Monitoring User Interactions for Supporting Failure Reproduction

Tobias Roehm, Nigar Gurbanova, Bernd Bruegge
Technische Universität München
Munich, Germany
{roehm, bruegge}@in.tum.de, nigarkurbanova@yahoo.com

Christophe Joubert
Prodevelop
Valencia, Spain
cjoubert@prodevelop.es

Walid Maalej
University of Hamburg
Hamburg, Germany
maalej@informatik.uni-hamburg.de

***Abstract*—The first step to comprehend and fix a software bug is usually to reproduce the corresponding failure. Reproducing a failure requires information about steps to reproduce, i.e. the steps necessary to make a failure occur in the development environment. In case of an application with a user interface, steps to reproduce consist of the interactions between a user and the application that precede the failure. Unfortunately, bug reports typically lack this information. Users are either unaware of its importance to developers, are unable to describe it, or simply do not have time to report it.**

In this paper, we present a simple but effective and resource efficient approach to monitor interactions between users and their applications selectively at a high level of abstraction, e.g. editing operations and commands. This minimizes the monitoring overhead and enables developers to analyze user interaction traces. We map monitored interactions to a taxonomy of user interactions to help developers comprehend user behavior. Further, we present the Timeline Tool that visualizes monitored interaction traces preceding failures. To evaluate our approach we conducted an experiment with 12 participants and asked them to reproduce bug reports from an open-source project. We found that developers are able to derive steps to reproduce from monitored interaction traces. In particular, inexperienced developers profit from the Timeline Tool, as they are able to reproduce failures that they cannot reproduce without it. The monitoring overhead is rather small (approx. 5 % CPU and 2-5% memory) and users feel it does not influence their work in a negative way.

***Index Terms*—Bug fixing, Failure reproduction, Steps to reproduce, User monitoring, Application instrumentation, Trace analysis, Program comprehension, Software maintenance, Software evolution**

I. Introduction

The first step for a developer fixing a bug is usually to reproduce the corresponding failure[1]. Reproducing a failure allows to confirm the existence of the corresponding bug, helps to comprehend the bug and identify its cause, and gives developers directions which artifacts to explore in their comprehension process, e.g. which part of source code to analyze or which part of the application to debug [18].

To reproduce a failure, developers need information about steps to reproduce, i.e. the steps necessary to make the failure occur on their development machines. In the case of an application with a user interface, steps to reproduce are the user interactions preceding a failure. Studies by Zimmermann et al. [23] and Laukkanen et al. [11] have shown that steps to reproduce represent important information for developers during maintenance tasks. These studies also found that steps to reproduce are often incomplete or incorrect in bug reports submitted by users, which makes failure reproduction difficult and time consuming.

We present an approach to monitor *high level* user interactions with *meaning* using application instrumentation. Examples of such interactions are editing operations or commands issued. As those types of user interactions are less frequent, the monitoring overhead is small and developers can manually "read" and analyze collected information. We map monitored user interactions to a taxonomy of interactions, giving monitored events a semantic meaning helping developers to comprehend user behavior. Further, we present the Timeline Tool that visualizes monitored user interaction traces. Our hypothesis is that developers can elicit steps to reproduce from traces of monitored, high level, meaningful user interactions that precede a failure. Our approach is complementary to existing record & replay approaches.

We investigated the impact of our approach on bug fixing tasks through an experiment comparing failure reproduction with and without our tool. Additionally, we evaluated the performance overhead introduced by our instrumentation by simulating user actions in a plain and instrumented version of a real world application. Further, we conducted a user study to collect user feedback on performance overhead.

The contribution of this paper is threefold. First, it introduces an approach to monitor high level user interactions, map them to a taxonomy, and exploit them for supporting failure reproduction. Second, it presents the Timeline Tool that visualizes monitored interactions . Third, it presents the design and results of an evaluation for the impact of our (and possibly other's) failure reproduction approaches using a combination of experiment, simulation, and user survey.

This paper is organized as follows. In Section II we discuss some background information. In Section III we describe our approach and the Timeline Tool. In Section IV and Section V we evaluate our approach from a developer perspective and the performance overhead introduced. After discussing important findings in Section VI, we review related work in Section VII and conclude in Section VIII.

[1]We use the term "failure" to denote a crash, an exception, or other wrong application behavior and the term "bug" to denote an algorithmic or coding mistake.

II. Failure Reproduction Spectrum

Before describing our approach in detail, we compare different perspectives of users and developers on failures and sketch a granularity spectrum of user interactions as reference framework.

A. Comparing User and Developer Perspectives

Users employ a software application in order to accomplish a certain task such as preparing slides for a presentation. Tasks consist of activities such as creating the slide structure, designing a slide, or checking the spelling. In order to conduct each activity, users interact with the user interface multiple times. When users encounter a failure, they theoretically know their current task, their current activity, and the UI elements they interacted with. Consequently, they can report this knowledge in bug reports like "Application crashed while designing a slide" or "Application crashed when clicking 'Save' button".

On the other hand, developers are interested in bugs as they want to fix them. Developers reproduce and analyze failures to confirm the existence of a bug and locate it in source code. Developers need information about the application and its execution in order to locate the code that should be changed.

When comparing the perspective of users and developers we observe a mismatch. Users cannot provide information needed by developers as they usually do not have information about internals of the application and its execution. In contrast, developers do not know how a user exactly used the application to trigger a failure. Developers usually close this gap in a time-consuming comprehension process. For instance, they try to reproduce a failure by interacting with the UI [18]. They are usually guided by steps to reproduce, i.e. the interactions that the user performed before the failure occurred. After reproducing the failure in the debugging environment, they analyze the code triggered by user interactions in order to locate the bug. The interactions of users with the user interface represent a "bridge" between user and developer perspective.

