Making Offline Analyses Continuous

Kıvanç Muşlu[†], Yuriy Brun[UM], Michael D. Ernst[†], David Notkin[†]
[†]Computer Science & Engineering
University of Washington
Seattle, WA, USA
{kivanc, mernst}@cs.washington.edu

[UM]School of Computer Science
University of Massachusetts
Amherst, MA, USA
brun@cs.umass.edu

ABSTRACT

Developers use analysis tools to help write, debug, and understand software systems under development. A developer's change to the system source code may affect analysis results. Typically, to learn those effects, the developer must explicitly initiate the analysis. This may interrupt the developer's workflow and/or the delay until the developer learns the implications of the change. The situation is even worse for impure analyses — ones that modify the code on which it runs — because such analyses block the developer from working on the code.

This paper presents Codebase Replication, a novel approach to easily convert an offline analysis — even an impure one — into a continuous analysis that informs the developer of the implications of recent changes as quickly as possible after the change is made. Codebase Replication copies the developer's codebase, incrementally keeps this copy codebase in sync with the developer's codebase, makes that copy codebase available for offline analyses to run without disturbing the developer and without the developer's changes disturbing the analyses, and makes analysis results available to be presented to the developer.

We have implemented Codebase Replication in Solstice, an open-source, publicly-available Eclipse plug-in. We have used Solstice to convert three offline analyses — FindBugs, PMD, and unit testing — into continuous ones. Each conversion required on average 436 NCSL and took, on average, 18 hours. Solstice-based analyses experience no more than 2.5 milliseconds of runtime overhead per developer action.

Categories and Subject Descriptors:
D.2.6 [Software Engineering]: Programming Environments
D.2.3 [Software Engineering]: Coding Tools and Techniques
General Terms: Design
Keywords: Continuous analysis, Codebase Replication, Solstice

1. INTRODUCTION

Many program analysis tools exist that can inform developers about the code they write. The sooner the developer learns the implications of a code change, the more useful the feedback is; a delay between making the change and learning the analysis result can lead to wasted effort or confusion [1, 17, 24]. Ideally, the developer learns the implications of a change as soon as the change is made. For example, modern IDEs, such as Eclipse and Visual Studio, continuously compile the code and inform developers about a compilation error as soon as the developer causes one. Similarly, continuous testing informs the developer as soon as possible after a change breaks a test [24], speculative conflict detection informs developers of conflicts soon after they commit their changes locally [5], and speculative quick fix informs developers of the implications of making compilation-error-fixing changes before the change even takes place [19].

This paper introduces Codebase Replication, an approach that enables a tool builder to easily convert an offline analysis that developers must run manually, into a continuous analysis that runs constantly and always provides the developer with up-to-date results. Our goal is not to make an analysis run faster nor incrementally, but to run it more frequently and to simplify the developer's workflow — all without requiring a redesign of the analysis tool. This allows developers to rely on the continuous analysis tool to notify the developer if, and when, something relevant happens, instead of the developer having to interrupt their work to run the analyses and examine the results. These notifications may display analysis results, indicate that potentially interesting information is available, or cue other relevant analyses or tools to run.

Making an analysis continuous — executing quickly after each code change to identify problems in a timely manner — is challenging. This explains why few continuous analyses exist, despite their benefits. Two major challenges are *isolation* and *currency*. Isolation requires that the analysis should not prevent the developer from making new changes, and that code changes made by an impure analysis should alter the code neither visibly nor functionally while the developer is working. Currency requires that an analysis is optionally restarted and old results are marked as stale when the developer makes an edit, and that analysis results are made available as soon as the analysis completes.

We have addressed these challenges with Codebase Replication, an approach that enables conversion of offline analyses — even impure ones — into continuous analyses that inform the developer of the implications of recent changes as quickly as possible after those changes are made. Codebase Replication employs four principles to overcome the challenges of isolation and currency: *replication* — keeping a separate, in sync copy of the program on which the analysis executes, *buffer-level synchronization* — including the latest changes in the analysis, *exclusive ownership* — allowing analyses to request exclusive write access to the program under analysis, and *invalidation detection* — identifying results made stale by new developer changes. We demonstrate Codebase Replication by converting three previously offline analyses into continuous-analysis plug-ins

ESEC/FSE '13, August 18–26, 2013, Saint Petersburg, Russia

for Eclipse. Each plug-in uses the offline analysis internally.

The key idea underlying Codebase Replication is to create and maintain a 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 can start quickly because the copy codebase is always in sync; it does not need to make a fresh copy of the code before executing. If the developer edits the code while the analysis is running, Codebase Replication allows the analysis to: terminate and restart the offline analysis, so that the results are always accurate at the moment they are delivered; defer propagating the developer's edits 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 invalidate parts of the results.

The main contributions of our work are:

- The identification of *isolation* and *currency* as challenges to the creation of continuous analyses.
- Codebase Replication: An approach to using a copy codebase to meet the challenges with little overhead (~2.5 milliseconds per developer action).
- Solstice, an open-source, publicly-available implementation of Codebase Replication within Eclipse.
- Three new, publicly-available continuous-analysis Eclipse plug-ins, each based on an existing analysis: FindBugs [14], PMD [22], and unit testing.
- An evaluation of the effort needed to create these plug-ins, demonstrating that Solstice, and therefore Codebase Replication, significantly reduces the programmer effort needed to develop continuous-analysis Eclipse plug-ins.
- A preliminary case study, in which one of the authors used a Solstice-based continuous testing plug-in on a small program (7 KLoC), showing that the plug-in is fast, stable, and useful.

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 [1, 17, 24], or less frequently to avoid distracting and annoying the developer [2]. Codebase Replication supports both.

The rest of this paper is structured as follows. Section 2 formally defines continuous analysis. Section 3 introduces Codebase Replication and explains how it addresses the challenges of isolation and currency for continuous analysis implementation. Section 4 introduces Solstice, a Codebase Replication implementation for Eclipse. Section 5 evaluates Solstice's performance and our experience implementing three proof-of-concept continuous analyses with Solstice. Section 6 places our work in the context of related research. Finally, Section 7 summarizes our contributions.

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 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 user input. A *continuous analysis* is one that automatically computes an up-to-date result without user intervention. Finally, a *pure analysis* is one that does not modify the snapshot on which it runs, while an *impure analysis* does. More formally:

Definition 1 (Snapshot)**.** A *snapshot* is a single developer's view of a program at a point in time, including the latest file contents from unsaved editor buffers, if any, or from the disk otherwise.

Each of a developer's changes (e.g., made via an IDE) creates a new snapshot.

In this paper, we limit ourselves to considering analyses that run on a single developer's codebase. Some analyses, e.g., conflict detection [5, 6], may need multiple developers' codebases of the same program. Our work is applicable to such analyses as well, although the definition of a snapshot would need to be modified.

Definition 2 (Analysis)**.** An *analysis* is a function $A\colon S \to R$, 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.

As an 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 configuration parameters or the location of a resource.

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

Let $T_{I_A}(s)$ be the time it takes an analysis implementation I_A to compute A on a snapshot s.

It is our goal to convert an offline analysis implementation I_A into a continuous analysis implementation I_A^c that executes I_A internally.

