Systematic Black-Box Analysis of Collaborative Web Applications

Marina Billes
Department of Computer Science
TU Darmstadt, Germany
marina.billes@crisp-da.de

Anders Møller
Department of Computer Science
Aarhus University, Denmark
amoeller@cs.au.dk

Michael Pradel
Department of Computer Science
TU Darmstadt, Germany
michael@binaervarianz.de

Abstract

Web applications, such as collaborative editors that allow multiple clients to concurrently interact on a shared resource, are difficult to implement correctly. Existing techniques for analyzing concurrent software do not scale to such complex systems or do not consider multiple interacting clients. This paper presents Simian, the first fully automated technique for systematically analyzing multi-client web applications.

Naively exploring all possible interactions between a set of clients of such applications is practically infeasible. Simian scales to real-world applications by using a two-phase black-box approach. The first phase systematically explores the application with a single client to infer potential conflicts between client events. The second phase synthesizes multi-client interactions targeted at triggering misbehavior that may result from the potential conflicts, and reports an inconsistency if the clients do not converge to a consistent state.

We evaluate the analysis on three widely used systems, Google Docs, Firepad, and ownCloud Documents, where it reports a variety of inconsistencies, such as incorrect formatting and misplaced text fragments. Moreover, we find that the two-phase approach runs 10x faster than exhaustive exploration, making systematic analysis feasible.

CCS Concepts • **Software and its engineering** → **Dynamic analysis**; **Software testing and debugging**

Keywords Testing, collaborative editing, dynamic analysis

1. Introduction

The web platform has greatly expanded its capabilities in recent years with improvements in browser implementations and APIs, which have enabled rich internet applications that perform tasks previously reserved to dedicated desktop applications. This work focuses on *collaborative web applications*, where multiple clients concurrently work on a shared resource, such as a text document, a spreadsheet, or source code. Popular examples of such applications are Google Docs,[1] Microsoft Office Online,[2] and the Cloud9 IDE.[3]

Synchronizing the clients of a collaborative web application is a nontrivial problem. Typically, a snapshot of the shared data is modified locally via a JavaScript-based client-side implementation, while updates are sent to the server and from there pushed out to other clients. The goal of these systems is to ensure that all clients eventually converge to a consistent state. In practice, implementing concurrency control for a collaborative web application is challenging because clients may trigger various UI actions and because these actions may interleave in a multitude of ways. Due to this complexity, collaborative web applications are particularly error-prone concurrent applications, while finding errors is difficult because of the enormous space of possible interactions between clients.

As a simple real-world example, consider a document in Google Docs with the text “**testing**” in a single line, as shown in Figure 1. Suppose one client writes “ this” at the end of the line, while another client concurrently selects and deletes the existing text. After the clients synchronize their actions via the server, the first client shows “ **this**” in bold font, whereas the second client shows the text without using bold font. The clients now have inconsistent states that do not converge, which is clearly a bug.

Unfortunately, state-of-the-art analysis techniques fail to detect such bugs. Existing approaches applicable to collaborative web applications include server-side load testing, which does not test the client UI implementation, and UI testing on a client, e.g., using Selenium,[4] which is geared toward single-client scenarios. As a result, current techniques easily miss bugs that arise from the interactions of clients and their synchronization via a server.

PLDI'17, June 18–23, 2017, Barcelona, Spain
ACM. 978-1-4503-4988-8/17/06...$15.00
http://dx.doi.org/10.1145/3062341.3062364

[1] http://docs.google.com/

[2] https://www.office.com/

[3] https://c9.io/

[4] http://www.seleniumhq.org/

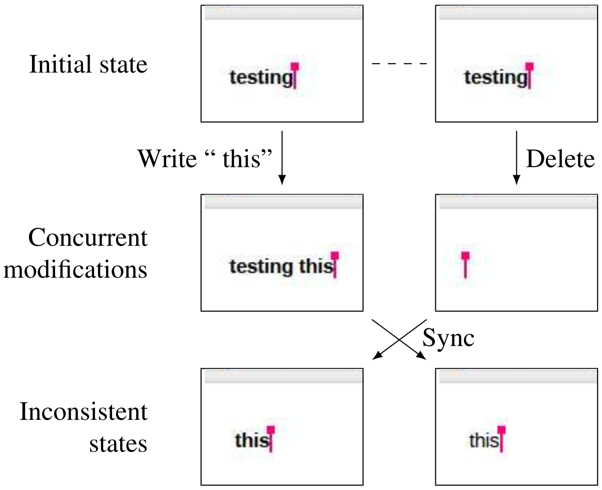


Figure 1: A bug in Google Docs where client states differ after concurrent modification of a line.

This paper presents Simian, a systematic black-box approach to analyze collaborative web applications.[5] Given a set of UI actions that exercise individual features of an application, Simian synthesizes concurrent interactions aimed at reaching inconsistencies between clients, i.e., a situation where the application fails to provide eventual consistency. The main challenge is the huge number of possible interactions, which are practically impossible to explore. Simian addresses this challenge with a novel two-phase approach for analyzing concurrent software. It consists of a sequential learning phase, which exhaustively checks for potentially conflicting actions, and a concurrent analysis phase, which exercises the potential conflicts using concurrent clients that interact on the same shared state. Simian reports a problem when a concurrent interaction leads to an inconsistency between the clients, such as the example in Figure 1.

Our approach provides several benefits. First, it is automatic and does not need a developer to specify potentially bug-revealing interactions. Second, due to its black-box view on the application code, Simian easily scales to real-world applications with complex client-side and server-side implementations. Third, it is precise in the sense that it does not report any false positives, because all reported inconsistencies are indeed observed during the analysis. Fourth, Simian is systematic, by exploring all potential conflicts up to a configurable bound, yet more efficient than naive exhaustive exploration.

It is important to note that our work focuses on synthesizing concurrent interactions, which is orthogonal to the problem of exploring the low-level runtime interleavings of these interactions. Previous work on software model checking [16, 20, 26, 44] addresses the latter problem. Adapting these techniques to collaborative web applications remains as a challenge for future work. We here focus on synthesizing concurrent interactions because not taking interleavings into account greatly reduces the search space and simplifies the implementation of our approach, while still revealing a large number of bugs in widely used applications.

We have implemented Simian for the domain of collaborative editors and evaluated it with three popular systems: Google Docs, Firepad,[6] and ownCloud Documents.[7] Simian reveals a variety of inconsistencies, including incorrect formatting, incorrectly ordered text fragments, duplicated text, and various visual discrepancies. The inconsistencies are triggered by different sequences of user actions, yet the root causes of the bugs that trigger the inconsistencies may overlap. Still, the inconsistencies represent a broad range of problems, as we discuss in Section 5. Compared to a naive systematic exploration of concurrent interactions, the two-phase approach makes the analysis 10x faster, finding an inconsistency approximately every 9 minutes, on average.

In summary, this paper contributes the following:

- A black-box analysis technique to detect errors in complex concurrent programs. The key novelty is a two-phase approach to systematically analyze all potentially bug-exposing concurrent interactions up to a configurable depth.
- Applying the idea to collaborative web applications, an increasingly important class of applications that has not been targeted by existing analysis techniques.
- Empirical evidence that the approach finds a variety of bugs in widely used and well tested applications. To the best of our knowledge, our work is the first automatic approach that detects bugs in these systems.

The paper is organized as follows. In Section 2, we give an overview of the problem and our solution, Simian, along with a motivating example. In Section 3, we explain Simian in more detail, and in Section 4, we describe our implementation. We evaluate the effectiveness and efficiency of Simian and give further examples of the kinds of inconsistencies it detects in Section 5. Section 6 discusses related work, and Section 7 concludes.