B. User Interaction Granularity Spectrum

We identified several granularity levels of user actions (see Table I). On the left hand side there are interactions with the hardware periphery such as "mouse click at position [237, 25]" or "key 'M' pressed". The next level of granularity is an interaction with a single widget such as entering text in a text field or clicking a button. A single widget interaction consist of multiple interactions with UI hardware such as multiple key presses on the keyboard to enter a text in a text field. Also, more detailed information about a user interaction is available on widget level. For example, a mouse click can now be distinguished to be a click on a button or the selection of a window part because the clicked artifact is known. The next level of granularity is an aggregation of several single widget interactions, which we call multi widget interaction, e.g. creation of a new presentation using a wizard. Several single widget or multi widget interactions represent a user activity such as designing a slide, which in turn is a step of a user task.

Table I
User Interaction Granularity Spectrum

Granularity	UI hardware interaction	Single widget interaction	Multi widget interaction	User activity	User task
Description	Interaction betw. user and UI hardware	Interaction betw. user and single widget	Aggregation of single widget interactions	A step of a user task	A task a user wants to accomplish
Examples	Mouse action, keyboard action	Click button, selection, enter text	Complete wizard, fill and submit form	Design a slide, check spelling	Create presentation

Level of abstraction, Meaning →

← Frequency and monitoring overhead

III. Our Approach

In this section we describe how we monitor user interactions and application events, map them to an interaction taxonomy, and visualize interaction traces in the Timeline Tool. We monitor interactions on single widget interaction level. They can be monitored directly in contrast to user activities and user tasks. Their frequency is much smaller than that of interactions with UI hardware. Hence, we hypothesize that developers can analyze them manually.
We implemented and tested our approach using a modeling and diagram-editing application, which we use as running example in this section.

A. Monitoring of User Interactions

We follow a general user involvement framework as presented in [12], [13]. Figure 1 gives an overview how we collect user interactions. Users interact with an application. Their interactions are detected and recorded by software sensors. Sensors are framework specific, live inside the application process and can monitor detailed user interactions. After detecting a user interaction, a sensor gathers all relevant information and sends it to the Client component. The Client component allows users to control the sensors, collects additional context information that is not specific to a particular sensor, and obfuscates monitored information to deal with privacy issues. Then, it passes the interaction information to the Server component running on a central server machine. The server component stores the information in a database. Finally, the Timeline Tool fetches a trace of events preceding an exception from the database, displays it, and allows developers to analyze the trace and navigate through it. This architecture allows minimizing computation and data storage on the user's machine by delegating to a central server.

We implemented the following sensors:

- Command Sensor
  This sensor monitors user commands such as Open, Close or Save diagrams and files by hooking as listener into the graphical user interface framework.
- GUI Part Sensor
  This sensor monitors the activation of a particular part

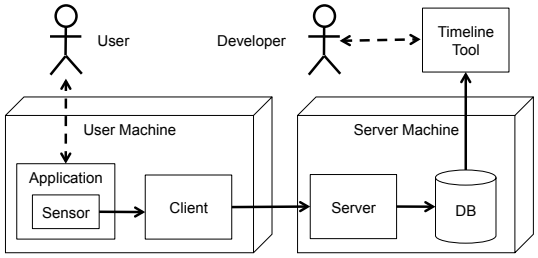

Figure 1. Overview of our Monitoring Approach

of the current window by hooking as listener into the window system.

- Diagram Sensor
  As we study a modeling and diagram-editing tool, this sensor monitors diagram manipulations such as adding or deleting nodes, adding or deleting edges, or entering text into diagram elements or their properties. It hooks as listener into the diagram modeling framework used.
- Application Sensor
  This sensor monitors start and shutdown of the application as well as exceptions occurring by hooking as listener into the application framework.

If a sensor observes a user interaction or an application event, it collects the following context information for each event:

- Event Type
  The type of event, e.g. NodeAdded or ActivateGuiPart. It corresponds to an event type in the taxonomy (see Section III-B).
- Timestamp
  The local system time when the event occurred.
- Artifact Type
  The artifact represents the object of a user interaction. For example when a user enters text, the artifact is the text field in which the text is entered. The artifact type thereby denotes the type of artifact.
- Artifact Id
  The artifact id uniquely identifies the specific artifact instance. In our text field example, it identifies the text field in which the user entered text.
- Machine Id
  The MAC address of the machine on which an event occurred. It allows distinguishing between events from different machines on the server side.

Additionally to this generic information, other information pieces that are specific for the type of event are collected. For example, the type of node in case of a NodeAdded event.

B. Taxonomy of User Interactions and Application Events

We developed a taxonomy of user interactions and application events that is shown in Figure 2. User interactions are divided into three subgroups. First, GuiNavigation interactions with the purpose to navigate within the user interface. For example, the ActivateGuiPart interaction represents activation of a certain part of the current window. Second, ExecuteCommand interactions denote execution of menu or shortcut key commands. Examples are saving a file, importing a file or creating a new diagram. Third, DiagramingEvents represent adding nodes, removing nodes, adding edges, or removing edges. The last interaction type is domain specific for diagram-editing applications. While GuiNavigation and ExecuteCommand interactions are single widget interactions, DiagramingEvents are usually multi widget interactions. Start, close, crash and exception events can occur in the application. Every sensor instantiates this taxonomy by assigning the corresponding event type to a monitored user interaction or application event.

C. Visualization of Interaction Traces in the Timeline Tool

The purpose of the Timeline Tool is to visualize traces of monitored user interactions together with application events. Figure 2, left side, shows a screenshot of the tool interface. A trace of monitored user interactions is shown in a timeline (Figure 2: 1) and single events are represented by diamonds placed on the timeline at the point in time when they occurred. Next to the diamond, the event type is rendered. The color of the diamond depends on the category of event. Each event is categorized as one of two categories: user interaction (i.e. an interaction between user and user interface) or application event (i.e. an event happening in the application). These categories correspond to the most abstract level in our event taxonomy and developers can filter to display or hide events from particular categories (Figure 2: 2). Further, developers can zoom in and out to inspect particular subparts of the trace in more detail. When a developer clicks on an event, detailed information about that particular event is displayed (Figure 2: 3 and 4). The information displayed is the information that was collected by software sensors. On the left side (Figure 2: 3), general event properties are displayed, while on the right side (Figure 2: 4) properties that are specific for the current event type are displayed in a key-value fashion. In the interaction trace shown in Figure 2, the NodeAdded interaction occurring at 10:20:27 is selected by the developer and its properties are shown on the bottom.