A user may edit the program faster than an analysis implementation is able to complete — that is, faster than $T_{I_A}(s)$. In this case, I_A^c should be responsive to changes in the program and the completion of the analysis in two ways: computing the analysis on the latest snapshot, and notifying the developer that previous analysis results are stale.

There are multiple policy choices for what I_A^c may do in terms of computing the analysis on the latest snapshot. For example, I_A^c may terminate I_A, since it is running on a stale snapshot, and restart it on the current snapshot. Or I_A^c might permit I_A to complete, to deliver the results, even if somewhat stale. Other options include restarting I_A only during a sufficiently-long pause in user edits, or computing whether a specific user edit might affect the results (ignoring edits to comments, for example).

Similarly, there are multiple policies I_A^c could follow in notifying the developer about possibly outdated results. For example, I_A^c could remove the possibly-stale results from developer's view, or merely mark them as stale. When an analysis completes the computation of a new result, I_A^c could display those results immediately, or could delay to reduce churn and give users a chance to save work (such as marking analysis results as inspected).

For simplicity of presentation, our definition of a continuous analysis assumes the most eager policies. It could be extended to accommodate other policy choices.

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* I_A^c that uses I_A is ε-*continuous* if $\exists \varepsilon_a, \varepsilon_i \leq \varepsilon$ such that for all snapshots s, both of the following are true:

1. I_A^c makes $r = A(s)$ available no later than $t_s + T_{I_A}(s) + \varepsilon_a$ if no new snapshot is created before this time.

   ε_a is the analysis delay: the time it takes to terminate an ongoing analysis (execution of I_A), apply any pending edits to the copy codebase, restart the analysis, and deliver the results (e.g., to a UI or a downstream analysis).

2. For all times after $t_s + \varepsilon_i$, I_A^c indicates that all results for s and snapshots prior to s are stale.

   ε_i is the invalidation delay: the time it takes to invalidate the displayed results after the moment they become stale.

We often refer to ε-continuous analyses as simply *continuous*, implying that appropriately small ε_a and ε_i exist.

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 analysis implementation)**.** An analysis implementation I_A 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.

3. CODEBASE REPLICATION

Codebase Replication converts an offline analysis I_A into a continuous analysis I_A^c while addressing the two major challenges in creating continuous analysis tools: isolation and currency. The developer should be isolated from I_A^c: I_A^c should neither block the developer nor change the code as the developer is editing it (which could happen for an impure I_A). Additionally, I_A^c should be isolated from the developer: Developer edits should not alter the snapshot in the middle of an I_A execution, potentially affecting the results. Despite isolation between the developer and I_A^c, currency requires I_A^c to react quickly to developer edits and to I_A results. Whenever the developer makes an edit, I_A^c should be notified so that it can mark old results as stale, terminate and restart I_A, or take other actions. I_A^c should react to fine-grained changes in the developer's buffer, without waiting until the developer saves the changes to the file system or commits them to a repository. I_A^c should also react quickly to I_A results, making them promptly but unobtrusively available to the developer or to a downstream analysis.

Most IDEs have a unique UI thread, so simply executing I_A in a continuous loop would block the developer from interacting with the IDE whenever I_A runs. Unless I_A is nearly instantaneous, this would render the IDE unusable. Running I_A in a separate thread requires synchronizing with the UI thread, which may interact with the code and may still block the developer. As a consequence, and to our best knowledge, none of the existing continuous analysis tools follow this approach, and neither does Codebase Replication.

Codebase Replication addresses the isolation challenge of converting I_A into I_A^c by creating, and maintaining, a copy of the developer's codebase. The goal of this copy codebase is to provide a stable snapshot for I_A to use without affecting the developer, and without being affected by the developer. Codebase Replication addresses the currency challenge by providing notifications for events that occur in the developer's IDE and in I_A^c; these events can trigger terminating and restarting I_A and updating the UI.

Figure 1 shows Codebase Replication's high-level overview. The IDE API generates events for all developer actions, including developer's changes to the code. Codebase Replication keeps a queue of these events, and applies them to the copy codebase. Meanwhile, I_A^c can pause the queue, run I_A on the copy snapshot, compute I_A results, resume the queue, and modify the developer's editor UI based on the results. Codebase Replication also notifies I_A^c about new developer actions, so that I_A^c may decide to interrupt, alter, or continue I_A's execution.

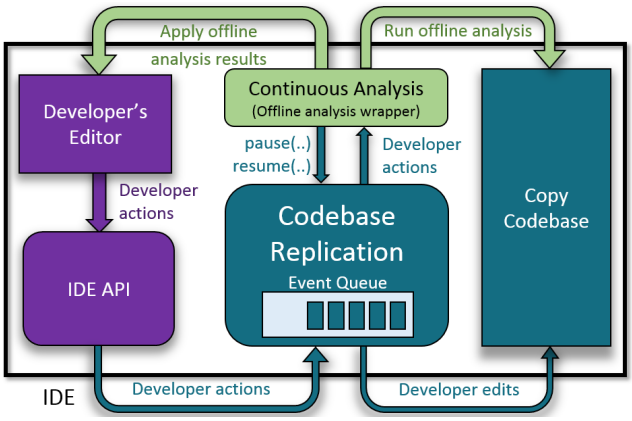

Figure 1: A high-level Codebase Replication overview. Codebase Replication (turquoise) facilitates communication between I_A^c (green) and a developer's IDE (purple) via asynchronous events.

Codebase Replication supports multiple I_A^c using the same copy codebase, managing locks and parallel execution. However, in this paper, for exposition, we focus on the simpler case of a single I_A^c running at a time.

Codebase Replication employs four principles to overcome the challenges of isolation and currency:

1. *Replication:* Instead of running I_A directly on the developer's snapshot, Codebase Replication provides a separate copy of the snapshot by maintaining an in-sync copy of the codebase. This ensures that the developer's codebase is never affected by I_A. This principle is based on the idea of sandboxing. If I_A modifies the copy snapshot, or crashes in the middle of the execution, the effects are confined to the copy, and the developer may continue to edit unaffected by the modifications and crashes. Further, this approach separates the UI logic from the analysis logic, which is a good design principle that improves the tool's maintainability.

2. *Buffer-level synchronization:* One of the goals of I_A^c is to provide feedback on the latest possible snapshot. This snapshot often resides in the IDE buffer, and not on disk. Unfortunately, most existing I_A execute on files or even on binaries. Codebase Replication synchronizes the copy snapshot with the developer's IDE's unsaved buffers, allowing I_A to run on the latest snapshot, or binary compiled from that snapshot, without without affecting the developer's buffer, files, and binaries.

Some I_A are incremental, computing results based on code changes since the last snapshot, as opposed to on an entire snapshot. For most I_A, creating an incremental version presents a significant obstacle, and amounts to developing a brand new I_A. Codebase Replication's goal is to minimize the cost of building I_A^c by allowing developers to create continuous tools by using existing I_A implementations without modifying or redesigning those implementations. While easing the process of building I_A^c (Section 5.2 will empirically evaluate the effort required to develop I_A^c using a prototype implementation of Codebase Replication), we should note that one disadvantage of Codebase Replication over developing incremental I_A is the extra runtime overhead incurred by listening for buffer changes, identifying and discarding duplicated file system changes, and detecting discarded buffers. Meanwhile, the incremental approach needs to only listen for file system changes. Nevertheless, Section 5.1 will show that for our prototype Codebase Replication implementation, this overhead is no more than 2.5 milliseconds per developer action.

