[19] An Eclipse plug-in supporting the dLabPro domain specific language. (2015, Feb. 13) [Online]. Available: https://github.com/matthias-wolff/dLabPro-Plugin/

[20] An Eclipse plug-in supporting the LATEX domain specific language. (2015, Feb. 13) [Online]. Available: https://github.com/walware/docmlet/

[21] Eclipse API: IDocumentListener. (2014, Oct. 02) [Online]. Available: http://help.eclipse.org/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/jface/text/IDocumentListener.html

[22] Eclipse API: IResourceChangeListener. (2014, Oct. 02) [Online]. Available: http://help.eclipse.org/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/core/resources/IResourceChangeListener.html

[23] Eclipse project builders and natures. (2014, Sep. 21) [Online]. Available: http://www.eclipse.org/articles/Article-Builders/builders.html

[24] Eclipse-Checkstyle integration. (2014, Sep. 21) [Online]. Available: http://eclipse-cs.sourceforge.net/

[25] Eclipse: How do I use a model reconciler? (2015, Jan. 31) [Online]. Available: https://wiki.eclipse.org/FAQ_How_do_I_use_a_model_reconciler%3F

[26] Eclipse: Java compile errors/warnings preferences. (2014, Sep. 21) [Online]. Available: http://help.eclipse.org/topic/org.eclipse.jdt.doc.user/reference/preferences/java/compiler/ref-preferences-errors-warnings.htm

[27] Eclipse: JDT core component. (2014, Sep. 21) [Online]. Available: http://www.eclipse.org/jdt/core/index.php

[28] Eclipse Metrics plug-in. (2014, Sep. 21) [Online]. Available: http://sourceforge.net/projects/metrics/

[29] Eclipse: The jobs API. (2014, Sep. 21) [Online]. Available: http://www.eclipse.org/articles/Article-Concurrency/jobs-api.html

[30] Eclipse: Views. (2014, Sep. 21) [Online]. Available: http://help.eclipse.org/topic/org.eclipse.platform.doc.isv/reference/extension-points/org_eclipse_ui_views.html

[31] An Eclipse plug-in supporting the Haskell domain specific language. (2015, Feb. 13) [Online]. Available: https://github.com/JPMoresmau/eclipsefp/

[32] Embedded CAL. (2015, Feb. 13) [Online]. Available: https://github.com/levans/Embedded-CAL/

[33] An Eclipse plug-in supporting the Ecore metamodel. (2015, Feb. 13) [Online]. Available: https://github.com/DevBoost/EMFText/

[34] eVHDL: Eclipse plug-in for developing VHDL code. (2014, Oct. 2) [Online]. Available: https://github.com/HepaxCodex/eVHDL/

[35] FindBugs. (2014, Sep. 21) [Online]. Available: http://findbugs.sourceforge.net/

[36] S. R. Foster, W. G. Griswold, and S. Lerner, "WitchDoctor: IDE support for real-time auto-completion of refactorings," in *Proc. 34th Int. Conf. Softw. Eng.*, Zurich, Switzerland, Jun. 2012, pp. 222–232.

[37] D. S. Glasser, "Test factoring with `amock`: Generating readable unit tests from system tests," Master's thesis, Massachusetts Inst. Technol., Boston, MA, USA, Aug. 2007.

[38] An Eclipse plug-in supporting the Go domain specific language. (2015, Feb. 13) [Online]. Available: https://github.com/GoClipse/goclipse/

[39] M. L. Guimarães, and A. R. Silva, "Improving early detection of software merge conflicts," in *Proc. 34th Int. Conf. Softw. Eng.*, Zurich, Switzerland, Jun. 2012, pp. 342–352.

[40] P. Henderson and M. Weiser, "Continuous execution: The VisiProg environment," in *Proc. 8th Int. Conf. Softw. Eng.*, London, England, Aug. 1985, pp. 68–74.

[41] Hibernate Synchronizer. (2015, Feb. 13) [Online] Available: https://github.com/jhudson8/hibernate-synchronizer/

[42] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proc. 34th Int. Conf. Softw. Eng.*, ICSE'12, Zurich, Switzerland, Jun. 2012, pp. 837–847.

[43] D. Hovemeyer and W. Pugh, "Finding bugs is easy," in *Proc. 19th Conf. Object-Oriented Programm. Syst., Language Appl.*, Vancouver, BC, Canada, Oct. 2004, pp. 132–136.

