Automated Extraction of Failure Reproduction Steps from User Interaction Traces

Tobias Roehm
Technische Universität München
Munich, Germany
roehm@in.tum.de

Stefan Nosovic
Technische Universität München
Munich, Germany
stefan.nosovic@in.tum.de

Bernd Bruegge
Technische Universität München
Munich, Germany
bruegge@in.tum.de

***Abstract*—Bug reports submitted by users and crash reports collected by crash reporting tools often lack information about reproduction steps, i.e. the steps necessary to reproduce a failure. Hence, developers have difficulties to reproduce field failures and might not be able to fix all reported bugs.**

We present an approach to automatically extract failure reproduction steps from user interaction traces. We capture interactions between a user and a WIMP GUI using a capture/replay tool. Then, we extract the minimal, failure-inducing subsequence of captured interaction traces. We use three algorithms to perform this extraction: Delta Debugging, Sequential Pattern Mining, and a combination of both. Delta Debugging automatically replays subsequences of an interaction trace to identify the minimal, failure-inducing subsequence. Sequential Pattern Mining identifies the common subsequence in interaction traces inducing the same failure.

We evaluated our approach in a case study. We injected four bugs to the code of a mail client application, collected interaction traces of five participants trying to find these bugs, and applied the extraction algorithms. Delta Debugging extracted the minimal, failure-inducing interaction subsequence in 90% of all cases. Sequential Pattern Mining produced failure-inducing interaction sequences in 75% of all cases and removed on average 93% of unnecessary interactions, potentially enabling manual analysis by developers. Both algorithms complement each other because they are applicable in different contexts and can be combined to improve performance.

***Index Terms*—bug fixing, failure reproduction, field failures, bug reporting, reproduction steps, steps to reproduce, user interactions, delta debugging, sequential pattern mining, capture/replay, record/replay, software maintenance, software evolution**

I. Introduction

> "93% of 1,477 participating software developers encounter problems when reproducing failures due to missing knowledge at least weekly, 70% daily." [14]

Reproducing a failure allows software developers to verify that a problem exists and constitutes a first step towards identification of the failure cause and fixing the bug causing the failure. To reproduce a particular failure, developers need information about reproduction steps, i.e. the steps necessary to trigger the failure, and information about the failure environment, i.e. the setting in which the failure occurs [21]. Because developers are usually not present when users employ an application in the field, they do not have first hand information about reproduction steps and failure environments. Hence, developers often face problems when reproducing failures: In a survey [14], 93% of 1,477 participating developers reported that they face problems due to missing knowledge when reproducing failures at least weekly, 70% faced such problems on a daily basis.

Developers usually use two ways to acquire information about reproduction steps and failure environments: bug reporting and collection of field data. During bug reporting, users who experience a failure report the failure together with its reproduction steps and its failure environment. Research has shown that this approach has some challenges because bug reports submitted by users often do not contain reproduction steps or their reproduction steps are wrong or incomplete ([13], [23]). Alternatively, developers can instrument applications and collect field data, i.e. data about the runtime behavior and runtime environment of deployed programs [16]. Such an approach usually generates a huge amount of trace data and it is usually not possible for developers to analyze such datasets manually. Therefore, reproduction steps and information about the failure environment should be extracted automatically.

We present an approach to collect field data automatically and extract reproduction steps from captured data. Our vision is to automatically identify reproduction steps as they would be described by users, thereby relieving users from manually describing them. We use an industrial capture/replay tool to capture interactions between a user and a graphical user interface using the "windows, icons, menus, pointer" paradigm (WIMP GUI). Then, we use three algorithms to extract reproduction steps from captured interaction traces: Delta Debugging [22] (DD), Sequential Pattern Mining [15] (SPM), and a combination of both (SPM+DD). DD operates on a single interaction trace and systematically replays subsequences of it to identify the minimal, failure-inducing subsequence. SPM operates on a set of interaction traces which trigger the same failure and extracts a common subsequence, eliminating interactions which are irrelevant for failure reproduction and approximating the minimal, failure-inducing sequence. We also combined both algorithms and study whether the combination improves performance.
Our approach is inspired by Zimmermann et al. [23] who suggest to "use capture/replay tools to provide reproduction steps automatically". In contrast to most related work capturing field data on code execution level, we target interactions of