Overall, developers trying to reproduce a failure can inspect the user interactions preceding the failure and derive steps to reproduce by manually replaying all or a subset of monitored user interactions.

IV. Evaluation of Developer Perspective

In order to evaluate our approach and the Timeline Tool from the developer perspective, we conducted an experiment. We investigate in the following research questions:

- Can developers elicit steps to reproduce from monitored interaction traces visualized in the Timeline Tool (RQ1)?
- Does our approach enable developers to reproduce failures that could not be reproduced with textual bug reports because steps to reproduce are missing (RQ 2)?
- What is the difference between failure reproduction with the Timeline Tool and textual bug reports? (RQ 3)

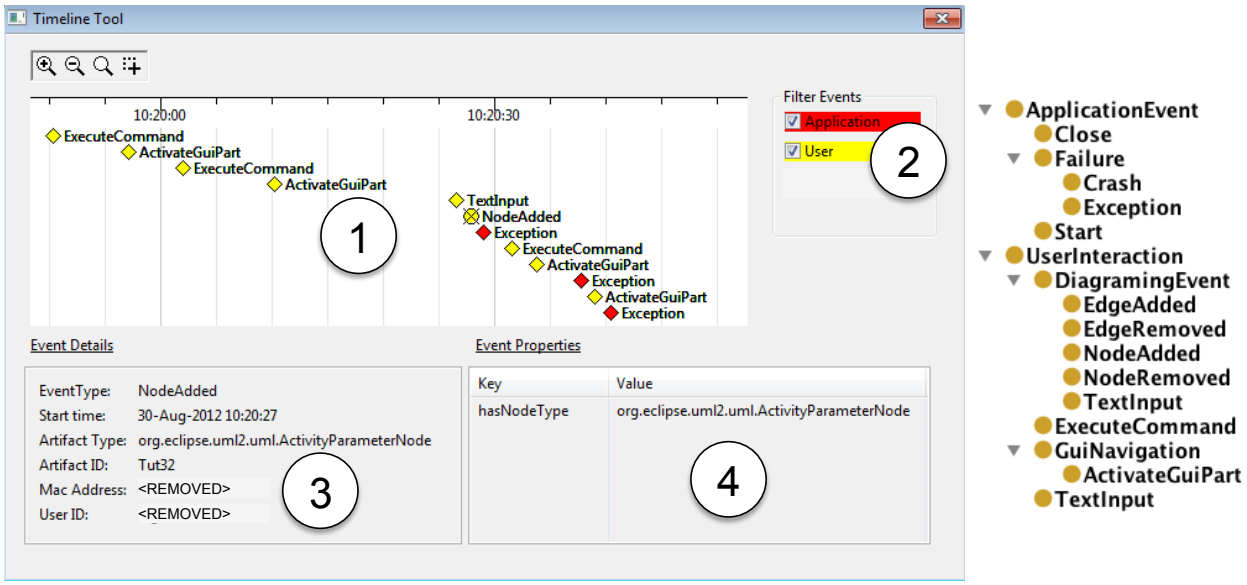

Figure 2. User Interface of Timeline Tool (Left) and Taxonomy of User Interactions and Application Events (Right)
Events in trace correspond to event types in taxonomy
1: Chronological view, 2: Event type filtering, 3: General event properties, 4: Event type specific properties

A. Experiment Design

We studied MOSKitt[2], an open-source modeling and diagram-editing tool, in our evaluation. MOSKitt is an RCP-based desktop application that supports modeling UML, BPMN, and Entity Relationship diagrams as well as capturing and managing textual software requirements. MOSKitt is constantly developed since 2007 by 27 contributing developers and has a size of 2 millions lines of code, which are mostly written in Java.

The sensors described in Section III-A were implemented as framework extensions of the Eclipse, RCP, SWT, or GMF frameworks and hook into the corresponding processing chains. This enables to install the sensors easily via the RCP update mechanism and to reuse the sensors for other applications using the same framework.

The main idea of our experiment is to give participants real bug reports from the MOSKitt bug repository and ask them to reproduce the corresponding failures in an application instance that we provided. We divided bug reports in two categories: bug reports that lack steps to reproduce ($\text{BugReport}_{\text{MissingSteps}}$) and bug reports that contain steps to reproduce ($\text{BugReport}_{\text{WithSteps}}$) (see Section IV-A for details on bug report selection). If steps to reproduce were not given in the bug report, we asked a developer to provide them to us.

Experimental Setting: We conducted two sub-experiments. Experiment 1 was conducted to investigate RQ1 and RQ3, i.e. to see if it is possible for developers to elicit steps to reproduce from the information presented in the Timeline Tool and compare failure reproduction with bug reports and interaction traces. Participants had to reproduce bug reports of the category $\text{BugReport}_{\text{WithSteps}}$. To avoid a dependency of the results on the order of tasks, the order was chosen randomly. Members of experiment group were given the Timeline Tool with corresponding interaction traces while members of control group were given traditional, textual bug reports.

Experiment 2 was conducted to investigate RQ1 and RQ2, i.e. to see if developers working with the Timeline Tool can reproduce failures that are lacking steps to reproduce in their textual bug reports. Each participant had to reproduce bug reports from category $\text{BugReport}_{\text{MissingSteps}}$. Members of control group were given traditional, textual bug reports while participants from experiment group had both traditional, textual bug reports and the Timeline Tool with the corresponding interaction trace.