2. Overview and Example

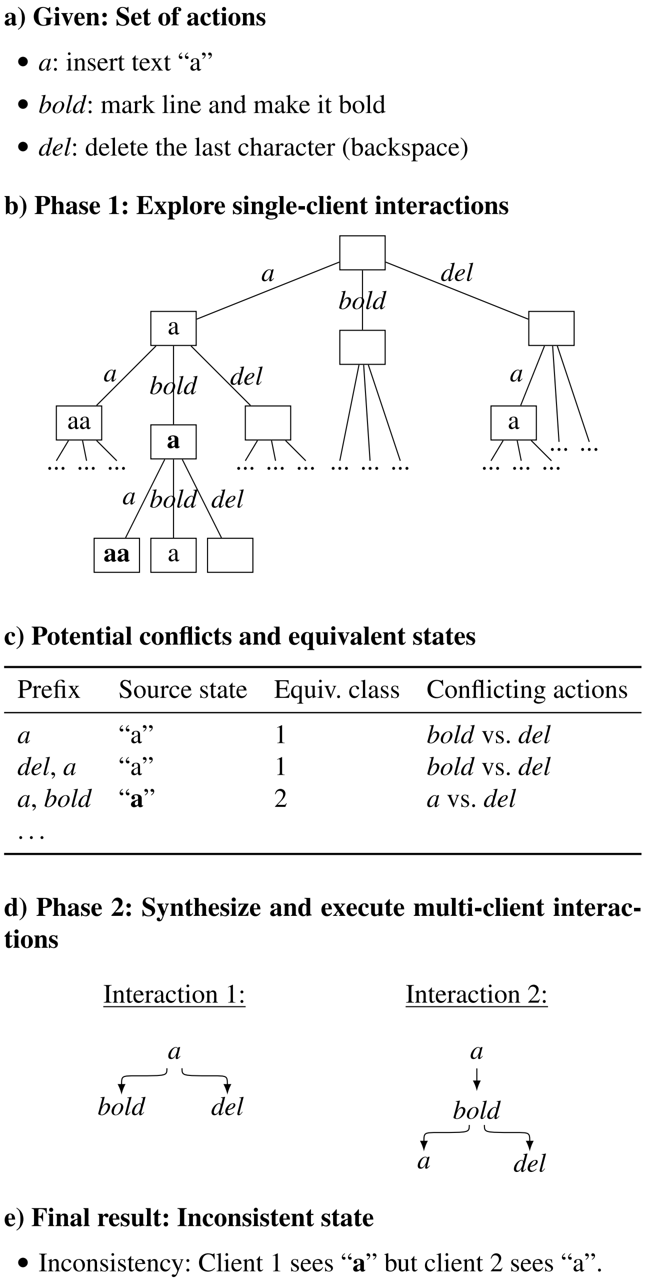
The following section informally describes the key ideas of our approach using a motivating example. The example is a simplified version of the bug in Google Docs illustrated in Figure 1. Google Docs is a collaborative editor implemented as a web application, where multiple clients can simultaneously edit a shared document. The intended behavior is that all clients see the same state of the shared document.

To analyze with Simian whether the application behaves as expected, a developer defines a set of actions. Each action

[5] Simian stands for "systematic exploration of multi-client interactions."

[6] https://firepad.io

[7] https://github.com/owncloud/documents/



a) Given: Set of actions

- *a*: insert text “a”
- *bold*: mark line and make it bold
- *del*: delete the last character (backspace)

b) Phase 1: Explore single-client interactions

c) Potential conflicts and equivalent states

Prefix	Source state	Equiv. class	Conflicting actions
a	“a”	1	*bold* vs. *del*
del, *a*	“a”	1	*bold* vs. *del*
a, *bold*	“**a**”	2	*a* vs. *del*
…			

d) Phase 2: Synthesize and execute multi-client interactions

e) Final result: Inconsistent state

- Inconsistency: Client 1 sees “**a**” but client 2 sees “a”.

Figure 2: Running example that illustrates our two-phase approach for synthesizing multi-client interactions that reach an inconsistent state.

implements a logical step triggered by a client when interacting with the application. For our running example, suppose the developer specifies the three actions listed in Figure 2a.

The goal of our approach is to synthesize an interaction of multiple clients that exposes an error in the application. The search space to find such an interaction based on the given set of actions comprises interactions between an arbitrary number of clients that each trigger an arbitrarily long sequence of actions. We limit the search space to interactions where, at first, one client triggers a sequence of actions, without any other concurrent clients, and then, two concurrent clients each trigger one action in parallel, with an arbitrary interleaving of low-level system events. Even when focusing on such interactions, the search space still remains infinitely large because of the arbitrary length of the sequential prefix.

To effectively explore the space of possible interactions, Simian takes a two-phase approach. In the first phase, it systematically explores all single-client interactions up to a fixed depth. The goal of this first step is to identify pairs of actions that may conflict with each other in a particular application state, and to identify application states that are equivalent to each other.

For the example, suppose that Simian explores all single-client interactions created from the given set of actions up to a depth of three. Figure 2b illustrates the tree of the explored sequences of actions. The nodes in the tree summarize the state of the application by showing the content of the document after triggering a sequence of actions. The edges in the tree represent the actions triggered by the client. For example, the root node is empty because Simian starts with an empty document and no action has been triggered yet. The left-most child of the root node contains “a” because the client has triggered the action that inserts the text “a”. The text cursor is abstracted away from the state.

By analyzing the states reached via the interactions in Figure 2b, Simian learns about potential conflicts between actions and about equivalent states. To this end, it abstracts the client-side states reached after each action and compares these states with each other. We choose to abstract the state into the pixels of a screenshot and compare states based on this representation.

Figure 2c summarizes some of the detected potential conflicts and equivalent states. For example, after triggering actions *a* and *bold*, the two actions *a* and *del* are detected as a potential conflict because the parts of the screen affected by these actions overlap. The “Source state” column of the table shows that the states reached via the sequences *a* and *del, a* are equivalent, because both yield a document that contains the text “a”.

In the second phase, Simian uses the knowledge learned from exploring single-client interactions to synthesize multi-client interactions that trigger potential conflicts. The main idea is to create one multi-client interaction for each potential conflict, without repeatedly analyzing the same pair of conflicting actions in equivalent states.

For the learned knowledge summarized in Figure 2c, Simian synthesizes two concurrent interactions, as illustrated in Figure 2d. The first interaction involves two clients that concurrently trigger *bold* and *del*, respectively, in a document initialized by action *a*. The second interaction concurrently triggers the actions *a* and *del* in a document initialized by *a, bold.* These two interactions are sufficient to cover the three potential conflicts in Figure 2c because two of the potential conflicts have different prefixes but reach a single equivalent state "a".

Simian executes each synthesized concurrent interaction and checks for inconsistencies. When executing the second concurrent interaction in Google Docs, the documents seen by the two clients may differ from each other. One client sees "**a**", whereas the other client sees "a", i.e., without the bold formatting. Waiting, e.g., a minute does not change anything, so the situation is not just an acceptable consequence of the eventual consistency policy. Such an inconsistency clearly violates the intended behavior of a collaborative editor.

3. Approach

We now present a detailed description of our approach for automatically analyzing collaborative web applications using a systematic black-box technique.

3.1 Correctness of Collaborative Editors

Our overall goal is to systematically detect erroneous behavior in concurrent software systems that have multiple clients. Based on experience from concurrency errors in multi-threaded programs, which shows that two threads are sufficient to trigger the vast majority of errors [21], we focus on two clients that interact with the system. The techniques presented here can be generalized to more than two clients.

A multi-client software system can be considered as a labeled transition system $(\mathcal{S}, \mathcal{A}, \rightarrow)$ where $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a set of actions that trigger transitions, and $\rightarrow$ is a transition relation, i.e., a subset of $\mathcal{S} \times \mathcal{A} \times \mathcal{S}$. Each state $s = (\sigma_1, \sigma_2, \sigma_{sys}) \in \mathcal{S}$ is a triple of state components that represent the state observable by the two clients and the state of the rest of the system, respectively. An action is either a *client action*, which is triggered by one of the clients, or any other action triggered by some other part of the system. The transition relation represents how triggering an action influences the overall system.