978-1-4799-8469-5/15 © 2015 IEEE

Accepted for publication by IEEE. © 2015 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

users with a WIMP GUI. We hypothesize that this approach introduces less performance overhead, produces a smaller amount of monitoring data, relieves users from manually describing reproduction steps, and allows developers to discuss reproductions steps with users. In previous work [17], we presented an approach which captures user interactions with a WIMP GUI and presents them to developers during failure reproduction. This work improves our previous work by minimizing the captured interaction trace, i.e. removing captured interactions which are not necessary for failure reproduction.

We evaluated our approach and compared the extraction algorithms in a case study. We injected four bugs in the source code of an e-mail application. To collect interaction traces, we asked five participants to "hunt" for the bugs, i.e. to interact with the application until a failure is triggered. Interactions between participants and the WIMP GUI were captured by an industrial capture/replay tool. Then, the algorithms DD, SPM, and SPM+DD were run on this dataset for evaluation purposes.

The contributions of this paper are threefold: First, we apply the Delta Debugging algorithm to traces of user-GUI interactions. Second, we propose to use Sequential Pattern Mining to extract reproduction steps from interaction traces. And third, we evaluate and compare both algorithms and their combination in a case study. Overall, we illustrate how reproduction steps can be extracted automatically from captured interaction traces.

This paper is organized as follows: Section II defines terminology and important concepts and Section III reviews related work. Section IV describes how we capture user-GUI interactions and extract reproduction steps from them. Section V presents design and results of the case study. Section VI discusses case study results as well as limitations and implications of both the case study and the approach. Finally, Section VII concludes the paper.

II. Background

In this section we define terminology and properties of interaction sequences. We use the term "failure" to denote an exception or a crash of the target application and the term "bug" to denote an algorithmic or coding mistake causing a failure. Further, we use the term "bug report" to denote a report containing information about a instance of failure.

A user interaction denotes any interaction between a user and a WIMP GUI that can be captured by a capture/replay tool. For example, button clicks, selection of main menu actions, clicks on toolbar icons, or text entry in text fields. Because each interaction occurs at a given time, captured interactions form a sequence, i.e. a number of interactions totally ordered by time. We use the term "interaction trace" to denote the sequence of user interactions captured by the capture/replay tool. Otherwise we use the term "interaction sequence". Each interaction trace is also an interaction sequence. By removing an arbitrary number of interactions from a given interaction sequence, a sub-sequence can be created.

A given interaction sequence can have different properties. If a given interaction sequence can be replayed successfully by the capture/replay tool, we call it "replayable" (*replayability property*). Successful replay means that all its interactions can be simulated by the capture/replay tool without triggering a replay failure. A given interaction sequence is "failure-inducing" when its execution - either manually by a user/ developer or automatically by a capture/replay tool - triggers a failure (*failure-induction property*). Failure-inducing interaction sequences may have another property: minimality (*minimality property*). A given, failure-inducing interaction sequence is minimal if no subsequence exists which also triggers the same failure. Our goal is to automatically identify the minimal, failure-inducing subsequence of an interaction trace. We define "reproduction steps of a failure" as a minimal, failure-inducing interaction sequence that triggers the failure. Reproduction steps of a given failure f can be unique or non-unique (*uniqueness property*). We call reproduction steps of a failure unique if no other reproduction steps for the same failure exist. For example, a failure triggered by special characters in text input does not have unique reproduction steps because entering different special characters lead to the same failure, i.e. multiple interaction sequences triggering the failure exist.

III. Related Work

This section reviews related work aiming to minimize failure-inducing input as well as approaches to support developers in reproducing field failures.