We used a within subject design and divided participants into an experiment group and a control group randomly. After Experiment 1, the groups of participants were switched. Participants in the control group for Experiment 1 were assigned to the experiment group in Experiment 2 and vice versa. This design ensures that each participant works with the Timeline Tool. An overview of the experiment procedure and the failures and bug reports used is given in Table II.

At the beginning of each session, we introduced participants to the UML modeling feature of MOSKitt and the Timeline Tool by a short tutorial video. Then, each participant had to explore an example trace and answer seven questions about it to make sure that participants understood how to use the Timeline Tool.

Bug Report Selection and Trace Generation: In order to find suitable bug reports for our experiment, we analyzed the MOSKitt bug repository which contains around 19 000 bug reports. Developers of MOSKitt report that around 90 % of

[2]http://www.moskitt.org/

Table II
EXPERIMENT SETTING
FAILURES TO BE REPRODUCED AND MATERIAL PROVIDED
ORDER OF FAILURES IN EXPERIMENT 1 IS DETERMINED RANDOMLY

Failure	Material for Experiment Group	Material for Control Group
Experiment 1		
F1 (from BR1)	Timeline Tool with interaction trace	Traditional, textual bug report
F2 (from BR2)	Timeline Tool with interaction trace	Traditional, textual bug report
Experiment 2		
F3 (from BR3)	Timeline Tool with interaction trace and traditional, textual bug report	Traditional, textual bug report

the tickets do not contain information about steps to reproduce. We used the following inclusion criteria for selecting suitable bug reports. First, the reported failure had to be reproducible in single MOSKitt version, namely in MOSKitt 1.3.7. Second, the reported failure had to concern the UML modeling feature of MOSKitt to minimize the familiarization effort for participants not familiar with MOSKitt. Third, the reported failure had to trigger an exception that is visible to the user in form of an error dialog message or an entry in the error log. This requirement ensures that we can clearly determine if the failure was reproduced. Fourth, the bug report had to be in English language. After application of those criteria only four bug reports remained. There were two main reasons limiting suitable bug reports for our experiment. First, the bug repository is not only used for bug reports but also to capture feature requests and other information. Second, most bug reports were written in Spanish.

The four suitable bug reports were divided into two categories, namely $\text{BugReport}_{\text{WithSteps}}$ and $\text{BugReports}_{\text{MissingSteps}}$. When we could reproduce a failure based on the description in the bug report, it was categorized as $\text{BugReport}_{\text{WithSteps}}$. In other cases, it was categorized as $\text{BugReport}_{\text{MissingSteps}}$ and a developer of MOSKitt was asked to provide steps to reproduce. As one of the bug reports triggered several exceptions, it was used as example in the Timeline Tool tutorial and the other three bug reports BR1[3], BR2[4] and BR3[5] were used in our experiments (see Table III for details about the bug reports).

Unfortunately, we had no possibility to integrate our sensors into MOSKitt in a real usage setting. Therefore, we simulated the occurrence of each failure by executing the steps to reproduce in Table III and created an interaction trace using the MOSKitt sensors. This trace was shown to participants using the Timeline Tool.

Participants: We conducted the experiment with 12 participants (see Table IV for details). The participants were eight master students, two researches and two developers from the MOSKitt development team. Students and researchers did not have previous experience with MOSKitt. The self-assessment of UML experience - in our case domain knowledge of end users - differed between beginner and advanced with mode Intermediate (on a Beginner-Intermediate-Advanced scale). The frequency in which participants fix bugs in their daily work differed between daily to never with mode daily (on a Never - Once a month - Once a week - Daily scale).

[3]https://moskitt.gva.es/redmine/issues/165
[4]https://moskitt.gva.es/redmine/issues/138
[5]https://moskitt.gva.es/redmine/issues/139

B. Quantitative Results

The quantitative results of the experiment are summarized in Table V and we present the most important results in this section.

In Experiment 1, six of six members of the experiment group could reproduce the failures based on information from the Timeline Tool, as well as six out six participants from control group given textual bug reports. The average time needed by members of experiment group to reproduce failure 1 and failure 2 was 3:30 and 3:08 minutes, respectively. For members of the control group the time needed was 2:49 and 2:05 minutes, respectively. Further, four of six members of the experiment group in Experiment 2 were able to reproduce the failure with information from the Timeline Tool and textual bug reports.
We conclude that developers can elicit steps to reproduce from the information presented by the Timeline Tool. As we did only a brief introduction of the Timeline Tool, a possible explanation for the additional time needed by members of experiment group could be that they needed some time to familiarize with the Timeline Tool. This effect has to be further investigated.

In Experiment 2, four of six members of the experiment group were able to reproduce the failure with interaction traces visualized in the Timeline Tool and textual bug reports. The two participants from the experiment group that could not reproduce the failure were P10 and P12. Participant P12 could not reproduce the failure because he was missing some actions in the trace. Participant P10 could not reproduce the failure because he was exploring the trace backwards, i. e. right to left instead of left to right. In contrast, only one of six members of the control group was able to reproduce the failure with the textual bug report. But this was the developer that originally fixed the bug and hence his success is biased and cannot be generalized. Members of the experiment group needed 6:28 min in average while members of the control group gave up after 6:14 min in average.
We conclude that the Timeline Tool enables developers to reproduce failures whose bug reports lack steps to reproduce. This is a major improvement as developers depending on the textual bug reports could not reproduce those failures.

C. Qualitative Findings

Here we summarize the findings from observing participants during their work with the Timeline Tool.