3. *Exclusive ownership:* Most I_A assume the snapshot does not change while they execute. Thus, changing the snapshot in the

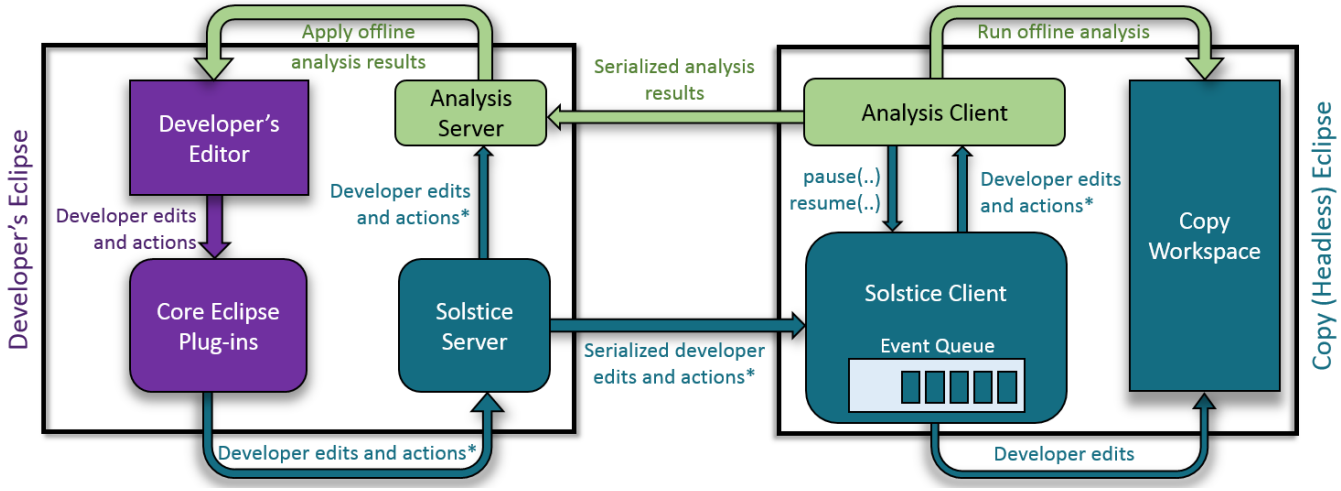

Figure 2: High-level Solstice overview. Eclipse (purple) generates events for developer actions, including edits. `Solstice Server` (turquoise) listens for these events, serializes them and sends them to `Solstice Client` (turquoise), and notifies I_A^c of the actions. `Solstice Client` stores these actions temporarily in the event queue, deserializes them, applies the edits to the copy workspace, notifies I_A^c of these actions, and provides pause-resume API for managing the exclusive ownership of the copy workspace. An example I_A^c (green) interacts with the developer's editor, Solstice, and the copy workspace by: `Analysis Client` runs I_A on the copy workspace, generates and serializes the results, and sends those results to `Analysis Server`. `Analysis Server` deserializes the results, modifies developer's editor accordingly, and implements invalidation logic.

middle of an execution may cause I_A to crash or fail in other ways. The situation is worse if multiple I_A run on the same code at once, and at least one of them is impure. Codebase Replication allows one I_A^c at a time to claim exclusive write access to a copy snapshot while its I_A executes, pausing all other analyses and synchronization updates with the developer's buffer. This allows I_A to complete with expected behavior.

4. *Invalidation detection:* Developer's changes may invalidate the I_A results on an old snapshot. If the change occurs while I_A executes, the result may already be invalid by the time I_A completes. Codebase Replication allows I_A^c to terminate I_A, or to let if finish and to examine the change to determine whether the results, or parts of the results are valid.

These principles allow Codebase Replication to convert I_A into I_A^c while ensuring isolation and currency. Multiple I_A^c can use a single copy snapshot, amortizing the already-low overhead (see Section 5.1). Multiple impure I_A^c can each take turns exclusively owning the copy snapshot to provide fast, accurate results with buffer-change-level precision. The changes the impure I_A make to the copy snapshot remain completely hidden from the developer's copy. (Codebase Replication does require I_A^c using impure I_A to implement a mechanism to revert the I_A changes, and the correctness of the copy synchronization is only guaranteed if the reverting works properly. However, most impure I_A already implement this logic. For those that do not, since the copy codebase *is* saved on the file system, I_A^c can implement a simple revert mechanism using version control. Embedding such a mechanism in Codebase Replication is future work.)

Finally, we impose three performance requirements on Codebase Replication: First, Codebase Replication must have nearly negligible computational overhead in maintaining the copy codebase. I_A^c is already computation intensive, and Codebase Replication should not consume more of that resource. Second, the copy codebase should be synchronized with the developer's codebase as quickly as possible, to facilitate I_A executing on the most up-to-date version of the code. Third, Codebase Replication should be easy to use to create I_A^c via a comprehensive API. Section 5 will revisit how well our implementation satisfies these requirements.

4. SOLSTICE: CODEBASE REPLICATION FOR ECLIPSE

To evaluate Codebase Replication, we have built Solstice, an Eclipse-based, open-source prototype. Solstice is publicly-available at https://bitbucket.org/kivancmuslu/solstice. This section describes Solstice (Section 4.1), explains how to implement continuous analysis tools using Solstice (Section 4.2), and then demonstrates one such implementation with an example (Section 4.3). Later, Section 5.2 will describe our experience using Solstice to develop three continuous analysis tools.

4.1 Solstice Implementation

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

Figure 2 details Solstice's architecture, and an example Solstice-based continuous analysis tool. Solstice consists of two main parts: `Solstice Server` runs on developer's Eclipse and is responsible for listening to the developer actions. `Solstice Client` runs on a background copy of Eclipse 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 instance of Eclipse to be associated with (and have access to) only one workspace. Solstice manages two copies of Eclipse running at once: The developer's normal copy of Eclipse manages the developer's workspace (`Solstice Server` deals with this workspace), and a second, background copy of Eclipse 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 also resides in a hidden folder on disk. That folder name

includes a hash of the absolute path to the developer's workspace, to guarantee uniqueness, with high probability.

Each time the developer starts Eclipse, Solstice executes an *initialization protocol* that blocks the developer and makes sure that the copy workspace is in sync with the developer's workspace before the development begins. First, `Solstice Client` connects to `Solstice Server` through localhost. Then, the *initialization synchronization protocol* executes. The first time the developer uses Solstice, initialization synchronization protocol acts as a full synchronization and creates a complete copy of the developer's workspace on disk. The future executions, on the other hand, only verify the integrity of the files in the copy workspace through checksums, and update the files that are added, removed, or changed in the developer's workspace outside the IDE.