Minimizing Failure-Inducing Input: In their seminal paper [22], Zeller and Hildebrand apply DD to interactions of users with the Mozilla browser for one particular bug. In this spirit, we apply the DD algorithm to traces of user-GUI interactions, replicating their case study and confirming their results. We are not aware of further related work which applies DD to traces of user-GUI interactions. Hsu et al. [10] collect traces of branching events and method calls at runtime and use the BIDE algorithm to identify bug signatures, i.e. patterns which are frequently part of failing traces. We use a similar approach, but we apply it to user interactions.

Minimization approaches which take a failure-inducing trace as input and automatically identify the minimal subpart required for failure induction have been proposed for several types of input such as textual input [22], file system and stream operations [5], and object interactions [4]. In contrast to those approaches, we apply minimization to traces of user interactions.

Reproducing Field Failures: Clause and Orso [5] present an approach to debugging field failures by recording file system and stream actions, minimizing captured traces, and replaying them. Their goal is similar to ours but our approach operates on user interactions.

Several approaches to reproduce field failures use a combination of field data collection and in-house execution synthetization: BugRedux [11] is a general framework to collect failing field executions and to synthesize executions which trigger the same failure. F^3 [12] extends BugRedux to synthesize passing and failing executions and exploits them to identify potentially

faulty program entities. MIMIC [24] extends F^3 by comparing a model of correct behavior to failing executions and identifying violations of the model as potential explanations for failures. While those approaches operate on code level, our approach operates on user-GUI interaction level.

Capture/replay approaches have been proposed to capture and replay user, application, and system events at different levels of abstraction (e.g. [2], [3], [8]). Herbold et al. [9] and Steven et al. [18] capture the application events triggered by user actions and enable developers to replay them. Those tools focus on capturing and replaying software execution but do not minimize captured traces.

Other approaches collect runtime data from deployed software: Tucek et al. [19] re-execute applications to collect additional failure information. Artzi et al. [1] generate and execute multiple tests to reproduce a given failure. Several tools for automated crash reporting exist which report failures and context information via the Internet such as MS Windows Error Reporting [7] or Apple Crash Reporter. Those tools collect information about system runtime and context but usually don't capture user interactions. Roehm et al [17] monitor high-level user interactions preceding failures and present them to developers but do not minimize captured traces.

Zimmermann et al. [23] propose to educate bug reporters to provide information about reproduction steps when composing a bug report which is complementary to our approach.

IV. Approach

This section describes our approach to automatically extract reproduction steps from interaction traces. Figure 1 shows an overview of the approach which consists of two main steps. First, interactions between a user and a WIMP GUI are capture with a capture/replay tool and temporarily stored on the user device. Examples of captured user interactions are menu selections, button clicks, context menu selections, drag-and-drop actions, textual inputs, and keyboard shortcuts. Because the approach uses a capture/replay tool to collect field data, interaction traces can be replayed automatically. Failure instances are detected by monitoring the failure log of the target application. Whenever a log entry originating from a failure instance is detected, information about the failure is extracted from the log. Then, the trace of interactions preceding the failure instance together with failure information is sent to an analytics server and stored in a database. Second, interaction traces in the database are analyzed on the analysis server. The purpose of the analysis is to extract the minimal, failure-inducing subsequence of a trace and present this subsequence as reproduction steps to developers. We use Delta Debugging (DD), Sequential Pattern Mining (SPM), and a combination of both algorithms (SPM + DD) to perform this extraction. We chose the DD algorithm because it has demonstrated capabilities to minimize failure-inducing input in several occasions. Additionally, we consider SPM additionally to DD because our case study showed that some interaction