In a collaborative web application, the client states σ_1 and σ_2 correspond to the respective DOM states of the web clients. The system state σ_{sys} comprises the server-side state of the application, the state of the network, and any other state that influences the application. A client action is any logical operation performed by a client. For example, for a text editor, "append text to end of line" or "select line and make it bold" are client actions. A single client action may correspond to multiple implementation-level steps, e.g., multiple events triggered in the client-side JavaScript code or multiple network interactions between client and server. Other, non-client actions are any other operations performed by the application, e.g., sending a request from the server to the client or replicating the server-side state into redundant databases.

Correctness can be defined using the notion of operational transformations [13], a non-blocking method of concurrency control. Consider two clients C_1 and C_2 that concurrently generate the operations $op_1 \parallel op_2$. Client C_1 applies op_1 to its state immediately upon creation, as does C_2 with op_2, and they send update notifications to each other. Upon arrival, the operations are transformed by the transformation function T. Client C_1 executes $T(op_2) = op_2'$ and C_2 executes $T(op_1) = op_1'$. The transformation is chosen to ensure that, for input state $s = (\sigma_1, \sigma_2, \sigma_{sys})$ with equivalent client states $\sigma_1 \equiv \sigma_2$, the transitions

$$\sigma_1 \xrightarrow{op_1, op_2'} \sigma_1'$$

$$\sigma_2 \xrightarrow{op_2, op_1'} \sigma_2'$$

produce equivalent output client states $\sigma_1' \equiv \sigma_2'$. Operational transformation does not require the result of $op_1 \parallel op_2$ to be equal to a serialization of the two operations.

Ellis and Gibbs [13] define the correctness of a collaborative editor as follows:

DEFINITION 1 (Correctness). *A collaborative editor is* correct *if and only if it fulfills the following two properties:*

- Precedence property – *For each pair of operations op_1 and op_2, if the client generating op_2 executes op_1 before generating op_2, then op_1 is executed before op_2 on each client.*
- Convergence property – *When all generated operations have been executed on all clients, then all clients have identical states.*

The convergence property is an eventual consistency [45] property in the sense that consistency of client states can only be guaranteed after all operations have been fully performed and the system is in a quiescent state.

3.2 Problem Statement and Challenges

In this work, we reveal violations of the above correctness property by searching for quiescent states where the client states are not equivalent, i.e., violations of the convergence property. The input to our approach are the system under test and a set $\mathcal{A}$ of client actions that each implement a logical step of a client interacting with the system. The complexity of real-world collaborative web applications makes systematic reasoning about the entire application state practically impossible. To deal with this problem, our work treats the system state σ_{sys} as a black box and restricts itself to triggering client actions.

A sequence of client actions $(a_1, .., a_j)$ triggers transitions $s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \ldots \xrightarrow{a_j} s_{j+1}$. We call a sequence of actions by a single client a *single-client interaction*. We assume that the execution of serializable interactions, and as such single-client interactions, is deterministic. In case this assumption fails, Simian may miss bugs related to possible non-determinism of single-client interactions, however, we primarily target bugs that involve multiple clients.

In contrast to single-client interactions, a *multi-client interaction* is an interleaving of two sequences of actions, each of which is triggered by one client. Simian generates multi-client interactions $a_1, .., a_j, (a_x || a_y)$, which consist of a single-client prefix $a_1, .., a_j$, where all actions are triggered by only one client, and a concurrent suffix, where two clients each trigger an action, a_x and a_y, concurrently. For the suffix, there may be many possible interleavings of the low-level events triggered by the actions a_x and a_y. The problem of exploring these interleavings is orthogonal to the problem considered in this work.

Our work focuses on violations of the convergence property caused by executing two concurrent actions in two clients. Focusing on this kind of multi-client interaction allows the analysis to detect concurrency errors that match various classes of problems known from multi-threaded applications. In particular, the analysis is able to detect atomicity violations, because a single action typically consists of multiple implementation-level operations, and data races, because concurrent actions may trigger concurrent reads and writes of shared data.

Based on the above definitions, we formulate the problem addressed in this work as follows: Given a collaborative web application and a set of client actions, find a multi-client interaction $a_1, .., a_j, (a_x \mid\mid a_y)$ where the states of the clients after the interaction do not converge. For such an interaction, we say that the actions a_x and a_y are *conflicting* actions.

The key challenge for achieving this goal is the huge search space. We partially address this challenge by limiting the length of interactions to a maximum length $k = j + 1$, for a configurable bound k. A naive approach to generating multi-client interactions of this restricted form would be to exhaustively try all possible interactions within the bound k. However, the number of such interactions is exponential in k, making it practically infeasible to perform this exhaustive exploration, even with a low bound.

3.3 Overview of Simian

Our analysis, Simian, has two phases. First, it systematically explores single-client interactions up to a maximum length k to learn which pairs of actions may conflict with each other in a particular state. Second, it exploits the potential conflicts identified in the first phase to generate multi-client interactions targeted at exploring each potential conflict, and reports divergent states. We present the two phases of the analysis in Sections 3.4 and 3.5, respectively.

The approach relies on techniques to identify potentially conflicting actions, equivalent states, and inconsistent states. These techniques depend on the kind of application our approach is applied to. Section 3.6 presents techniques suitable for a black-box analysis of collaborative web applications, such as collaborative text editors.

3.4 Phase 1: Systematic Sequential Exploration

The first phase of Simian explores all sequences of actions in $\mathcal{A}$ triggered by a single client up to a configurable bound k. We refer to this set of sequences as the *action tree* $\mathcal{A}^k$, as seen in Figure 2b. Each path through the action tree is one sequence of actions triggered by a single client in a sequential manner. We call states with outgoing actions *source states*. We identify a state s_j by the sequence of actions $s_1 \xrightarrow{a_1} ... \xrightarrow{a_j} s_j$ that leads to the state.

Algorithm 1 Explore single-client interactions

```
Input: Set $\mathcal{A}$ of actions, exploration depth $k$
Output: Set $\mathcal{C}$ of potential conflicts
 1: $\mathcal{E} \leftarrow \emptyset$      ▷ Maps $(a_1, ..., a_j)$ to the data affected by $a_j$
 2: for each $(a_1, a_2, .., a_k) \in \mathcal{A}^k$ do
 3:     $effect_1, effect_2, .., effect_k \leftarrow execute(a_1, ..., a_k)$
 4:     for $i = 1, .., k-1$ do
 5:         $\mathcal{E} \leftarrow \mathcal{E} \cup \{a_1, .., a_{i-1} \mapsto effect_i\}$
 6: $\mathcal{C} \leftarrow \emptyset$
 7: for each $(a_1, a_2, .., a_j) \in \mathcal{A}^j, j \in \{1, \ldots, k-1\}$ do
 8:     $seq \leftarrow (a_1, a_2, .., a_j)$                ▷ Source state
 9:     for each $(a_x, a_y)$ s.th. $a_x \in \mathcal{A}, a_y \in \mathcal{A}$ do
10:         $seq_x \leftarrow (a_1, a_2, .., a_j, a_x)$
11:         $seq_y \leftarrow (a_1, a_2, .., a_j, a_y)$
12:         if $\mathcal{E}(seq_x) \cap \mathcal{E}(seq_y) \neq \emptyset$ then
13:             $\mathcal{C} \leftarrow \mathcal{C} \cup \{(seq, a_x, a_y)\}$
```

Simian checks for each node in the tree, which pairs of outgoing actions affect the same data, and identifies these actions as potential conflicts.

Definition 2 (Potential conflict). *A potential conflict is a triple (seq, a_x, a_y) where $seq = a_1, .., a_j$ is a sequence of actions, such that a_x and a_y are actions that affect the same data when executing after seq.*

The rationale for this way of identifying potential conflicts is that two actions must affect the same data to produce an inconsistency. Potential conflicts may not produce actual conflicts of the actions a_x and a_y when they are executed concurrently, which is why the second phase of Simian validates them.