After the initialization synchronization protocol, `Solstice Server` attaches multiple listeners (see Figure 2) to the developer's Eclipse to keep track of all developer's events: actions and edits to the source code. Currently, in addition to all edits developers make to the code, Solstice listens to the following developer actions: current cursor location, current selected file, current selected Eclipse project, invocations of Quick Fix, the proposals offered for a Quick Fix invocation, and selected Quick Fix proposals and completions and cancellations of Quick Fix. `Solstice Server` catches the developer's events, serializes them, and sends them to the `Solstice Client`, which, in turn, deserializes the incoming events, makes them available to the continuous analysis tool through simplified observer patterns, and applies developer edits to the copy workspace. `Solstice Client` keeps track of all developer edits — through the events sent by `Solstice Server`— and applies them to the copy workspace in the order they are generated, guaranteeing that the developer's workspace will always be in sync with the copy workspace, with the exception of the events currently being processed.

Instead of the localhost connection, Solstice could use either the Java Remote Method Invocation (RMI)[1] or the file system for `Solstice Server` and `Solstice Client` to communicate. RMI abstracts away the underlying networking protocol, which can allow for a cleaner design. Meanwhile, using the file system enables higher flexibility for the serialization and communication protocols, but requires these inter-process protocols to be implemented from scratch. Solstice (see architecture in Figure 2) has two requirements for inter-process communication: (1) the ability to send messages (events) between the components running on different Eclipses, and (2) the flexibility of pausing and resuming event processing (for exclusive ownership) within a component. Localhost connection abstracts the underlying inter-process communication protocol, and gives Solstice the flexibility to implement the event-based execution — with the ability to pause and resume the event queue — immediately over the Java networking layer. As a result, Solstice and the continuous analysis tool can completely abstract away the underlying event-based execution, leading to a cleaner logic. We chose to use the localhost connection mechanism because of the combination of abstraction and flexibility it provides.

4.2 Building Solstice-Based Tools

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

To implement I_A^c, the author has to accomplish three tasks: separate the I_A^c computation and interaction logic, formalize the I_A^c communication, and write the result-invalidation logic.

[1] http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html

First, the author needs to separate I_A^c computation logic — how I_A runs and produces results — and interaction logic — how I_A^c shows results and interacts with the developer. The computation logic is implemented as a `Solstice Client` extension, represented as `Analysis Client` in Figure 2. The interaction logic is implemented as a `Solstice Server` extension, represented as `Analysis Server` in Figure 2. The same way Eclipse manages the life-cycle of its extending plug-ins, Solstice manages the life-cycle of the extending I_A^c: they start after Solstice starts (when the developer opens Eclipse) and terminate before Solstice terminates (when the developer closes Eclipse). The author does not need to create and manage a thread for I_A^c as Solstice takes care of these low-level details. The computation logic always runs on the background Eclipse and on the copy workspace, and the interaction logic always runs on the developer's Eclipse and workspace. For the rest of the section, we assume that I_A^c interacts with the developer. Continuous analysis tools that do not interact with the developer (e.g., observational I_A^c that only log developer actions), do not need a `Solstice Server` component as `Solstice Client` duplicates all developer edits and I_A^c can access all those events directly from `Solstice Client` via listeners.

Next, the author needs to formalize 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 Figure 2. However, the communication does not have to be one-sided (although the example communication shown in Figure 2 is one-sided). For example, the `Analysis Server` can allow the developer to modify I_A^c settings, which it would then send to the `Analysis Client`. As the `Analysis Client` and the `Analysis Server` run on different Eclipses, all communication between them occurs through serializable objects. Solstice provides wrappers that reduce the inter-process communication code. The author needs to implement the analysis logic, via the `runAnalysis(..)` method that returns a serialized `AnalysisResult` object, which is then transparently sent to `Analysis Server`. The sent object arrives as an event, received via the `processAnalysisResult(..)` method.

Finally, the author needs to write the logic for invalidating an I_A result based on developer edits. Solstice timestamps every developer action and edit, I_A start, and I_A finish, to ensure that no event is lost and that Solstice knows exactly to which snapshot an I_A result applies. Using these timestamps transparently, `Analysis Client` may either kill the ongoing I_A computation and start a new one — after applying the developer edits to the copy workspace — or allow the current I_A computation finish, possibly producing stale results. The `Analysis Server` may erase the current I_A results or mark them as stale. For common scenarios, such as clearing I_A results with each developer edit, Solstice provides built-in APIs that reduce the code that needs to be written for keeping track of developer changes. For example, if I_A^c wants to immediately invalidate I_A results after all source code changes, that I_A^c needs to only implement the `EditInvalidator.invalidate(..)` method. In contrast, without Solstice, to get the same outcome through pure Eclipse API, I_A^c would need to implement one `IResourceChangeListener` to keep track of the resources being added and deleted, one `IDocumentListener` for the current editor file to keep track of the buffer changes, and one `IPartListener` to keep track of the current editor when using Eclipse API. At the very least, a complete implementation without Solstice would require implementing nine methods. Finally, the implementation needs to process these events and decide whether the current event invalidates the results. For example, each keystroke generates a `DocumentEvent`, which should invalidate the results. However, when the developer saves the current file, it generates a `ResourceChangedEvent`, which needs to be

ignored as it is already subsumed by the previous `DocumentEvents`. Solstice takes care of all these low-level details.

4.3 Example Solstice Usage

Suppose an author wants to use Solstice to build an I_A^c using an I_A. For simplicity, the author decides that when the developer makes a change while I_A executes, I_A might as well finish, but I_A^c invalidates stale results and does not display them to the developer.

The author would have to write the following interaction logic for I_A^c (`Analysis Server`):

```
class Server extends AnalysisServer {
  public Server() {
    super(true);
  }
  void processAnalysisResult(AnalysisResult r) {
     // Update results.
  }
}
// Invalidation logic
class Invalidator extends EditInvalidator {
  void invalidate() {
    // Invalidate results.
  }
}
```

`Server` passes `true` to `AnalysisServer`, which represents that I_A^c only wants to process recent I_A results. Solstice pre-processes all I_A results received from `Analysis Client`, compares their timestamps with the current timestamp, and only propagates the recent results to `processAnalysisResult(..)`.

The author would also have to write the following computation logic for I_A^c (`Analysis Client`):

```
class Analysis extends ResourceBasedAnalysis {
  AnalysisResult runAnalysis() {
    AnalysisResult result = // compute the result.
    return result;
  }
}
```

`Analysis` extends `ResourceBasedAnalysis`, which tells Solstice to call `resumeAnalysis(..)` each time it processes an event that modifies the copy workspace. The `resumeAnalysis(..)` call is a no-op while I_A is running. However, when I_A is paused, invoking `resumeAnalysis(..)` wakes up the I_A^c thread, which eventually calls `Solstice.pause(..)`, `runAnalysis(..)`, and `Solstice.resume(..)`. The author does not need to do anything special to manage the exclusive ownership of the copy; Solstice manages that transparently.

5. EVALUATION

To evaluate Solstice, we empirically measured its performance overhead (Section 5.1), observed the ease of using Solstice while implementing three proof-of-concept continuous analysis tools (Section 5.2), and compared Solstice to other methods of implementing IDE-integrated continuous analyses (Section 5.3).

5.1 Solstice Performance Evaluation

To be effective, Solstice has to meet the following requirements:

R1: The start up time should not block the developer for an unreasonable amount of time.

R2: The overhead to synchronizing developer actions with the copy codebase should be close to zero.