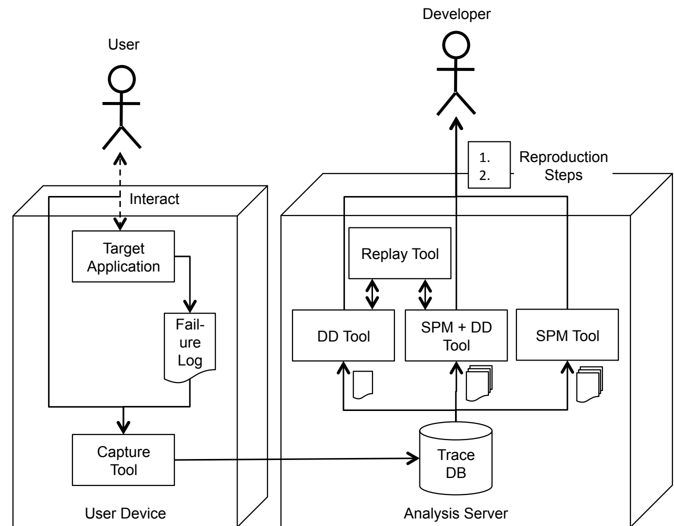

Figure 1. Overview of Approach
DD = Delta Debugging, SPM = Seq. Pattern Mining
Arrows denote information flow

traces cannot be replayed automatically which makes DD inapplicable.

A. Delta Debugging-Approach

The Minimizing Delta Debugging algorithm [22] (DD) minimizes a failure-inducing interaction trace to a minimal, failure-inducing interaction sequence. DD performs a binary search on the input interaction sequence. It splits the input interaction sequence in two subsequences and tests each of them individually. If any of the two subsequences trigger the failure, DD marks it as a minimal, failure-inducing interaction sequence and recursively continues to search in it for a shorter, failure-inducing subsequence. If none of the sub-sequences fails, DD increases the granularity and splits the input interaction sequence in multiple subsequences. DD increases the granularity until each subsequence contains only one interaction. DD checks whether a given interaction sequence induces a failure by replaying the interaction sequence with the capture/replay tool.
Output interaction sequences of DD are guaranteed to be 1-minimal, i.e. no single interaction can be removed from the output sequence without removing its failure induction. We refer to [22] for further details on the DD algorithm.

Figure 2 shows how DD is used in our approach. Intuitively, DD tests subsequences of an interaction trace until it finds a minimal, failure-inducing subsequence. The prerequisite for employing DD is a failure-inducing interaction trace which can be replayed by the capture/replay tool. During its operation, DD uses the capture/replay tool to replay subsequences of the input sequence and to check whether they cause a failure. The output of DD is guaranteed to be a replayable, 1-minimal, failure-inducing interaction sequence.

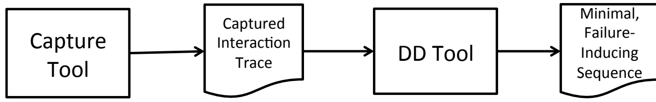

Figure 2. Overview of Delta Debugging (DD)-Approach

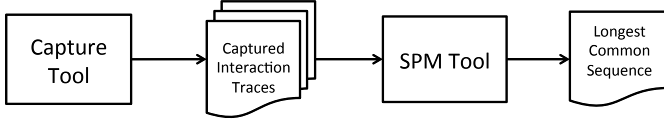

Figure 3. Overview of Sequential Pattern Mining (SPM)-Approach

B. Sequential Pattern Mining-Approach

Sequential Pattern Mining (SPM) [15] is a field of data mining concerned with finding frequent subsequence in a database of sequences. The fraction of sequences which a subsequence has to be contained in to qualify as resulting subsequence is called the support value. Several algorithms have been designed to solve the SPM problem. We use the BIDE algorithm [20]. It mines frequent, closed sequences, i.e. the longest subsequence with a given support.