Algorithm 1 summarizes how Simian explores the tree of single-client interactions to detect potential conflicts. The algorithm takes as an input the set $\mathcal{A}$ of actions and the exploration depth k. The output is the set of potential conflicts $\mathcal{C}$. The algorithm consists of two main steps. The first step (lines 1–5) is to execute each path of length k through the action tree while recording the effects of individual actions. Intuitively, the effect of an action triggered in a particular state represents the data affected by the action. The second step (lines 6 to 13) is to compute potential conflicts by analyzing to what extent recorded effects of different actions overlap. There are various options for how to record the effect of an action and how to compute overlaps between effects. Section 3.6.2 presents an approach suitable for black-box analysis of collaborative web applications, which is based on the state that is visible to the clients. We next describe the two steps of Algorithm 1 in more detail.

Algorithm 2 Synthesize and execute multi-client interactions

```
Input: Set $\mathcal{C}$ of potential conflicts
Output: Interactions that reach an inconsistent state
1: $explored \leftarrow \{\}$   $\triangleright$ Maps equivalence classes of states to sets of action pairs
2: for each $(seq, a_x, a_y) \in \mathcal{C}$ do
3:     if $(a_x, a_y) \in explored(equivCls(seq))$ then
4:         continue
5:     $explored(equivCls(seq)) \leftarrow$
               $explored(equivCls(seq)) \cup \{(a_x, a_y)\}$
6:     $interaction \leftarrow [a_1, .., a_j, (a_x \mid\mid a_y)]$
               where $seq = a_1, .., a_j$
7:     $s \leftarrow execute(interaction)$
8:     if $isInconsistent(s)$ then
9:         report interaction and error
```

At first, the algorithm initializes a map $\mathcal{E}$ (line 1) that for each sequence of actions $(a_1, ..., a_j)$ records the effect of action a_j in the state reached via $a_1, ..., a_{j-1}$, Then, the algorithm populates $\mathcal{E}$ by exploring all sequences of actions up to depth k (lines 2–5). To this end, the algorithm executes each sequence of length k, while recording the effects of each individual action (line 3). The effect of an action a_i is represented by $effect_i$.

Based on the effects of all actions, the algorithm computes the set of potential conflicts $\mathcal{C}$. It iterates over all source states, i.e., all sequences of actions of length between 1 and $k-1$ (lines 7–13). For each source state, it considers all pairs (a_x, a_y) of actions, $a_x, a_y \in \mathcal{A}$ that may be executed in the source state and compares their effects. For this purpose, the algorithm queries the map $\mathcal{E}$ with the two sequences that results from appending a_x and a_y to the sequence that leads to the source state. If and only if the effects of a_x and a_y overlap (line 12), the potential conflict is stored into $\mathcal{C}$.

For illustration, consider the action tree in Figure 2b and suppose that the exploration bound is $k = 3$. Algorithm 1 executes all 27 possible sequences of three actions, such as, (*a,a,a*), (*a,a,bold*), and (*a,a,del*), and records the effect of each action. For example, for the source state reached by (*a,a*), it records the effect of *a*, *bold*, and *del*, and checks these effects for overlaps.

The first phase of Simian is systematic and deterministic. It is systematic because the approach explores all single-client interactions up to a configurable length. It is deterministic under the assumption that single-client interactions with a collaborative web application are deterministic. We do not experience any non-determinism during the first phase of Simian in our experiments.

3.5 Phase 2: Conflict-Guided Concurrent Exploration

Algorithm 2 summarizes how our analysis synthesizes and executes multi-client interactions to find interactions that end in an inconsistent state. The input to Algorithm 2 is the set of potential conflicts $\mathcal{C}$, and the output is a set of multi-client interactions that reach an inconsistent state. For each potential conflict $((a_1, .., a_j), a_x, a_y)$, the algorithm assembles an interaction that is comprised of a single-client prefix $a_1, .., a_j$ and a concurrent suffix $(a_x \mid\mid a_y)$ (line 6). The algorithm executes this interaction and checks whether the resulting state is inconsistent. We describe in Section 3.6.3 how Simian identifies inconsistent states based on the state visible in clients.

To avoid exploring conflicts redundantly, Algorithm 2 uses a function *equivCls* that assigns to each state an equivalence class and keeps track of the equivalence classes in which particular pairs of actions have already been explored. To this end, it maintains a map *explored* that assigns to each state equivalence class the set of action pairs that have already been executed in states of this equivalence class. If a potential conflict has already been analyzed for another source state of the same equivalence class, the algorithm skips the conflict (lines 3–4).

We find in our experiments that this optimization significantly reduces the effort spent in the second phase of Simian. The effectiveness of the optimization and whether the optimization may introduce false negatives, i.e., missed bugs, depends on how accurately *equivCls* identifies equivalence classes of states. Section 3.6.1 describes how Simian realizes *equivCls* for collaborative web applications.

For our running example, reconsider Figure 2d, which shows the two multi-client interactions that Algorithm 2 executes based on the potential conflicts and the equivalence classes shown in Figure 2c. Suppose that executing these interactions reveals that the second interaction leads to an inconsistent state. The algorithm then reports the interaction, along with a description of the error, to the developer.

A naive alternative to our two-phase approach would be to exhaustively explore all multi-client interactions up to a particular bound k, as outlined at the end of Section 3.2. As we show in detail in our evaluation, focusing on potential conflicts instead of a naive exhaustive exploration greatly reduces the number of executions, which makes the approach viable in the first place.

3.6 Reasoning about Actions and States

Simian relies on techniques for identifying equivalent states, conflicting actions, and inconsistent states. We now present such techniques suitable for collaborative web applications. The main idea is to abstract the state of a client based on the rendered web site shown in the browser, and to use this state abstraction to reason about actions and states. Alternative approaches include to reason about the state of the DOM, the JavaScript heap of the client-side application, and the various client-side storage mechanisms, such as cookies and web storage. The benefit of our approach is that it treats the application as a black box, making Simian easily applicable to complex applications without requiring any knowledge about the implementation of the client-side or server-side of the application.

The core of our technique to reason about actions and states is a pixel-based abstraction ϕ of client state. Given two client states σ_1 and σ_2, the abstraction considers them to be equivalent, $\phi(\sigma_1) = \phi(\sigma_2)$, when a screenshot of both rendered web sites contains exactly the same pixels. More concretely, suppose that ϕ yields a double-indexed array of pixels, then the following function computes the differences between two states:

$$\mathit{diff}(\sigma_1, \sigma_2) = \{(x, y) \mid \phi(\sigma_1)[x, y] \neq \phi(\sigma_2)[x, y]\}$$

Using this definition, we consider two abstracted states to be equivalent if $\mathit{diff}(\sigma_1, \sigma_2) = \emptyset$.

Since different rendering engines, window sizes, operating systems, etc. may influence the state abstraction, we keep all these factors stable when implementing Simian. Because our entire analysis can easily run on a single computer, this constraint as not an issue in practice.

3.6.1 Identifying Equivalent States

The second phase of our analysis uses a function $\mathit{equivCls}$ that assigns an equivalence class to each state. Based on the pixel-based state abstraction, we consider two states to belong to the same equivalence class when they have the same state abstraction. In other words,

$$\mathit{equivCls}(\mathit{seq}_1) = \mathit{equivCls}(\mathit{seq}_2) \Leftrightarrow \mathit{diff}(\sigma_1, \sigma_2) = \emptyset$$

where σ_1 and σ_2 are the client states reached through the sequences of actions seq_1 and seq_2, respectively.

For our running example, the initial state (i.e., the root node in Figure 2b) and the state reached by triggering "bold" in the initial state are equivalent. The reason is that both states correspond to an empty document and therefore the same pixel-based state abstraction. However, note that state equivalence may not be preserved by triggering further actions. For example, after performing the action *a* on both of the above states, the resulting states are not equivalent anymore. The reason is that *a* is printed in a non-bold font in one state but in bold font in the other state.