Workspace		Codebase		Sync Time (ms)	
Content	Size (MB)	LoC	KNCSL	Full	Inc.
CrosswordSage	1.3	3.6	3.1	131	133
Asmx	11	43	26	611	292
Voldemort	28.4	160	115	1,809	656
JDT	176	1,890	1,362	2.9 m	6.9 s

Figure 3: Solstice initial synchronization protocol performance. Each cell is the mean of 20 experimental executions. Full synchronization starts with an empty copy workspace, whereas incremental synchronization starts with an in-sync copy codebase, and Solstice only verifies the copies are in-sync.

R3: While using the IDE, the developer should experience negligible delay in performing actions.

This section presents the results of Solstice performance experiments addressing these three requirements. We ran all experiments on a modern laptop (MacBook Pro, i7 2.3 GHz quad core, 16 GB RAM, SSD); Solstice ran with 512 MB RAM limitation for each of the server and client applications.

5.1.1 *Initial Synchronization Protocol Cost*

As Section 3 explained, when the IDE starts, Solstice needs to either install file-system-level resource listeners, or go through an initial synchronization protocol to make sure that the copy codebase and the developer's codebase are in sync. Without this precaution, if the developer (or some other application) modifies the codebase when the IDE is not running, the copy codebase would become out-of-sync. Solstice makes the latter choice (recall Section 4). To satisfy requirement R1, the blocking incurred by the initial synchronization protocol must be reasonable.

We have tested Solstice's initial synchronization protocol on four different workspace settings (Figure 3). For each setting, we created a workspace with one program (a program may consist of many Eclipse projects; for example, JDT consists of 50), and invoked the initial synchronization protocol for two extreme cases: full and incremental. In the full case, the copy workspace is empty, which requires Solstice to copy the entire workspace. In the incremental case, the copy workspace is already in-sync, which requires Solstice to only verify that the copies are in-sync with checksums. We expect the most common use case to resemble the incremental case, with the full case only applying on the first time the developer runs Solstice. Figure 3 shows that Solstice has negligible overhead for small programs: CrosswordSage [8], asmx[2], and Voldemort [32].

Even for large programs, such as JDT, Solstice's incremental synchronization overhead is several (6.9) seconds, which we believe to be reasonable, considering JDT[3] has ~10K files. However, for even larger programs, the incremental synchronization may become an issue. It is possible to make the initial synchronization protocol lazy, only processing the active Eclipse project and its dependencies. For industrial products composed of many Eclipse projects, this approach would block the developer for only a few seconds each time the developer changes which project is active, and only once for each project per IDE restart. Consequently, amortizing the incremental synchronization cost for the workspace to independent, smaller components would improve the developer experience. We leave this improvement to future work.

5.1.2 *Synchronization Overhead*

[2]Extended asm, a byte code manipulator: http://asm.ow2.org

[3]Our JDT evaluation includes four products: eclipse.jdt.core, eclipse.jdt.debug, eclipse.jdt, and eclipse.jdt.ui

Operation Name	Size	Initial File Size (chars)	IDE Overhead (ms)	Sync Delay (ms)
Text Insert	1	0	1.0	1.5
		100	1.1	1.8
		1,000	1.1	1.7
		10,000	2.4	1.9
	100	0	1.2	1.7
		100	1.0	2.0
		1,000	1.1	2.1
		10,000	2.3	2.5
Text Delete	1	1	0.8	1.5
		101	1.1	1.8
		1,001	1.2	1.6
		10,001	2.5	1.7
	100	1	0.8	1.6
		101	1.1	1.9
		1,001	1.1	2.1
		10,001	2.3	2.4
Text Edit	1	100	1.0	1.7
		1,000	1.0	1.9
		10,000	2.2	2.2
	100	100	0.9	1.9
		1,000	1.0	1.9
		10,000	2.2	2.2
Text Edit Summary			≤ 2.5	≤ 2.5
File Add	1	1,000	1.2	1.1
	100		102	157
	1,000		1,464	1,305
File Remove	1	1,000	0.5	1.4
	100		56	106
	1,000		566	2,491
File Edit Summary			grows linearly with size	

Figure 4: The Solstice-induced overhead on developer edits for keeping the copy workspace in-sync. Text operations of size 1 are single keystrokes, and text operations of size 100 represent cut, paste, and tool applications, such as applying a refactoring. File operations of size 1 are manual file generation, copy, and removal, and file operations of size 100 and 1,000 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 to be up-to-date. Text operations are means over 20,000 executions. File operations are means over 200 executions.

To satisfy requirement R2, and to allow I_A^c to access the most recent version of the developer's code, Solstice must quickly synchronize the copy workspace. Figure 4 shows the delay Solstice incurs during synchronization for the most common developer operations. Except for importing and deleting large Eclipse projects, synchronization takes no more than 2.5 milliseconds. Thus, Solstice provides I_A^c access to the developer's code that is no more than 2.5 milliseconds old. Since importing and deleting large Eclipse projects is fairly rare, and the operations already take several seconds for Eclipse to execute, the Solstice delay should be acceptable.

5.1.3 Developer Action Delay

As Solstice keeps track of all developer changes at the buffer level, it listens to each keystroke. To satisfy requirement R3, it is important that the overhead and delay experienced by the developer are close to zero so the developer is not adversely affected by Solstice. Figure 4 shows, for the most common developer actions, the IDE overhead that Solstice introduces when the action is initiated programmatically. One of the most common developer actions is editing a file with keystrokes. Figure 4 represents such text edits as size 1. The IDEs also support complex operations, such as refactoring, auto-complete, etc. that execute multiple edits at once. Figure 4 represents such text edits as size 100. The results show that the overhead is independent of the edit size, and even for large files (10,000 characters), the overhead is no more than 2.5 milliseconds.

Manually adding and removing files to an Eclipse project are represented as 1-, 100-, and 1,000-sized file operations. Results suggest that the overhead for file operations increase linearly with operation size, as expected. Removals are faster than additions, and even for large operations, the overhead never exceeds a few seconds. Since Eclipse already takes several seconds to import a project with 1,000 files, the results suggest that Solstice introduces negligible IDE overhead.

Our performance analysis demonstrates that Solstice introduces negligible overhead to the IDE, almost never interrupts the development process — except during the initial synchronization protocol — and provides access to a recent copy codebase with negligible delay.

5.2 Solstice Usability Evaluation

We have used Solstice to build three proof-of-concept I_A^c Eclipse plug-ins, each using one of three existing I_A implementations. This section describes these implementations and reports on our experience. Sections 5.2.1, 5.2.2, and 5.2.3 describe the continuous FindBugs, the continuous PMD, and the continuous testing plug-ins, respectively. Figure 5 summarizes the implementation and evaluation details of each plug-in.

5.2.1 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 [14]. It is available as a command-line and a GUI tool, an ant task extension, and an Eclipse plug-in [11].

The FindBugs ant task extension and Eclipse plug-in can automate FindBugs invocations but both fall short of being ε-continuous (recall Definition 5 in Section 2): The ant task extension executes only with each ant build. Meanwhile the plug-in has a setting to run whenever Eclipse's incremental build runs, which typically happens when the developer saves the file, and analyzes only that saved file. Both tools require the developer to perform an action to run, either execute the build script or save a file, and neither reacts to changes made only to the buffer. Further, since changes to one file may affect the analysis results of another, a Solstice-based implementation may find warnings missed by the FindBugs plug-in.