Figure 3 shows how SPM is used by our approach. SPM requires a set of interaction sequences which trigger the same failure as input. We run the BIDE algorithm with a support of 100% on such a dataset. Intuitively, this identifies common subsequences within the dataset. Thereby SPM removes interaction which occur only in a single sequence or a subset of sequences (and hence are probably unnecessary to reproduce the failure) and approximates the minimal, failure-inducing interaction sequence.
There is no guarantee that the resulting interaction sequence is replayable, minimal, or failure-inducing. If the failure has unique reproduction steps, then they are contained in the SPM output sequence.

C. Combination of SPM and DD

DD and SPM have different, complementary strengths. DD has strong guarantees regarding minimality and failure-induction but might requires a replayable interaction sequence and might require considerable execution time. Instead, SPM represents a fast way to filter out unnecessary interactions. Hence, we combined both algorithms to improve performance (see Fig. 4). Prerequisite for the combined approach (SPM + DD) is a set of interaction traces triggering the same failure (same as for SPM). Those interaction traces are first processed by the SPM algorithm. If the resulting interaction sequence is replayable and failure-inducing (the prerequisites for DD), then DD can process it and the final output is a 1-minimal, failure-reproducing interaction sequence. If the intermediate sequence does not fulfill those properties, SPM + DD will not provide useful output.

D. Illustrative Example

In the following we discuss an example to illustrate the operation of the extraction approaches introduced above. Figure 5 shows interaction traces of four different users. Each user interaction is represented as a letter. Each interaction trace triggers the same failure, indicated by $✗_1$. We assume that the minimal, failure-inducing interaction sequence is ABEH. The result of DD is ABEH for the four user interaction traces, assuming that each interaction trace is replayable. This equals the minimal, failure-inducing interaction sequence. The result of SPM for the four interaction traces is ABCEFH which is not minimal but contains the minimal, failure-reproducing interaction sequence as a subsequence. When combining SPM and DD, first SPM is run on the four interaction traces which results in the interaction sequence ABCEFH. This intermediate result is further minimized by DD to ABEH, the minimal, failure-inducing interaction sequence.

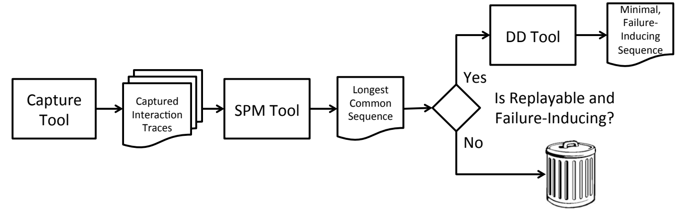

Figure 4. Overview of Combined Approach (SPM+DD)

Reproduction Steps: A B E H $✗_1$

Captured Interaction Traces:

User 1:	A	B	C	D	E	F	G	H $✗_1$
User 2:	A	B	C		E		G	H $✗_1$
User 3:	A	B	C		E	F	G	H $✗_1$
User 4:	A	B	C	D	E		G	H $✗_1$

Result of Sequential Pattern Mining (support=100%) : A B C E G H

Result of Delta Debugging: A B E H

Result of SPM + DD: A B E H

Figure 5. Illustrative Example

V. Evaluation

We prototypically implemented the approach described in the section above and evaluated it in a case study. This section discusses design and results of this case study. In this case study, we defined the following research questions:

- **RQ1:** Are the failure-inducing interaction traces captured by the QF Test tool replayable?
- **RQ2:** How well do the extraction approaches (DD, SPM, SPM + DD) perform in terms of filtering, i.e. which fraction of unnecessary interactions can they remove?
- **RQ3:** Can the extraction approaches (DD, SPM, SPM + DD) automatically identify reproduction steps, i.e. a minimal, failure-inducing interaction sequence?

RQ1 addresses the prerequisites for using DD: to have a replayable, failure-inducing interaction trace. We explicitly investigate this question because our industry partner reported that they rarely can replay a captured interaction trace without

manual manipulations. QF Test is the capture/replay tool we use in this case study and investigation of this question constitutes an investigation of how well an industrial capture/replay tool can process interactions of real users. The goal of this research is to automatically minimize captured, failure-inducing interaction traces, i.e. to identify and remove interactions which are not necessary to reproduce the failure. RQ2 and RQ3 address the performance of the proposed approaches. RQ2 addresses in general what fraction of unnecessary interactions were removed while RQ3 addresses whether the ideal situation was achieved by the different extraction approaches.