Table III
BUG REPORTS USED IN EXPERIMENT

Bug Report	Category	Title and Description of Bug Report	Steps to reproduce
BR1	With Steps	Error when assigning a StateMachine to a SubmachineState: When adding a "Submachine State", Moskitt throws an exception: Unhandled event loop exception java.lang.StackOverflowError Then, the submachine is shown into the diagram, but it cannot be deleted.	1) Create new 'MOSKitt' project. 2) Create new UML2 diagram of type 'UML State machine'. 3) Create a new 'Submachine State' element 4) Select the parent StateMachine as the referenced state machine. -> StackOverflow error is thrown and diagram is no longer editable.
BR2	With Steps	[Use Case] Error when create an extension point into a Use Case figure: [Use Case] Error when create an extension point into a Use Case figure	0) Open the 'Error log' view of MOSKitt. 1) Create a new 'MOSKitt' project. 2) Create new UML2 diagram of type 'UML UseCase'. 3) Create a 'UseCase' element . 4) Create an 'ExtensionPoint' inside the use case . -> An error appears in the error log view with a stack trace.
BR3	Missing Steps	[Statemachine] Error when create a State Submachine: [Statemachine] Error when create an State Submachine	0) Open the 'Error log' view of MOSKitt. 1) Create a new 'MOSKitt' project. 2) Create new UML2 diagram of type 'UML State machine'. 3) Create another UML2 diagram of type 'UML State machine'. 4) Save both diagrams and keep them open. 5) In the first diagram, create a 'Submachine State' . When prompted to select a State Machine, select the one from the other diagram. -> A StackOverflow error happened and MOSKitt needs to be restarted to continue working.

Table IV
EXPERIMENT PARTICIPANTS
S = STUDENT, R = RESEARCHER, D = DEVELOPER

Participant	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
Position	S	S	S	S	S	S	S	S	R	R	D	D
Gender	M	M	M	M	F	F	M	F	F	M	M	M
MOSKitt Experience	No	No	No	No	No	No	No	No	No	No	Yes	Yes
UML Experience	Interm.	Interm.	Beg.	Beg.	Interm.	Interm.	Adv.	Interm.	Interm.	Adv.	Adv.	Adv.
Bug Fixing Frequency	Daily	Weekly	Daily	Daily	Monthly	Never	Daily	Daily	Monthly	Weekly	Daily	Monthly

Table V
QUANTITATIVE EXPERIMENT RESULTS
REPR.=FAILURE REPRODUCED? (YES/ NO)

Participant	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
Experiment 1												
Group	Contr.	Exp.	Exp.	Contr.	Contr.	Exp.	Contr.	Exp.	Exp.	Contr.	Exp.	Contr.
Order	F1,F2	F1,F2	F1,F2	F2,F1	F1,F2	F2,F1	F1,F2	F1,F2	F2,F1	F2,F1	F2,F1	F1,F2
F1 (BR1) Time	03:00	5:50	06:00	03:00	03:30	02:00	03:20	03:50	01:30	01:55	01:50	02:10
F1 (BR1) Repr.	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
F2 (BR2) Time	1:15	2:00	03:00	03:00	04:00	05:30	01:25	03:10	02:40	02:10	02:30	00:40
F2 (BR2) Repr.	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Experiment 2												
Group	Exp.	Contr.	Contr.	Exp.	Exp.	Contr.	Exp.	Contr.	Contr.	Exp.	Contr.	Exp.
F3 (BR3) Time	07:30	06:00	08:00	10:00	07:00	07:45	05:00	07:40	03:10	04:50	04:50	04:30
F3 (BR3) Repr.	Y	N	N	Y	Y	N	Y	N	N	N	(Y)	N

Direction of Trace Exploration: We found two trace exploration strategies. Nine participants explored the trace chronologically, i.e. left to right, while three participants explored the trace backwards, i.e. right to left. The backwards strategy seems to resemble the analysis of a stack trace, where an exception is analyzed by exploring executed methods starting with the latest method called (top down in the stack trace). In order to be able to reproduce failures using the Timeline Tool, participants had to work chronologically. Participants were not told this before. Also, they did not know that the Timeline Tool presents minimal reproduction traces, i.e. all user interactions of a trace have to be reproduced to make an exception occur. Hence, they were probably trying to identify the first user interaction necessary for failure reproduction.

Participant Feedback: We asked participants about their opinion about the Timeline Tool and its impact on failure reproduction tasks in a short questionnaire. Eleven participants agreed (8) or strongly agreed (3) that the meaning of information presented in the Timeline Tool is clear and easy to understand. Eleven participants agreed (3) or strongly agreed (8) that the Timeline Tool is helpful when reproducing failures. Similarly, eleven participants agreed (6) or strongly agreed (5) that it is clear what the user did by analyzing the trace in the Timeline Tool. Two participants preferred textual bug reports to the Timeline Tool, given that both contain the same information, while eight participants preferred the Timeline Tool.

Impact for Developers and Students: When looking at the results, we see a difference between students and developers. As the students were all master students and most of them work as developers, we can assume that the students mimic new developers without experience with MOSKitt. Looking at the results of the students in experiment 2, we found that all four students in the experiment group were able to reproduce the failure while all four students in the control group were not able to reproduce it. Consequently, we conclude that the Timeline Tool has a big impact on failure reproduction of developers unfamiliar with an application. Looking at the results of MOSKitt developers in experiment 2, the results differ. While P12 working with the Timeline Tool could not reproduce the failure, P11 working with the textual bug report could reproduce it. But there are effects that doubt the generalizability of those results. P12 did not work in a chronological order with the Timeline Tool and P11 was the developer fixing the corresponding bug for F3. Because of these effects and the limited number of developers in the experiment further investigation is necessary. Both developers strongly agreed that the Timeline Tool is helpful in failure reproduction.

D. Limitations

Our experiment setup has three main limitations that we discuss below. First, the interaction traces presented in the Timeline Tool were generated manually. Thereby, we followed the steps to reproduce elicited from the textual bug report or given by MOSKitt developers. Hence, we assume that the manual generation of traces will not influence the results of the experiment completely. Second, the limited number of failures that participants had to reproduce. We could find only four bug reports fulfilling our criteria. By choosing bug reports from a real bug repository, we can assume a good practical relevance of the bug reports studied. But the generalizability of our approach has to be further investigated. Third, participants were asked to reproduce failures on a laptop with the same environment and the same MOSKitt instance that was also used to record the interaction traces. As this is usually not the case in reality, further research is necessary to discriminate failures that only occur in special environments and to extend our approach to deal with heterogeneity of environments.