We have used Solstice to build a proof-of-concept, continuous FindBugs Eclipse plug-in [29]. The plug-in uses the command-line FindBugs to analyze the .class files for all the classes in the currently active Eclipse project, and of all the libraries on which they depend. The plug-in's simple visualization displays the FindBugs warnings in an Eclipse view[4], which is a configurable window similar to the Eclipse Console. The plug-in invalidates the warnings whenever the developer makes an edit or makes another project active. Figure 6 shows two continuous FindBugs plug-in screenshots, one before (left) a developer makes a change that removes a FindBugs warning, and one (right) after.

[4]http://help.eclipse.org/topic/org.eclipse.platform.doc.isv/reference/extension-points/org_eclipse_ui_views.html

Analysis	Lines of Code					Dev.	Evaluation Subject Program				Analysis	ε_a	ε_r
	UI	IPC	Core	Other	Total	Time	Name	Version	KLoC	KNCSL	Runtime	(ms)	(ms)
FindBugs	196	39	138	16	389	25 hours	Voldemort	1.1.2	160	115	58.6 s	59	2
PMD	226	37	110	18	391	4 hours					8.7 s	67	2
Testing	297	136	327	18	778	25 hours	Commons CLI	1.3	10.5	5.8	294 ms	56	3

Figure 5: Summary of three Solstice-based proof-of-concept tools. Each tool consists of "UI" code for result visualization and pretty printing, "IPC" code for serialization, "Core" code for setting up and running the I_A, and "Other" code for extension points. PMD I_A use is similar to FindBugs, and PMD I_A^c was implemented after FindBugs I_A^c, which led to a significant reduction in development time. In calculating ε values, we used PMD's `java-basic` ruleset and all of Commons CLI's 361 tests. Each ε value is the max observed value over at least 10 executions.

Figure 5 summarizes the implementation and evaluation details for the continuous FindBugs plug-in. The implementation is fewer than 200 LoC, excluding the code for visualization. It is also ε-continuous, with empirically computed values: $\varepsilon_a = 59$ milliseconds and $\varepsilon_i = 2$ milliseconds.

5.2.2 *Continuous PMD*

PMD [22] 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, and applies different kinds of analyses. 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, continuous PMD Eclipse plug-in [30]. The plug-in uses the command-line PMD to analyze the `.java` files for the currently active Eclipse project. The plug-in's simple visualization displays the PMD results in an Eclipse view. The plug-in invalidates the results whenever the developer makes an edit or makes another project active. Figure 7 shows two continuous PMD plug-in screenshots, one before (left) a developer makes a change that removes a PMD warning, and one (right) after.

Figure 5 summarizes the implementation and evaluation details for the continuous PMD plug-in. The implementation is fewer than 200 LoC, excluding the code for visualization. It is also ε-continuous, with empirically computed values: $\varepsilon_a = 67$ milliseconds and $\varepsilon_i = 2$ milliseconds.

5.2.3 *Continuous Testing*

Continuous testing 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 up to 15% [24]. There are Eclipse [27] and Emacs plug-ins for continuous testing. The Eclipse plug-in modifies Eclipse core plug-ins, making it difficult to update the implementation for new Eclipse releases; in fact, the plug-in does not support the 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, continuous testing Eclipse plug-in [31]. The plug-in runs the tests of the currently active Eclipse project, and invalidates the test results whenever the developer makes an edit. The plug-in's simple visualization displays the test results in an Eclipse view. Figure 8 shows two continuous testing plug-in screenshots, one before (left) a developer makes a change that breaks a test, and one (right) after.

Figure 5 summarizes the implementation and evaluation details for the continuous testing plug-in. The implementation is fewer than 500 LoC, excluding the code for visualization. It is also ε-continuous, with empirically computed values: $\varepsilon_a = 56$ milliseconds and $\varepsilon_i = 3$ milliseconds.

5.2.4 *Solstice Continuous Testing Case Study*

One of the authors used the Solstice continuous testing plug-in ($\text{Solstice}_{\text{CT}}$) in a personal BibTeX management project, consisting of 7 Java KLoC. Before starting using $\text{Solstice}_{\text{CT}}$, the project was exhibiting `RuntimeException` crashes on a specific input. The case study took place while removing the bug that was resulting in this crash by writing tests and using $\text{Solstice}_{\text{CT}}$. The bug turned out to

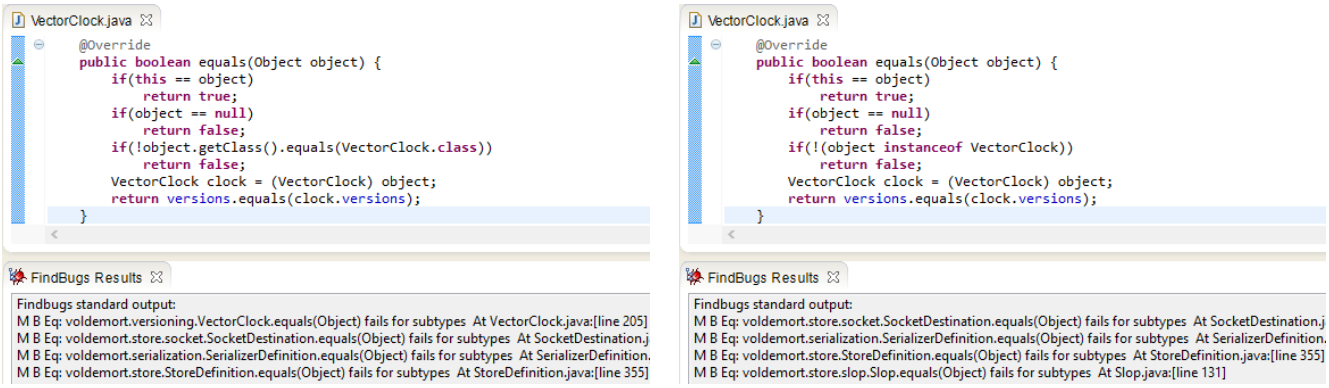

Figure 6: Continuous FindBugs running on Voldemort. Both images show the top four warnings. The left screenshot shows the original Voldemort implementation; its first FindBugs warning suggests that `.equals(..)` is too restrictive. The developer changes `.equals(..)` with `instanceof` (right screenshot)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

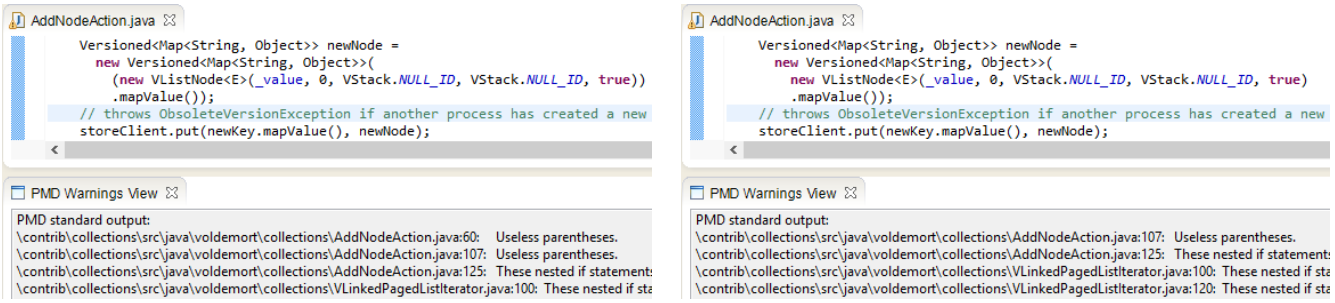

Figure 7: Continuous PMD running on Voldemort. Both images show the top four warnings. The left screenshot shows the original Voldemort implementation; its second PMD warning suggests that the parentheses around `new` are unnecessary. The developer removes these parentheses (right screenshot) and the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

be significant and required an extension to the project's architecture and writing more than 100 LoC. The case study lasted for 3 days and resulted in a successful removal of the bug.