3.6.2 Identifying Conflicting Actions

The first phase of our analysis identifies potential conflicts by recording the effects of individual actions and by comparing these effects with each other. Simian reasons about the effects of actions based on the pixel-based state abstraction by comparing the abstracted state before and after an action:

- *Effects*. For an action a invoked in a state σ that leads to a state σ', the effect of a is the set $\mathit{diff}(\sigma, \sigma')$ of pixels that differ between the two states.
- *Overlap of effects*. To check whether two actions a_1 and a_2 overlap, suppose that σ_1, σ_1', σ_2, and σ_2' are the states before and after these actions. We consider the effects of a_1 and a_2 as overlapping if and only if the following condition holds:

$$\mathit{diff}(\sigma_1, \sigma_1') \cap \mathit{diff}(\sigma_2, \sigma_2') \neq \emptyset$$

In the running example, our technique detects no conflicting actions among the three actions triggered in the root node of Figure 2b. The reason is that neither *bold* nor *del* influence the abstracted state, and therefore do not have overlapping effects with any other action. In contrast, the technique identifies several conflicting actions in the state reached by triggering *a*: both *bold* and *del* affect the existing character "a". Furthermore, triggering *a* again writes another character next to the existing character, and because both characters are adjacent, the areas of their affected pixels overlap. As a result, all three actions are found to be pairwise conflicting with each other.

3.6.3 Identifying Inconsistent States

When executing multi-client interactions in the second phase of Simian, the analysis uses a function $\mathit{isInconsistent}$ that decides whether the state of the application is inconsistent. To find a violation of the convergence property (Definition 1), Simian first needs to wait until the system reaches a quiescent state, in which all operations have been performed on all clients. Because we use a black-box approach that cannot directly detect when the system reaches quiescence, we instead heuristically wait for a configurable amount of time and assume all clients have processed all events within that window.

Simian uses the pixel-based state abstraction to define a generic consistency check between the states seen by the two clients involved in the interaction. Concretely, let σ_{C1} and σ_{C2} be the states of the two clients, then the approach considers the two states as inconsistent if and only if the following condition holds:

$$\mathit{diff}(\sigma_{C1}, \sigma_{C2}) \neq \emptyset$$

For the running example, the second multi-client interaction in Figure 2d produces an inconsistent state, where one client shows "**a**" while the other client shows "a", as discussed in Section 2. Because the pixel-based abstractions of these two states differ, the analysis reports an inconsistency.

Our approach for synthesizing multi-client interactions is independent of the correctness criterion used to identify inconsistent states. Alternative to or in addition to the pixel-based check for inconsistencies, other correctness oracles, such as neutral event sequences [1] or application-specific specifications, can easily be plugged into our approach.

The pixel-based approach is accurate for most applications, such as collaborative text editors, drawing applications, and shared calendars, and has the benefit of being easy to reason about and to implement. It is, however, not applicable to every kind of collaborative application. For example, consider a chat window, where most actions insert a new line into the chat, pushing existing lines further to the top. In such an application, the pixel-based state abstraction would yield a large set of changed pixels even for trivial actions, reducing the effectiveness of Simian's two-phase approach. While such applications certainly exist, we find that the simple pixel-based state abstraction is suitable for a large class of widely used applications (Section 5).

4. Implementation

We have implemented our analysis using the Java edition of the Selenium WebDriver framework and Mozilla Firefox 45. Our implementation is available at https://github.com/marinabilles/simian. Users provide implementations of actions using the Selenium framework.

The implementation takes screenshots for determining affected areas of actions. Our pixel-by-pixel comparison algorithm overapproximates potential conflicts, as some pixels identified as changed are not related to the performed user actions, but are rather caused by the idiosyncrasies of the platform's native font rendering. To exclude minor differences between screenshots, we use a 8-neighbor flood filling algorithm to detect the size of connected changed pixel areas and exclude such areas that are less than 10 pixels in size.

For Phase 2, we spawn separate Java threads, which each take control of their own WebDriver instance. We use Selenium Grid with grid nodes running in their own X server session running on the same machine as the grid hub. Using multiple X server sessions avoids problems that Selenium would encounter when trying to control a not currently focused window.

Before triggering the concurrent actions of a multi-client interaction, the implementation sets the cursor of both clients to the location where it is after executing the prefix, to ensure that the concurrent actions are the same as the ones explored during the first phase of the approach. There is a waiting time before and after the concurrent part of a multi-client interaction to give the application time to synchronize and to reach a consistent state. This timeout can be configured individually for each subject application depending on the time each application needs for the states to stabilize.

5. Evaluation

5.1 Research Questions

We pose the following research questions:

RQ1 How effective is Simian at finding inconsistencies in collaborate web applications, and what is the nature of these inconsistencies?

$$\mathcal{A}_{large}\begin{cases} \mathcal{A}_{small}\begin{cases}\text{Type “a”}\\ \text{Press Return}\\ \text{Toggle bold on line before cursor}\\ \text{Set font face to Verdana on line before cursor}\\ \text{Select and delete line before cursor}\end{cases}\\ \text{Press Tab}\\ \text{Press Space}\\ \text{Type “b”}\\ \text{Toggle italic on line after cursor}\\ \text{Set font size to 18 on line before cursor}\end{cases}$$

Figure 3: Actions used in the evaluation.

RQ2 How do the effectiveness and the efficiency of Simian depend on the size of the action set $\mathcal{A}$ and the depth bound k?

RQ3 How does Simian compare to a single-phase, exhaustive exploration of multi-client interactions?

5.2 Experimental Setup

Benchmark applications We run Simian on three widely used collaborative editors: Google Docs, Firepad, and ownCloud Documents. According to their developers, these systems use operational transformations [13] for synchronizing changes between clients[8]. Since the implementation of Google Docs is not available to us, we use the publicly available installation hosted by Google. All experiments were performed between Nov 4 and Nov 15, 2016. For Firepad and ownCloud Documents, which are available as open-source, we locally install the applications, using version 1.4.0 and 0.13.1, respectively. To avoid false positives caused by our pixel-based state abstraction (Section 3.6), we specify for each application particular areas of the screen that are ignored, such as a chat box shown by Google Docs and the DOM element that represents the blinking cursor.

We set the waiting time before and after the concurrent actions of multi-client interactions to 7 seconds for Google Docs and Firepad, and to 12 seconds for ownCloud. Based on our experience, this is sufficient for allowing the applications to stabilize after user input.

Actions We define two sets, $\mathcal{A}_{small}$ and $\mathcal{A}_{large}$, of actions that we implement for all benchmark applications (Figure 3). We choose these action sets to reflect typical user actions that are available across all three applications. Each action is implemented as an application-specific sequence of Selenium commands that interact with the text editor. For example, for Google Docs, the *Type "a"* action sends a series of keys to the hidden `iframe` element in the page that Google Docs uses to handle keyboard input. For Firepad, there is a hidden `textarea` that accepts text input. We also tried a larger set

[8] https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs_22.html,
https://firepad.io/,
http://webodf.org/about/features.html

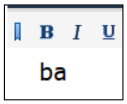
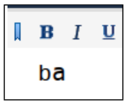
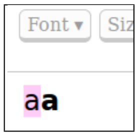
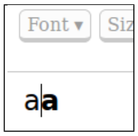
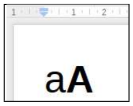
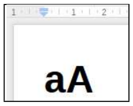
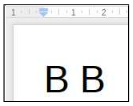
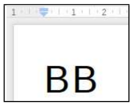
Application	Interaction	Screenshots		Description
		Client 1	Client 2	
ownCloud	*Set font Verdana* ↙ ↘ *Type “a”* *Type “b”*	 ba	 ba	Client 1 sees character “b” formatted in Verdana, Client 2 in the default font.
Firepad	*Type “a”* ↓ *Toggle bold* ↙ ↘ *Type “a”* *Toggle bold*	 aa	 aa	Client 1 has a shadowed highlight on the first “a” character, a marker that another client has currently selected this text. Client 1 still sees that Client 2 has selected text, while this is no longer the case.
Google Docs	*Type “a”* ↓ *Toggle bold* ↙ ↘ *Type “a”* *Press Space*	 aA	 aA	Pressing space capitalizes the first “a”. The text fragments are swapped: the second “a” gets incorrectly inserted before the capitalized “A”. Both clients see an incorrect state. Additionally, the font weight of “a” is inconsistent.
Google Docs	*Set font size to 18* ↓ *Type “b”* ↙ ↘ *Press Tab* *Press Space*	 B B	 BB	Pressing space auto-capitalizes the “b”, but the concurrent action by the other client causes the application to insert another spurious “B” after the space. Additionally, Client 2 keeps font size 18 for the space character, whereas on Client 1 resets it to the default.