V. Evaluation of Performance Overhead

In order to evaluate the performance overhead introduced by our sensors, we conducted a simulation study and a user survey that we describe in this section. We investigate the following research questions.

- How big is the performance overhead introduced by our sensors in terms of time and memory (RQ4)?
- How do users perceive the performance overhead (RQ5)?

MOSKitt, a modeling and diagram-editing tool was used as case study application in this evaluation. It is the same application as used in the evaluation from developer perspective and is described in more detail in Section IV.

A. Design

Here we present the design of the simulation and user survey.

Simulation: In order to study RQ4 and measure the performance overhead introduced by our sensors, we conducted a simulation. We generated sequences of diagram manipulation actions automatically and used them to simulate user interactions. These generated sequences were injected in a plain MOSKitt instance as well as in an instrumented MOSKitt instance. We measured and compared differences regarding time and memory consumption in order to evaluate the performance overhead introduced by our diagram sensor, i.e. the GMF-based sensor monitoring diagram manipulations.

The types of diagram manipulations simulated are Create and Delete Project (CP, DP), Create Model (CM), Create/Update/Delete Element (CE, UE, DE), Execute Transformation (ET), Change Size (CS), Move Figure (MF), Switch Editor (SW), Undo (UN) and Redo (RE). The occurrence frequency of each manipulation type is determined by parameters. As the order of manipulations is not arbitrary, e.g. an element can only be created when an umbrella model exists, a state machine is used to determine the next manipulation. A simplified version (showing only a subset of all diagram manipulations) is shown in Figure 3. As diagram elements are parts of models and models are organized in projects, the states InElement, InModel, and InProject represent the current focus. During creation of the manipulation sequence, the diagram is in a certain state and randomly selects a manipulation that is possible in the current state. If a manipulation is feasible in a

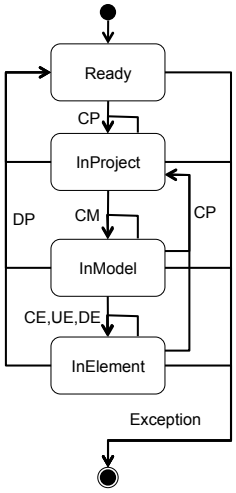

Figure 3. State Machine Used to Generate Diagram Manipulation Sequences C=Create, D=Delete, U=Update, P=Project, M=Model, E=Element Simplified version

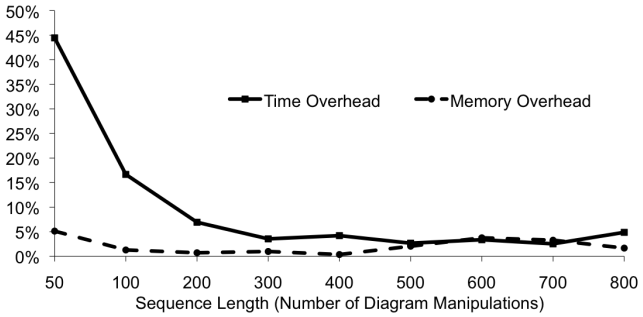

Figure 4. Performance Overhead of Instrumentation

state, it is represented as an edge leaving the state node in the state machine. Projects can be created and deleted at any time, as well as exceptions can occur at any time. Between any two manipulations, a time break of 0, 1 or 2 seconds is inserted randomly. The resulting sequence of diagram manipulations is stored in a file.

The generated sequences of diagram manipulations are injected in a plain MOSKitt instance and in a MOSKitt instance instrumented with our sensors. The total time needed for the execution of all manipulations is measured and the time per manipulation is calculated by dividing measured total time by the number of diagram manipulations. Further, the current RAM consumption of the application is measured using nmon utility tool periodically and the average of a simulated session is calculated. Both metrics are compared for the plain and the instrumented MOSKitt instance. A machine with similar configuration as the typical user machine is used to run the simulations, i.e. a laptop running Ubuntu Linux equipped with a Intel Core 2 Duo processor (2 cores at 2.00 GHz) and 3 GB RAM.

User Survey: We conducted a user survey in order to investigate RQ5 and obtain user feedback on the performance overhead introduced by our sensors. We asked six MOSKitt users to work with a MOSKitt version instrumented with our sensors for two weeks and report their experiences in a questionnaire. As this study was part of a larger evaluation study in the FastFix project [16], the instrumentation consisted of our sensors as described in Section III-A and additionally sensors for a record & replay approach that monitored user interactions on a lower level of granularity. User feedback was collected using an anonymized, web-based questionnaire. Users were asked the following questions: "The application behaves the same way with as without the sensors." (Q1, agreement on a 5 point Likert scale from Strongly agree to Strongly disagree), "Did you notice changes in performance (e.g. longer response time) since you started using MOSKitt with sensors?" (Q2, Yes/ No) and "If yes, do you agree with the following statement - The performance changes introduced by the sensors are tolerable and do not hinder my work." (Q3, Likert scale like Q1).

B. Results

Here we present the results of both the simulation and the user survey.

Simulation: We simulated 9 user sessions ranging from 50 to 800 diagram manipulations. Figure 4 shows the time and memory overhead between plain and instrumented MOSKitt version. The results show that the time overhead is big for short sequences and then converges to approx. 5 %. We expected a big overhead for short sequences as the sensor initialization overhead is proportionally large for sequences with few diagram manipulations. The average memory overhead introduced was 2-5 %. Concluding, we found that our diagram sensor introduces a small overhead regarding time and memory consumption.

User Survey: Table VI gives an overview over participants of the user survey and their answers to the questions. Users used the instrumented version of MOSKitt for 5 days in average and performed 6 sessions on average during this time. Five users agreed or strongly agreed that they do not perceive a difference in the behavior of MOSKitt as compared to MOSKitt without our sensors. User U2 disagreed and perceived a performance overhead, but judged it to not hinder his or her daily work. Concluding, we found that our sensors do not introduce a performance overhead that hinders users in their daily work.