[44] Hudson. (2014, Sep. 21) [Online]. Available: http://hudson-ci.org/

[45] Infinitest. (2014, Sep. 21) [Online]. Available: http://infinitest.github.io/

[46] InPlace Activator. (2015, Feb. 13) [Online]. Available: https://github.com/eirikg/no.javatime.inplace/

[47] An Eclipse plug-in supporting the Blackberry Java domain specific language. (2015, Feb. 13) [Online]. Available: https://github.com/blackberry/Eclipse-JDE/

[48] Jenkins. (2014, Sep. 21) [Online]. Available: http://jenkins-ci.org/

[49] Lustre plug-in for Eclipse with JKind analysis support. (2014, Oct. 2) [Online]. Available: https://github.com/agacek/jkind-xtext/

[50] JSON Schema Validation. (2015, Feb. 13) [Online]. Available: https://github.com/sabina-jung/JSON-Schema-Validation-Eclipse/

[51] JUnitLoop. (2015, Feb. 13) [Online]. Available: https://github.com/DevBoost/JUnitLoop/

[52] R. R. Karinthi and M. Weiser, "Incremental re-execution of programs," in *Proc. Symp. Interpreters Interpretive Tech.*, St. Paul, MN, USA, 1987, pp. 38–44.

[53] H. Katzan Jr., "Batch, conversational, and incremental compilers," in *Proc. Am. Fed. Inf. Process. Soc. Spring Joint Comput. Conf.*, Boston, MA, USA, May 1969, pp. 47–56.

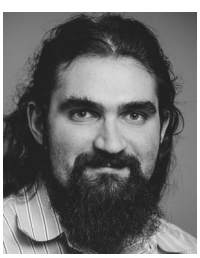
[54] S. K. Lahiri, K. Vaswani, and C. A. R. Hoare, "Differential static analysis: Opportunities, applications, and challenges," in *Proc. Workshop Future Softw. Eng. Res.*, Santa Fe, NM, USA, Nov. 2010, pp. 201–204.

[55] Lighthouse. (2015, Feb. 13) [Online]. Available: https://github.com/uci-sdcl/lighthouse/

[56] An Eclipse plug-in supporting the standard ML language. (2015, Feb. 13) [Online]. Available: https://github.com/andriusvelykis/ml-dev/

[57] K. Muşlu, Y. Brun, M. D. Ernst, and D. Notkin, "Making offline analyses continuous," in *Proc. 9th Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Saint Petersburg, Russia, Aug. 2013, pp. 323–333.

[58] K. Muşlu, Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Speculative analysis of integrated development environment recommendations," in *Proc. 3rd Conf. Object-Oriented Programm. Syst., Lang. Appl.*, Tucson, AZ, USA, Oct. 2012, pp. 669–682.
[59] K. Muşlu, Y. Brun, and A. Meliou, "Data debugging with continuous testing ," in *Proc. 9th Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng., New Ideas Track*, Saint Petersburg, Russia, Aug. 2013, pp. 631–634.
[60] NCrunch. (2015, Jan. 26) [Online]. Available: http://www.ncrunch.net/
[61] OcaIDE: OCaml plug-in for Eclipse. (2014, Oct. 2) [Online]. Available: https://github.com/nbros/OcaIDE/
[62] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu, "Differential symbolic execution," in *Proc. 16th Symp. Found. Softw. Eng.*, Atlanta, GA, USA, Nov. 2008, pp. 226–237.
[63] PMD. (2014, Sep. 21) [Online]. Available: http://pmd.sourceforge.net/
[64] G. Ramalingam and T. Reps, "A categorized bibliography on incremental computation," in *Proc. 20th Symp. Principles Programm. Lang.*, Charleston, SC, USA, Jan. 1993, pp. 502–510.
[65] D. Saff and M. D. Ernst, "Reducing wasted development time via continuous testing," in *Proc. 14th Int. Symp. Softw. Rel. Eng.*, Denver, CO, USA, Nov. 2003, pp. 281–292.
[66] D. Saff and M. D. Ernst, "An experimental evaluation of continuous testing during development," in *Proc. Int. Symp. Softw. Testing Anal.*, Boston, MA, USA, Jul. 2004, pp. 76–85.
[67] D. Saff and M. D. Ernst, "Continuous testing in Eclipse," in *Proc. 27th Int. Conf. Softw. Eng.*, St. Louis, MO, USA, May 2005, pp. 668–669.
[68] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, Nov. 1997.
[69] SConsolidator. (2015, Feb. 13) [Online]. Available: https://github.com/IFS-HSR/Sconsolidator/
[70] An Eclipse plug-in supporting the Scribble domain specific language. (2014, Oct. 2) [Online]. Available: https://github.com/glozachm/scribble-plug-in/
[71] A. Shankar and R. Bodík, "Ditto: Automatic incrementalization of data structure invariant checks (in Java)," in *Proc. Conf. Programm. Lang. Design Implementation*, San Diego, CA, USA, Jun. 2007, pp. 310–319.
[72] Smart save plug-in. (2014, Sep. 21) [Online]. Available: http://marketplace.eclipse.org/content/smart-save
[73] SonarQube. (2015, Jan. 29) [Online]. Available: http://www.sonarqube.org/
[74] Sureassert UC. (2015, Jan. 26) [Online]. Available: http://www.sureassert.com/uc/
[75] TSLint. (2015, Feb. 13) [Online]. Available: https://github.com/palantir/eclipse-tslint/
[76] An Eclipse plug-in supporting the TypeScript domain specific language. (2015, Feb. 13) [Online]. Available: https://github.com/palantir/eclipse-typescript/
[77] Voldemort. (2014, Sep. 21) [Online]. Available: http://www.project-voldemort.com/voldemort/
[78] E. M. Wilcox, J. W. Atwood Jr., M. M. Burnett, J. J. Cadiz, and C. R. Cook, "Does continuous visual feedback aid debugging in direct-manipulation programming systems?" in *Proc. Conf. Human Factors Comput. Syst.*, Atlanta, GA, USA, Mar. 1997, pp. 258–265.
[79] An Eclipse plug-in supporting the NesC domain specific language. (2015, Feb. 13) [Online]. Available: https://github.com/mimuw-distributed-systems-group/wNesC-Eclipse-Plug-in/
[80] Xtext. (2014, Oct. 2) [Online]. Available: https://www.eclipse.org/Xtext/