Table 1: Examples of inconsistencies detected by Simian.

of 15 actions, but each experiment with it exceeded our time budget of 24 hours.

Hardware Simian and the locally installed web applications are running on a 2.10 GHz 4-core machine running Ubuntu 16.04.

5.3 Inconsistencies Detected by Simian (RQ1)

Analyzing the three benchmark applications with the $\mathcal{A}_{large}$ actions yields a total of 195 unique inconsistencies: 37 in Google Docs, 32 in Firepad, and 126 in ownCloud Documents. Unique here means that each inconsistency is triggered by a different multi-client interaction. Because we are not familiar with the implementations of the benchmark applications and have only partial access to them, we can only speculate about the number of unique root causes that trigger these inconsistencies. We observe that the inconsistencies represent a diverse set of problems, some of which we present next.

5.3.1 Representative Examples

Table 1 presents examples of the detected inconsistencies.[9] The ownCloud example as well as the motivating example for Google Docs in Figure 1 are inconsistencies where different clients see differently formatted variants of the same characters in the document. The Firepad example in Table 1 is an inconsistency where editing state that should be shared across clients is not correctly displayed. The two Google Docs examples in Table 1 are inconsistencies that lead to an erroneous state with problems beyond differently shown text and different editing state. These examples are particularly interesting because they demonstrate that inconsistent visible client states, as identified by Simian, sometimes indicate an even more severe problem. In both cases, both clients show an incorrect state due to incorrectly ordered or extra characters, and additionally there is a formatting inconsistency between the two clients, which allows Simian to report the interaction in the first place.

Overall, we find that the inconsistencies found by Simian cover a diverse set of problems across all three applications. Given our setup, the analysis identifies inconsistencies that involve only a few characters. Many of them can also be easily reproduced with larger text fragments.

5.3.2 Influence of Non-Determinism

To better understand to what extent the detected inconsistencies are due to non-deterministic behavior, we conduct two experiments. First, we re-execute ten times each multi-client interaction that has revealed an inconsistency and report how often it reveals the inconsistency again. Figure 4 presents the results. The number of multi-client interactions that reproduce the inconsistency in ten out of ten executions is shown in the right-most column. For example for Google

[9] Some screenshots have been modified to remove whitespace.

Application	$\lvert\mathcal{R}_1\rvert$	$\lvert\mathcal{R}_2\rvert$	$\lvert\mathcal{R}_1 \cap \mathcal{R}_2\rvert$	$\lvert\mathcal{R}_2 \backslash \mathcal{R}_1\rvert$
Google Docs	37	32	20	12
Firepad	32	42	24	18
ownCloud	126	127	125	2

Table 2: Comparison of inconsistencies detected by two runs of Simian. $\mathcal{R}_i$ refers to the set of inconsistency reports for run i.

Docs (Figure 4a), 11 of the total of 37 inconsistencies are reproduced in all ten re-executions. For Google Docs and Firepad, the majority of inconsistencies occur only in some executions, and ownCloud also has a non-negligible number of such inconsistencies.

Second, we re-execute all multi-client interactions considered in Phase 2 of the approach, i.e., independent of whether an interaction has revealed an inconsistency in our initial experiment. Then, we compare the set $\mathcal{R}_1$ of reported inconsistencies of the first run with the set $\mathcal{R}_2$ of reports of the second run. Table 2 summarizes the results. We find that, for Google Docs and Firepad, there is a considerable overlap between the two runs, yet the second run reveals a noteworthy number of additional inconsistencies. For ownCloud, we see little difference between the two runs, with most of the inconsistencies reproducing.

We conclude that many inconsistencies are caused by specific interleavings of the low-level events triggered by concurrent actions. These results motivate future work on exploring the interleavings of multi-client interactions in the context of collaborative web applications. Despite this limitation, our current approach has the significant advantage of being light-weight, not requiring instrumentation of application code or modifications of the low-level runtime system, and yet it is capable of effectively exposing bugs.

5.4 Influence of Actions and Exploration Depth (RQ2)

We run a series of experiments to examine the effect of the size $|\mathcal{A}|$ of the action set and the depth bound k. The experiments use the two action sets $\mathcal{A}_{small}$ and $\mathcal{A}_{large}$ (Figure 3) with bound $k = 3$. Furthermore, for $\mathcal{A}_{large}$, we additionally test $k = 1$ and $k = 2$.

Table 3 presents our results. The "Phase 1" block of the table shows for each experiment how many sequences of actions Simian explores during the first phase and how long it takes. Furthermore, we show the total number of source states to explore and how many equivalence classes of source states Simian identifies. The first column of the "# Pot. conflicts" block shows how many potential conflicts the approach detects (taking source state equivalence into account), which is equal to the number of multi-client interactions we execute. Finally, the "Phase 2" block of the table indicates how long exploring these potential conflicts takes (column "Simian") and how many inconsistencies the approach finds.

Figures 5a and 5b visualize the main results from these experiments. The figures compare the total analysis time and the number of detected inconsistencies per application across the experiments. For $\mathcal{A}_{large}, k = 1$, the execution time is insignificant but the analysis also is not very effective, only reporting a single inconsistency. For $\mathcal{A}_{small}, k = 3$ and $\mathcal{A}_{large}, k = 2$, the execution time is around one hour for Google Docs and ownCloud, and about 15 minutes for Firepad. Yet, the number of inconsistencies is still relatively small, ranging from 0 (Google Docs, $\mathcal{A}_{large}, k = 2$) to 19 (ownCloud, $\mathcal{A}_{small}, k = 3$). The $\mathcal{A}_{large}, k = 3$ setup shows the potential of our approach because the number of detected inconsistencies increases significantly. As expected, the execution time also increases, with Google Docs taking over 15 hours and Firepad being the fastest with 4:22h.

In the $\mathcal{A}_{large}, k = 3$ setup, Simian finds an inconsistency every 8:43 minutes, on average (24:29 minutes in Google Docs, 8:11 minutes in Firepad, 4:13 minutes in ownCloud). Given that the approach is a fully automated tool, we consider this time to be acceptable.