C. Limitations

As the performance of users was not considered during the user survey, we think there is no Hawthorne effect influencing the behavior of users and consequently our results. Our evaluation has three limitations. First, only diagram manipulations are simulated and consequently we evaluate the overhead of the diagram sensor and the processing and storage logic with our simulation. As diagram manipulations are frequent user interactions for a diagram manipulation application, we expect the results to generalize to other actions. Second,

Table VI
USER SURVEY PARTICIPANTS AND RESULTS

Participant	U1	U2	U3	U4	U5	U6
Days of Use	8	3	8	7	1	3
Sessions	8	3	12	7	4	3
Usual Usage Freq.	Never	Never	Once a day	Once a day	Several times a day	Once a week
Q1: Same behavior	Agree	Disagr.	Agree	Strong Agree	Agree	Agree
Q2: Perf. changes	No	Yes	No	No	No	No
Q3: No hindrance	-	Agree	-	-	-	-

the generated sequences of diagram manipulations simulate user behavior but might deviate from real user behavior. In order to minimize this limitation, the order of manipulations in generated sequences is determined randomly guided by the state machine and time breaks between two manipulations are introduced. Third, an additional record & replay-instrumentation was present in parallel to our sensors during the user study. Consequently, we cannot determine the extent to which our sensors are responsible for potential performance degradation.

VI. DISCUSSION

We discuss the applicability of our approach, important observations and findings in this section.

As we monitor and analyze user interactions, our approach is applicable to software applications with a user interface. We implemented and tested our approach with a traditional desktop application that is controlled using mouse and keyboard. But we expect our approach to be adaptable without big effort to other UI driven applications running on smart phones or tablets. In order to apply our approach to other applications, the sensors and the taxonomy have to be extended accordingly. The possibility and effort to implement sensors for an application depends on the existence of monitoring components, the hooks provided by the frameworks used, and the availability of source code. The domain-independent part of the taxonomy such as standard commands and UI navigation actions can be reused while domain-dependent interactions have to be added to the taxonomy. For employing our approach, two main scenarios are possible. First, augmenting an existing, textual bug report with monitored interaction traces. Second, in cases where no bug report exists, automatically generating a bug report containing a monitored interaction trace.

Considering user interactions only is not enough to reliably reproduce failures occurring in the field. First, other types of input such as data loaded from a file alters the state of an application and may cause a failure. Second, a failure may depend on a certain environment such as a certain library version. Consequently, it has to be investigated what subset of failures can be tackled with our approach and how the mentioned issues can be tackled.

Existing record & replay approaches focus on collecting application execution information such as window system events and UI hardware interactions. We share the goal to reproduce failures by monitoring user interactions, but propose to selectively monitor user interactions at a high level of granularity. Focusing on important interactions allows to minimize performance overhead introduced, reduces the number of interactions developers have to inspect during reproduction and thereby enables manual analysis by developers. We do not replay the interactions we monitored but present them as traces to developers for manual analysis, thereby avoiding problems sometimes associated with replaying monitored user interactions.

VII. RELATED WORK

Maalej et al. [12] and Maalej and Pagano [13] discuss how feedback and input of users can be considered in software development in general. Maalej et al. [12] also present a classification of user input types and our approach falls in their category "Pull communication & Implicit Feedback". In the following, we review related work in four areas below.

Record & Replay Approaches for User Interactions: Approaches to capture and replay user interactions have been developed by other researchers. Herbold et al. [7] monitor the messages between GUI objects in applications implemented using Microsoft Foundation Classes as target platform. Those message are triggered by user interactions like mouse clicks or key presses. Steven et al. [20] present jRapture, an approach to monitor and replay AWT and Swing events generated upon interactions of users with the user interface by a wrapper. We monitor user interactions selectively at a higher level of granularity.

Automated Failure Reporting: Murphy [14] discusses automated failure reporting and stresses the importance of user input and the events preceding a failure. Glerum et al. [6] present WER, the automated failure reporting tool used by Microsoft, and their experiences to collect and analyze failure data. Further tools to report failures and context information via the Internet have been developed. Apple Crash Reporter [2], BugBuddy[6], Mozilla Talkback[7], and Google Breakpad[8] collect memory dumps and report to central failure repositories. These approaches do not capture user interactions preceding failures.

Monitoring of User Interactions: Monitoring user actions has a long tradition. Hilbert and Redmiles [8] review the state of the art of monitoring user interface events. Saito et al. [19] observe window switches and summarize user tasks and compare user behavior. Several authors present approaches that help in usability testing in general [21], [1], usability testing of web pages [4] and usability testing of mobile apps [10], [17]. These approaches monitor user actions and exploit them to comprehend user behavior, but they do not deal with failure reproduction.

[6]http://directory.fsf.org/wiki/Bug-buddy
[7]http://talkback.mozilla.org
[8]http://code.google.com/p/google-breakpad/

Reproduction of Field Failures: Other researchers have studied reproduction of failures occurring in the field by monitoring the dynamic execution of applications. Orso and Kennedy [15] selectively monitor the executed methods, field accesses and exceptions. Jin and Orso [9] support in-house reproduction of field failures based on software execution information such as call traces. Similarly, Artzi et al. [3] automatically generate multiple unit tests to reproduce a given program failure. Bell et al. [5] capture and replay all sources of non-determinism to an application. Tucek et al. [22] employ lightweight monitoring to detect failures and collect additional information by re-execution on the user's machine. All those approaches collect information about application execution while we monitor user interactions.