At the start of the case study, 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 case study led to the following 3 observations:

1. **Solstice$_{CT}$ can speed up discovering unknown bugs:** The subject program parsed several files, specified as program arguments. When the specified path did not exist, the program threw a `FileNotFoundException`. This behavior made the program fragile, crashing on invalid or improperly formatted arguments, instead of failing gracefully. This bug was unknown prior to the case study. The author discovered this bug early, right after starting implementing the regression test, by initially entering an invalid path and before encoding the rest of the test. This intermediate step during writing the test caused the regression test to fail with a `FileNotFoundException`. Consequently, the author was able to detect and fix this bug immediately, as opposed to after implementing the entire test. Because the test needed to encode the expected output, and that output was large, Solstice$_{CT}$ caused the bug to be discovered much earlier than it would have been through traditional testing.

2. **Solstice$_{CT}$ makes debugging information available sooner:** While debugging, developers often use print statements to view the intermediate program state and assist in understanding the behavior of faulty executions. Based on self-discovered feedback during this case study, we augmented Solstice$_{CT}$ to make the continuous testing console output and error streams available to the developer. As a result, the developer was able to get 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.

3. **Solstice$_{CT}$ is unobtrusive:** In the course of this, *albeit* small case study, the author never experienced a noticeable slowdown in Eclipse's operation and never observed a stale or wrong test result.

5.3 Discussion

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, because of its speed, the developer would never notice the blocking. Of course, having a negligible running time is infeasible for many analyses.

It is possible to reduce the running time of offline analyses by making them incremental [23]. An incremental code analysis uses the analysis result on an earlier snapshot of the code and the edits made since that snapshot. There are many examples for incremental analyses, such as differential static analyses [18], differential symbolic checking [21], and incremental checking of data structure invariants [28]. 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 [17] is one such 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 a greedy, efficient algorithm that solves a problem for which only an inefficient algorithm that requires global information is known. For some such analyses (e.g., analyses that solve *NP*-complete problems), this task is likely impossible (unless $P = NP$).

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 Solstice. An impure analysis is freed from the burden of maintaining a copy codebase manually, or through working copies, as Solstice maintains the copy codebase and lets the analyses own it exclusively. For many analyses, the slightly stale results of recent development snapshots are useful. For example, the test results of a development snapshot from several minutes ago could help the developer realize a test failure that otherwise would not surface until the next scheduled test run. For such, slow analyses, Solstice allows processing recent snapshots and producing results, whereas other approaches would invalidate the result with every edit.

6. RELATED WORK

IDE-integrated continuous analyses are not new. Incremental code compilation [17] has been integrated in most modern IDEs, such as Eclipse[5]. Eclipse Java compiler warnings and errors[6], including the Nullness type system, are closely integrated into the incremental compiler. These continuous analyses are implemented by the IDE developers. For third party analyses, Eclipse provides project builders: An analysis author can implement the analysis logic as a project builder extension so that the analysis is executed with each incremental build. Eclipse FindBugs [11], Checkstyle [9], and Metrics [10] plug-ins are example continuous analyses imple-

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

[6]http://help.eclipse.org/topic/org.eclipse.jdt.doc.user/reference/preferences/java/compiler/ref-preferences-errors-warnings.htm

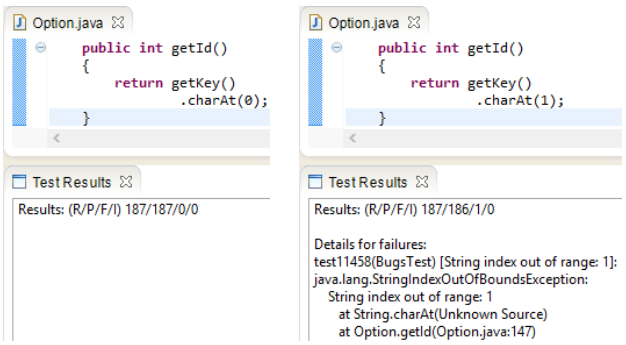

Figure 8: Continuous testing running on Apache commons.cli. The left screenshot shows the original commons.cli implementation, for which all tests pass. The developer defines the id of an option to be its second character (right screenshot) and immediately sees that this change causes an existing test to fail.

mented using project builders. Solstice and its API provides a richer, alternative way to integrate third party analyses into the IDE.

In addition to project builders, Eclipse — and some other IDEs — provide the analysis authors with a *working copy*[7]: the in-memory state of a compilation unit (e.g., a Java file). Working copies are useful for fast, impure analyses that make small changes, such as refactoring, quick fix, and auto complete. The analysis authors can modify a compilation unit without changing the developer's copy. The developer changes to the same compilation unit are reported to the working copy, and the analysis author may reconcile the working copy with the developer's copy. Solstice, in some sense, elevates the working copy concept to the workspace level, with other features, such as being able to access a stable development snapshot.

Speculative analysis [4] is a general concept of an impure analysis that calculates actions a developer is likely to perform soon, performs those actions in the background, evaluates the consequences of those actions, and reports on those consequences. Speculative analysis over version control can identify and prevent collaborative conflicts [5, 6, 12] by creating and managing a copy codebase for the analysis to use. Meanwhile, speculative analysis over Eclipse Quick Fixes [19] creates and maintains its copy codebase inside the developer's workspace. Solstice removes the burden of creating and maintaining the copy codebase, and relying on external operations from such impure analyses. Similarly, live programming focuses on providing continuous feedback to developers for the tasks they are working on [3, 33] by making the behavior of a program visible as that program is being developed [7]. Solstice enables continuous analyses that facilitate live programming practices.

Integrating continuous analyses into the IDE facilitates continuous development. Continuous development is not a new concept; a programming environment, modeled on spreadsheets, can continuously execute the program as it is being developed [13, 16]. Similarly, continuous testing [24, 25, 26] executes the tests available for a program as it is being developed. Meanwhile, continuous data testing applies the same ideas to data debugging: discovering system errors caused by well-formed but incorrect data [20]. While continuous testing is integrated into Eclipse [27] and IntelliJ [15], Solstice provides a generic framework for converting offline analyses into continuous ones, integrated into the IDE.