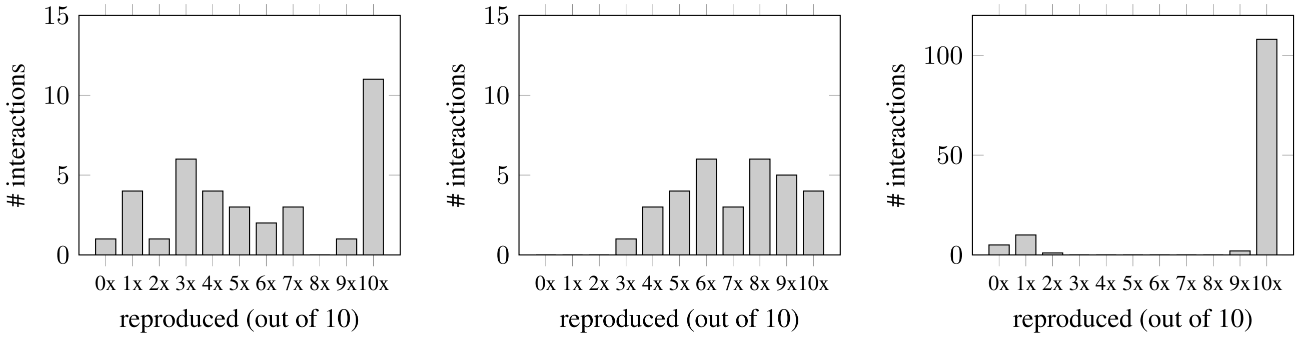
5.5 Comparison with Naive Exhaustive Exploration (RQ3)

Finally, we compare Simian to a naive approach that exhaustively explores all multi-client interactions, i.e., without first identifying potential conflicts. The "Naive" columns in Table 3 show the number of conflicts that such an approach considers and how long exploring all them would take. The times are estimated based on the average time taken to execute a multi-client interaction for the specific application. We find that, compared to the naive approach, Simian reduces the number of potential conflicts to explore by 92%, on average. As a result, the overall execution time of Simian reduces the execution time that the naive approach would take by 89%, i.e., Simian is 10x faster. We conclude that the two-phase approach taken in this work is worthwhile and key to scaling the systematic exploration of interactions to real-world applications.

6. Related Work

Simian relates to research on UI-level testing, concurrency bugs, and collaborative web applications, which we discuss in the following.

UI-level testing Various approaches for automatically testing applications at the UI-level have been proposed. To steer the testing toward potential problems, some approaches use feedback from executions, e.g., on coverage [3] or the performance of event handlers [32]. Other directions include to learn a finite state model of the application [9, 23, 25], to exploit informal specifications [43], to repeat typical user actions [6, 14], or to steer toward particular statements via symbolic execution [19]. None of these approaches analyzes



(a) Google Docs

(b) Firepad

(c) ownCloud

Figure 4: Reproducibility of inconsistencies across ten executions of synthesized multi-client interactions.

Application	$\mathcal{A}$	k	Phase 1				# Pot. conflicts		Phase 2		
			Sequences	Time	Source states		Simian	Naive	Time (hh:mm)		Reported inconsistencies
				(hh:mm)	All	Equiv. classes			Simian	Naive (est.)	
Google Docs	$\mathcal{A}_{small}$	3	125	00:30	31	7	46	310	00:34	04:55	4
	$\mathcal{A}_{large}$	1	10	00:02	1	1	1	45	$<$ 00:01	00:43	0
	$\mathcal{A}_{large}$	2	100	00:20	11	3	73	495	00:55	07:52	0
	$\mathcal{A}_{large}$	3	1000	04:01	111	23	698	4995	11:05	79:19	37
Firepad	$\mathcal{A}_{small}$	3	125	00:13	31	5	6	310	00:04	03:48	1
	$\mathcal{A}_{large}$	1	10	$<$ 00:01	1	1	1	45	$<$ 00:01	00:33	0
	$\mathcal{A}_{large}$	2	100	00:09	11	3	21	495	00:12	06:04	5
	$\mathcal{A}_{large}$	3	1000	02:02	111	19	190	4995	02:20	61:13	32
ownCloud	$\mathcal{A}_{small}$	3	125	00:25	31	14	42	310	00:46	07:04	19
	$\mathcal{A}_{large}$	1	10	00:02	1	1	2	45	00:02	01:02	1
	$\mathcal{A}_{large}$	2	100	00:17	11	7	52	495	00:49	11:17	15
	$\mathcal{A}_{large}$	3	1000	03:30	111	34	250	4995	05:21	113:49	126

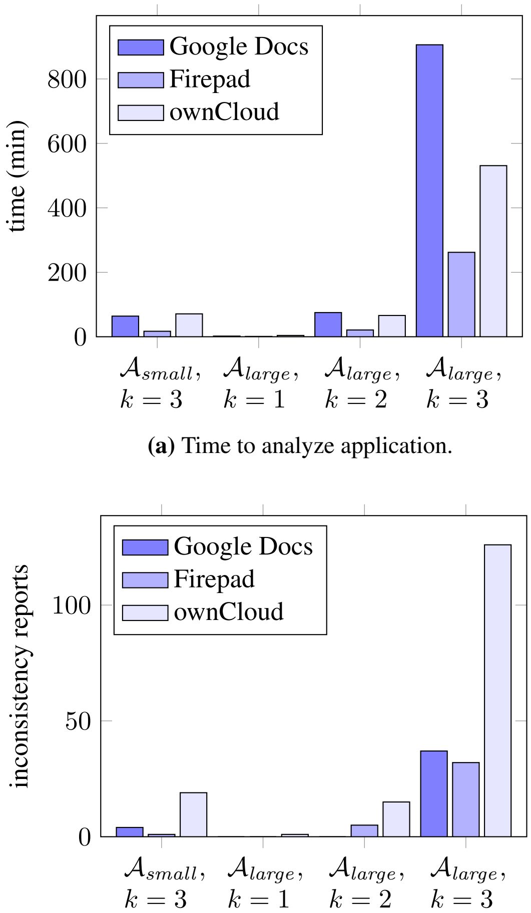
Table 3: Summary of results for different action sets and varying k.

multiple concurrent clients of an application, which is the focus of our work.

Races in event-based programs Event-based programs, such as web applications and other UI applications, suffer from data races that result from the non-deterministic execution order of event handlers. Recent work detects such races in web applications [29, 33] and Android applications [18, 22]. Other approaches filter potential harmful races by analyzing their effects on persistent state [27] and on the DOM [20]. All of these approaches target races that occur within a single client or a single application, whereas Simian addresses concurrency bugs that results from the interaction of multiple clients.

Generation of concurrent tests Several techniques test the correctness and performance of thread-safe classes by generating multi-threaded tests that exercise the methods of a shared instance. A random-based approach [30], a coverage-based approach [10], and several more heavyweight techniques that steer the generation toward data races [36], atomicity violations [35], deadlocks [34], crashing behavior [37], or particular interleavings [42] have been proposed. Simian and several of these approaches [34–37] share the idea to learn from sequential executions which concurrent interactions to explore. Our work differs from these approaches by analyzing at the application-level, where multiple clients interact, instead of the class-level, where multiple method calls interact. Complementary to generating concurrent tests, there is work on creating multi-threaded performance regression tests [31], which is orthogonal to the problem addressed here.

Concurrency bug detection Beyond test generation-based approaches, various other techniques for detecting concurrency bugs exist, such as dynamic analyses to detect data



(a) Time to analyze application.

(b) Inconsistencies found.

Figure 5: Comparison of different action sets and exploration bounds.

races [5, 12, 24] and atomicity violations [2, 15, 40], static analyses [17, 28, 47], and profilers to detect synchronization-related performance bottlenecks [48]. In contrast to these approaches, Simian is a black-box analysis that does not require low-level reasoning about the application code.

Schedule exploration To explore the different schedules of a concurrent program, several techniques have been proposed, e.g., software model checking [44], possibly optimized by dynamic partial order reduction [16, 46], random partial order sampling [38], preemption bounding [26], and other techniques to prioritize particular schedules [8, 11, 39, 41]. A prerequisite for schedule exploration is to have suitable inputs for the program, which is what Simian generates. Our work can be combined with existing schedule exploration techniques and may encourage future work to consider the problem in the context of collaborative web applications.

Collaborative web applications Several protocols for concurrency control of collaborative web applications have been proposed [4], e.g., based on operational transformations [13]. In practice, the problem of correctly implementing concurrency control for collaborative web applications remains challenging, as evidenced by the large amount of problems detected by Simian.