Kıvanç Muşlu received the BSc degree from the Koç University in 2010, and the MSc degree from the University of Washington in 2012. He is currently working toward the PhD degree in computer science & engineering at the University of Washington. His research focuses on software engineering, specifically increasing developer productivity and reducing developer mistakes by exploiting copy codebases. He is a member of the IEEE, the ACM, and ACM SIGSOFT. More information is available on his homepage: http://www.kivancmuslu.com/.

Yuriy Brun received the MEng degree from the Massachusetts Institute of Technology in 2003, and the PhD degree from the University of Southern California in 2008. He is an assistant professor in the School of Computer Science, University of Massachusetts, Amherst. He completed his postdoctoral work in 2012 at the University of Washington, as a CI fellow. His research focuses on software engineering, distributed systems, and self-adaptation. He received the US National Science Foundation (NSF) CAREER award in 2015, a Microsoft Research Software Engineering Innovation Foundation Award in 2014, and an IEEE TCSC Young Achiever in Scalable Computing Award in 2013. He is a member of the IEEE, the ACM, and ACM SIGSOFT. More information is available on his homepage: http://www.cs.umass.edu/~brun/.

Michael D. Ernst is a professor of computer science & engineering, University of Washington. His research aims to make software more reliable, more secure, and easier (and more *fun*!) to produce. His primary technical interests are in software engineering and related areas, including programming languages, type theory, security, program analysis, bug prediction, testing, and verification. His research combines strong theoretical foundations with realistic experimentation, with an eye to changing the way that software developers work. He was previously a tenured professor at MIT, and before that a researcher at Microsoft Research. More information is available on his homepage: http://homes.cs.washington.edu/~mernst/. He is a senior member of the IEEE.

David Notkin (1955-2013) received the ScB degree from Brown University in 1977 and the PhD degree from Carnegie Mellon University in 1984. He served as a professor and Bradley chair of computer science & engineering at the University of Washington, which he joined in 1984. His research interests were in software engineering in general and in software evolution in particular. He received the US National Science Foundation Presidential Young Investigator Award; served as the program chair of the first ACM SIGSOFT Symposium on the Foundations of Software Engineering; served as a program co-chair of the 1995 International Conference on Software Engineering; chaired the steering committee of the International Conference on Software Engineering; served as the general chair of the 2013 International Conference on Software Engineering; served as a charter associate editor and later as an editor-in-chief of the *ACM Transactions on Software Engineering and Methodology*; served as an associate editor of the *IEEE Transactions on Software Engineering*; was fellow of the ACM and IEEE; received the ACM SIGSOFT Distinguished Service Award, the ACM SIGSOFT Outstanding Research Award, the ACM SIGSOFT Influential Educator Award, and the A. Nico Habermann Award; served as the chair of ACM SIGSOFT; served as the department chair of Computer Science & Engineering; and received the University of Washington Distinguished Graduate Mentor Award.