VIII. Conclusion

In this paper we presented an approach to monitor user interactions at a higher level of abstraction than state of the art tools and establish a semantic meaning for monitored interactions by mapping them to a taxonomy of interactions. We also presented a tool that visualizes interaction traces preceding failures. By evaluating the impact on failure reproduction tasks using an experiment, we found that developers are able to derive steps to reproduce from monitored and visualized interaction traces. Inexperienced developers were able to reproduce failures that they could not reproduce before. By evaluating the performance overhead introduced by our instrumentation using a simulation and a user survey, we found that the overhead is small and does not hinder users in their work.

Four main future directions are necessary to improve our approach and understand its impact on maintenance tasks. First, the experiment should be replicated with more experienced developers, other applications, more failures, real interaction traces, and heterogeneous reproduction environments to tackle current limitations. Second, it should be investigated how developers can deal with large interaction traces as generated by users in their daily work and corresponding mechanisms such as filtering algorithms should be designed. Third, the Timeline Tool should be integrated into existing development and maintenance environments such as code editors, debuggers, and issue trackers. Fourth, the issue of privacy and its implications on interaction monitoring should be investigated.

Acknowledgements

We thank Dennis Pagano, Rebecca Tiarks, and anonymous ICPC reviewers for feedback, Marc Gil, Miguel Llácer, and Javier Cano for their support to collect data, and all participants of our evaluation studies. This work was supported by the European Commission under grant no. FP7-258109 (FastFix project) and partially supported by the Spanish MEC INNCORPORA-PTQ 2011 program.

References

[1] D. Akers, R. Jeffries, M. Simpson, and T. Winograd. Backtracking events as indicators of usability problems in creation-oriented applications. *ACM Transactions on Computer-Human Interaction*, 19(2):1–40, 2012.

[2] Apple Inc. CrashReporter. Technical Report TN2123. Technical report, 2004.

[3] S. Artzi, S. Kim, and M. D. Ernst. ReCrash: Making software failures reproducible by preserving object states. In *ECOOP 2008 - Object-Oriented Programming*, volume 5142 of *LNCS*, pages 542–565. Springer, 2008.

[4] R. Atterer, M. Wnuk, and A. Schmidt. Knowing the user's every move - User activity tracking for website usability evaluation and implicit interaction. In *Proc. of the 15th Int. Conf. on World Wide Web*, pages 203–212. ACM, 2006.

[5] J. Bell, N. Sarda, and G. Kaiser. Chronicler: Lightweight recording to reproduce field failures. In *ICSE 2013 Proceedings, To Appear.*, 2013.

[6] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: Ten years of implementation and experience. In *Proc. of the ACM SIGOPS 22nd Symp. on Operating Systems Principles*, pages 103–116. ACM, 2009.

[7] S. Herbold, J. Grabowski, S. Waack, and U. Bünting. Improved bug reporting and reproduction through non-intrusive GUI usage monitoring and automated replaying. In *Fourth Int. Conf. on Software Testing, Verification and Validation Workshops*, pages 232–241. IEEE, 2011.

[8] D. M. Hilbert and D. F. Redmiles. Extracting usability information from user interface events. *ACM Computing Surveys*, 32(4):384–421, 2000.

[9] W. Jin and A. Orso. BugRedux: Reproducing field failures for in-house debugging. In *ICSE 2012 Proceedings*, pages 474–484. IEEE, 2012.

[10] D. Kim and K. Lee. Development of interactive logger for understanding user's interaction with mobile phone. In *Human-Computer Interaction. Interaction Platforms and Techniques*, volume 4551 of *LNCS*, pages 394–400. Springer, 2007.

[11] E. I. Laukkanen and M. V. Mantyla. Survey reproduction of defect reporting in industrial software development. In *2011 Int. Symp. on Empirical Software Engineering and Measurement*, pages 197–206. IEEE, 2011.

[12] W. Maalej, H. Happel, and A. Rashid. When users become collaborators: Towards continuous and context-aware user input. *Proc. of the 24th ACM SIGPLAN Conf. on Object Oriented Programming Systems Languages and Applications*, pages 981–990, 2009.

[13] W. Maalej and D. Pagano. On the socialness of software. In *Ninth IEEE Int. Conf. on Dependable, Autonomic and Secure Computing*, DASC, pages 864–871. IEEE, 2011.

[14] B. Murphy. Automating software failure reporting. *Queue*, 2(8):42–48, 2004.

[15] A. Orso and B. Kennedy. Selective capture and replay of program executions. In *Proc. of the 3rd Int. Workshop on Dynamic Analysis*, pages 1–7. ACM, 2005.

[16] D. Pagano, M. Juan, A. Bagnato, T. Roehm, B. Bruegge, and W. Maalej. FastFix: Monitoring control for remote software maintenance. In *ICSE 2012 Proceedings*, pages 1437–1438. IEEE, 2012.

[17] F. Paternò, A. Russino, C. Santoro, and V. G. Moruzzi. Remote evaluation of mobile applications. In *Task Models and Diagrams for User Interface Design*, volume 4849 of *LNCS*, pages 155–169. Springer, 2007.

[18] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej. How do professional developers comprehend software? In *ICSE 2012 Proceedings*, pages 255–265. IEEE, 2012.

[19] R. Saito, T. Kuboyama, Y. Yamakawa, and H. Yasuda. Understanding user behavior through summarization of window transition logs. In *Databases in Networked Information Systems*, volume 7108 of *LNCS*, pages 162–178. Springer, 2011.

[20] J. Steven, P. Chandra, B. Fleck, and A. Podgurski. jRapture: A capture/replay tool for observation-based testing. *ACM SIGSOFT Softw. Eng. Notes*, 25(5):158–167, 2000.

[21] Y. Tao. Capturing user interface events with aspects. In *Human-Computer Interaction. HCI Applications and Services*, volume 4553 of *LNCS*, pages 1170–1179. Springer, 2007.

[22] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: Diagnosing production run failures at the user's site. In *Proc. of 21st ACM SIGOPS Symp. on Operating System Principles*, 2007.

[23] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schröter, and C. Weiss. What makes a good bug report? *IEEE TSE*, 36(5):618–643, 2010.