[7]http://wiki.eclipse.org/FAQ_What_is_a_working_copy%3F

7. CONTRIBUTIONS

Codebase Replication is a novel approach for converting offline analyses that developers must run manually into continuous analyses that run constantly and always provide the developer with up-to-date results. Codebase Replication solves two major challenges of building continuous analyses: *isolation* and *currency*. Isolation requires that the analysis does not prevent the developer from making new changes, and code changes made by an impure analysis neither alter the code visibly nor functionally while the developer is working. Currency requires that analysis results are made available quickly, and that stale results are marked as stale. To solve these challenges, Codebase Replication maintains an in-sync copy of the developer's codebase, allowing multiple analyses to use, and even temporarily exclusively own, the copy. Codebase Replication tracks developer changes at the buffer level, works with analyses that require either source code or compiled binaries to run, and can even be used to convert impure offline analyses into continuous ones.

Solstice, our prototype implementations of Codebase Replication, facilitates converting offline analyses into continuous-analysis Eclipse plug-ins. We have used Solstice to build three such proof-of-concept plug-ins. Each plug-in, on average, required writing only 519 LoC (436 NCSL) and took 18 hours to write. The Solstice-based plug-ins incurred 2.5 milliseconds overhead on the developer's IDE and experienced a 2.5 milliseconds delay before synchronizing the copy codebase with the developer's latest edits.

Overall, the cost to converting offline analyses to continuous ones with Codebase Replication is low, as compared to the alternative of redesigning the offline analyses to work continuously. Further, the benefits of continuous analysis tools greatly outweigh the cost of building them with Codebase Replication. Since continuous analyses reduce the delay between the time when a developer makes a change and when the effects and implications of this change become known, by reducing the cost to implement IDE-integrated continuous analyses, we believe Codebase Replication will bring more continuous analyses into existence. Consequently, developers will no longer need to interrupt development to get analysis feedback. Instead, continuous analysis feedback will be integrated into the software development process.

8. ACKNOWLEDGMENTS

This work is funded by NSF grants CCF-0963757 and CCF-1016701. Ezgi Mercan worked on an early stage of the Solstice prototype implementation. We thank Deepak Azad, Daniel Megert, and Stephan Herrmann for their help with the Eclipse internals throughout Solstice development, and the anonymous reviewers for their comments and feedback.

9. REFERENCES

[1] B. W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[2] C. Boekhoudt. The big bang theory of IDEs. *Queue*, 1(7):74–82, October 2003.

[3] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer. Example-centric programming: Integrating web search into the development environment. In *the 28th Conference on Human Factors in Computing Systems*, CHI'10, pages 513–522, Atlanta, GA, USA, April 2010.

[4] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Speculative analysis: Exploring future states of software. In *the Workshop on the Future of Software Engineering Research*, FoSER'10, pages 59–63, Santa Fe, NM, USA, November 2010.

[5] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Proactive detection of collaboration conflicts. In *the 8th joint meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ESEC/FSE'11, Szeged, Hungary, September 2011.

[6] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Early detection of collaboration conflicts and risks. *IEEE Transactions on Software Engineering*, 2013.

[7] M. M. Burnett, J. W. Atwood Jr., and Z. T. Welch. Implementing level 4 liveness in declarative visual programming languages. In *Symposium on Visual Languages*, VL'98, pages 126–133, September 1998.

[8] Crossword sage. http://sourceforge.net/projects/crosswordsage, 2005.

[9] Eclipse-Checkstyle integration. http://eclipse-cs.sourceforge.net, 2002.

[10] Eclipse Metrics plug-in. http://sourceforge.net/projects/metrics, 2002.

[11] FindBugs. http://findbugs.sourceforge.net, 2003.

[12] M. L. Guimarães and A. R. Silva. Improving early detection of software merge conflicts. In *the 34th International Conference on Software Engineering*, ICSE'12, pages 342–352, Zurich, Switzerland, June 2012.

[13] P. Henderson and M. Weiser. Continuous execution: The VisiProg environment. In *the 8th International Conference on Software Engineering*, ICSE'85, pages 68–74, London, England, August 1985.

[14] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *the 19th Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA'04, pages 132–136, Vancouver, BC, CANADA, October 2004.

[15] Infinitest. http://infinitest.github.io, 2010.

[16] R. R. Karinthi and M. Weiser. Incremental re-execution of programs. In *Symposium on Interpreters and Interpretive Techniques*, SIIT'87, pages 38–44, St. Paul, MN, USA, 1987.

[17] H. Katzan Jr. Batch, conversational, and incremental compilers. In *the American Federation of Information Processing Societies*, AFIPS'69, pages 47–56, Boston, MA, USA, May 1969.

[18] S. K. Lahiri, K. Vaswani, and C. A. R. Hoare. Differential static analysis: Opportunities, applications, and challenges. In *the Workshop on the Future of Software Engineering Research*, FoSER'10, pages 201–204, Santa Fe, NM, USA, November 2010.

[19] K. Muşlu, Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Speculative analysis of integrated development environment recommendations. In *the 3rd Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA'12, pages 669–682, Tucson, AZ, USA, October 2012.

[20] K. Muşlu, Y. Brun, and A. Meliou. Data debugging with continuous testing. In *the 9th joint meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering, new ideas track*, ESEC/FSE'13, Saint Petersburg, Russia, August 2013.

[21] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *the 16th Symposium on the Foundations of Software Engineering*, FSE'08, pages 226–237, Atlanta, GA, USA, November 2008.

[22] PMD. http://pmd.sourceforge.net, 2003.

[23] G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *the 20th Symposium on Principles of Programming Languages*, POPL'93, pages 502–510, Charleston, SC, USA, January 1993.

[24] D. Saff and M. D. Ernst. Reducing wasted development time via continuous testing. In *the 14th International Symposium on Software Reliability Engineering*, ISSRE'03, pages 281–292, Denver, CO, USA, November 2003.

[25] D. Saff and M. D. Ernst. Continuous testing in Eclipse. In *the 2nd Eclipse Technology Exchange Workshop*, eTX'04, Barcelona, Spain, March 2004.

[26] D. Saff and M. D. Ernst. An experimental evaluation of continuous testing during development. In *International Symposium on Software Testing and Analysis*, ISSTA'04, pages 76–85, Boston, MA, USA, July 2004.

[27] D. Saff and M. D. Ernst. Continuous testing in Eclipse. In *the 27th International Conference on Software Engineering*, ICSE'05, pages 668–669, St. Louis, MO, USA, May 2005.

[28] A. Shankar and R. Bodík. Ditto: Automatic incrementalization of data structure invariant checks (in Java). In *Conference on Programming Language Design and Implementation*, PLDI'07, pages 310–319, San Diego, CA, USA, June 2007.

[29] Solstice: Continuous FindBugs. https://bitbucket.org/kivancmuslu/solstice-continuous-findbugs, 2013.

[30] Solstice: Continuous PMD. https://bitbucket.org/kivancmuslu/solstice-continuous-pmd, 2013.

[31] Solstice: Continuous testing. https://bitbucket.org/kivancmuslu/solstice-continuous-testing, 2013.

[32] Voldemort. http://www.project-voldemort.com/voldemort, 2009.

[33] 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 *the Conference on Human Factors in Computing Systems*, CHI'97, pages 258–265, Atlanta, GA, USA, March 1997.