Eventual consistency Concurrency control protocols typically aim for eventual consistency [45], i.e., that eventually all clients converge to the same state. Although eventual consistency is a liveness property, the correctness condition we consider, expressed using operational transformations as explained in Section 3.1, is a safety property. Our analysis technique is therefore designed to search for states that fail to converge. This approach is related to recent work on serializability for data store clients [7]. That work presents a correctness criterion expressed as a notion of conflict serializability that takes eventual consistency into account, together with a dynamic analysis that approximates the criterion as a safety property based on specifications of commutativity and absorption for the relevant operation. An essential difference compared to Simian is that their dynamic analysis is not a black-box approach.

7. Conclusion

Collaborative web applications are complex software systems that are difficult to implement correctly. We have presented a novel analysis technique for detecting bugs in such applications. By choosing a black-box analysis design that abstracts away from the application source code and low-level event handling, the analysis becomes relatively simple to implement, yet highly effective at exposing bugs in widely used systems. The key insight of our approach is that efficiency can be achieved by organizing the analysis into two phases that first learn about conflicting actions and equivalent states and then synthesize multi-client interactions.

For future work, we want to apply the approach to other multi-client web applications not from the collaborative editor domain, such as chatting software and web-based multiplayer games. It may also be interesting to combine our approach of generating high-level multi-client interactions with existing model checking techniques to more thoroughly analyze the low-level non-determinism.

Acknowledgments

We thank the anonymous reviewers and our shepherd Stephen Freund for their comments and guidance. Our research has been supported by the DFG within ConcSys, by the BMBF and the HMWK within CRISP, and by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 647544).

References

[1] C. Q. Adamsen, G. Mezzetti, and A. Møller. Systematic execution of Android test suites in adverse conditions. In *nternational Symposium on Software Testing and Analysis (ISSTA)*, pages 83–93, 2015.

[2] C. Artho, K. Havelund, and A. Biere. High-level data races. *Software Testing, Verification and Reliability*, 13(4):207–227, 2003.

[3] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip. A framework for automated testing of JavaScript web applications. In *International Conference on Software Engineering (ICSE)*, pages 571–580, 2011.

[4] H. Attiya, S. Burckhardt, A. Gotsman, A. Morrison, H. Yang, and M. Zawirski. Specification and complexity of collaborative text editing. In *Symposium on Principles of Distributed Computing (PODC)*, pages 259–268, 2016.

[5] M. D. Bond, K. E. Coons, and K. S. McKinley. PACER: proportional detection of data races. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 255–268, 2010.

[6] P. A. Brooks and A. M. Memon. Automated GUI testing guided by usage profiles. In *International Conference on Automated Software Engineering (ASE)*, pages 333–342, 2007.

[7] L. Brutschy, D. Dimitrov, P. Müller, and M. Vechev. Serializability for eventual consistency: criterion, analysis, and applications. In *Symposium on Principles of Programming Languages (POPL)*, pages 458–472, 2017.

[8] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 167–178, 2010.

[9] W. Choi, G. Necula, and K. Sen. Guided GUI testing of Android apps with minimal restart and approximate learning. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 623–640, 2013.

[10] A. Choudhary, S. Lu, and M. Pradel. Efficient detection of thread safety violations via coverage-guided generation of concurrent tests. In *International Conference on Software Engineering (ICSE)*, 2017.

[11] K. E. Coons, S. Burckhardt, and M. Musuvathi. GAMBIT: effective unit testing for concurrency libraries. In *Symposium on Principles and Practice of Parallel Programming, (PPOPP)*, pages 15–24, 2010.

[12] L. Effinger-Dean, B. Lucia, L. Ceze, D. Grossman, and H.-J. Boehm. IFRit: Interference-free regions for dynamic data-race detection. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 467–484, 2012.

[13] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *International Conference on Management of Data (MOD)*, pages 399–407, 1989.

[14] M. Ermuth and M. Pradel. Monkey see, monkey do: Effective generation of GUI tests with inferred macro events. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 82–93, 2016.

[15] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Symposium on Principles of Programming Languages (POPL)*, pages 256–267, 2004.

[16] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Symposium on Principles of Programming Languages (POPL)*, pages 110–121, 2005.

[17] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 338–349, 2003.

[18] C. Hsiao, C. Pereira, J. Yu, G. Pokam, S. Narayanasamy, P. M. Chen, Z. Kong, and J. Flinn. Race detection for event-driven mobile applications. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 326–336, 2014.

[19] C. S. Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 67–77, 2013.

[20] C. S. Jensen, A. Møller, V. Raychev, and M. Vechev. Stateless model checking of event-driven applications. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 57–73, 2015.

[21] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 329–339, 2008.

[22] P. Maiya, A. Kanade, and R. Majumdar. Race detection for Android applications. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 316–325, 2014.

[23] A. Marchetto, P. Tonella, and F. Ricca. State-based testing of Ajax web applications. In *International Conference on Software Testing, Verification, and Validation (ICST)*, pages 121–130, 2008.

[24] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: effective sampling for lightweight data-race detection. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 134–143, 2009.

[25] A. M. Memon. An event-flow model of GUI-based applications for testing. *Software Testing, Verification and Reliability*, 17(3):137–157, 2007.

[26] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 267–280, 2008.

[27] E. Mutlu, S. Tasiran, and B. Livshits. Detecting JavaScript races that matter. In *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2015.

[28] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *International Conference on Software Engineering (ICSE)*, pages 386–396, 2009.

[29] B. Petrov, M. Vechev, M. Sridharan, and J. Dolby. Race detection for web applications. In *Conference on Programming*

Language Design and Implementation (PLDI), pages 251–262, 2012.

[30] M. Pradel and T. R. Gross. Fully automatic and precise detection of thread safety violations. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 521–530, 2012.

[31] M. Pradel, M. Huggler, and T. R. Gross. Performance regression testing of concurrent classes. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 13–25, 2014.

[32] M. Pradel, P. Schuh, G. Necula, and K. Sen. EventBreak: Analyzing the responsiveness of user interfaces through performance-guided test generation. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 33–47, 2014.

[33] V. Raychev, M. Vechev, and M. Sridharan. Effective race detection for event-driven programs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 151–166, 2013.

[34] M. Samak and M. K. Ramanathan. Multithreaded test synthesis for deadlock detection. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 473–489, 2014.

[35] M. Samak and M. K. Ramanathan. Synthesizing tests for detecting atomicity violations. In *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 131–142, 2015.

[36] M. Samak, M. K. Ramanathan, and S. Jagannathan. Synthesizing racy tests. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 175–185, 2015.

[37] M. Samak, O. Tripp, and M. K. Ramanathan. Directed synthesis of failing concurrent executions. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 430–446, 2016.

[38] K. Sen. Effective random testing of concurrent programs. In *International Conference on Automated Software Engineering (ASE)*, pages 323–332, 2007.

[39] K. Sen. Race directed random testing of concurrent programs. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 11–21, 2008.

[40] O. Shacham, N. Bronson, A. Aiken, M. Sagiv, M. Vechev, and E. Yahav. Testing atomicity of composed concurrent operations. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 51–64, 2011.

[41] S. Tasharofi, M. Pradel, Y. Lin, and R. Johnson. Bita: Coverage-guided, automatic testing of actor programs. In *Conference on Automated Software Engineering (ASE)*, 2013.

[42] V. Terragni and S.-C. Cheung. Coverage-driven test code generation for concurrent classes. In *International Conference on Software Engineernig (ICSE)*, pages 1121–1132, 2016.

[43] S. Thummalapenta, K. V. Lakshmi, S. Sinha, N. Sinha, and S. Chandra. Guided test generation for web applications. In *International Conference on Software Engineering (ICSE)*, pages 162–171, 2013.

[44] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *International Conference on Automated Software Engineering (ASE)*, pages 203–232, 2003.

[45] W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[46] C. Wang, M. Said, and A. Gupta. Coverage guided systematic concurrency testing. In *International Conference on Software Engineering (ICSE)*, pages 221–230, 2011.

[47] A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for Java libraries. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 602–629, 2005.

[48] T. Yu and M. Pradel. Syncprof: Detecting, localizing, and optimizing synchronization bottlenecks. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 389–400, 2016.