A. Case Study Design

The goal of this study is to collect user interaction traces and apply the extraction algorithms to evaluate their performance. To accomplish this goal, we had participants interact with an application that contained injected bugs and captured their interactions with an industrial capture/replay tool.

The capture/replay tool QF-Test [6] is used to capture user interaction traces in this case study. QF-Test was chosen after a review of existing capture/replay tools where it proved to be the most suitable tool. QF-Test captures GUI interaction events such button clicks, view activations, and text input which are performed by users using mouse and keyboard as input devices. Captured interaction traces are stored in a human-readable XML format. Every user interaction is stored as a separate XML entry. Because of this format, interaction traces captured with QF-Test can be manipulated, e.g. interactions can be added or removed. Manipulation interaction sequences can be replayed using QF-Test by passing a manipulated xml file.

The mail client "Simple Mail" is used as target application in this case study (see Figure 6). It is implemented using the Eclipse Rich Client Platform (RCP) technology and is contained in the Eclipse RCP distribution as example application. Simple Mail is a mail client which allows users to manage different mail accounts, compose new mails, and check for new mails. Its WIMP GUI consists of a main menu, an icon bar, account tree, and separate tabs for each mail.

We injected four bugs in the code of Simple Mail which were designed to mimic real failure scenarios. The "implementation" of each bug displayed a error message dialog (which named the bug/ failure to make it user-visible) and subsequently closed the application. The following bugs were injected:

Bug A Composing 8 mails at the same time
This bug mimics the scenario when an action is repeated too often and leads to a failure. It can be reproduced by creating seven new mails using the "Create mail" icon (one mail is automatically created at application startup).

Bug B Checking for new mails
This bug mimics the scenario when a failure occurs independent of user input. It can be reproduced by checking for new mails using the "Check mails" icon.

Bug C Entering invalid character(s) in mail text
This bug mimics the scenario when entering an invalid text

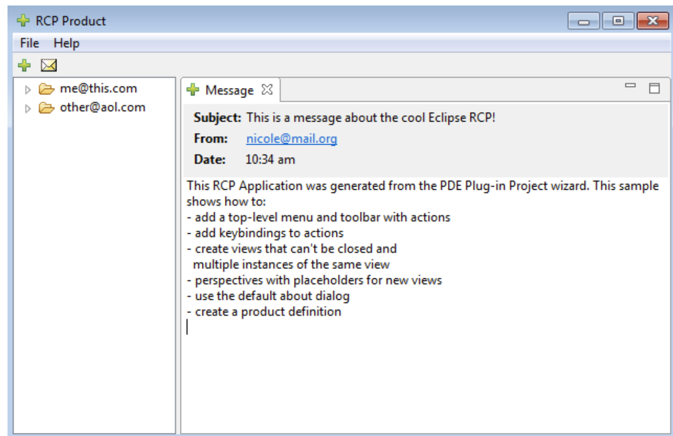

Figure 6. Screenshot of Simple Mail

character triggers a failure. It can be reproduced by entering a non-ASCII character in the mail body and pressing Enter.

Bug D Composing a mail which is too long
This bug mimics the scenario when excessive memory consumption leads to a failure. It can be reproduced by composing a mail body text which has more than 25 lines of text.

The case study was conducted with five participants aged between 25 and 30. All participants had a software engineering background, i.e. they had some experience with developing, bug-fixing, and using software. Participants had no previous knowledge about the injected bugs and Simple Mail.

Participants were presented the Simple Mail application. Their task was to "find" the injected bugs, i.e. to interact with Simple Mail till a bug is triggered and a failure message pops up. The participants interaction with the Simple Mail GUI were recorded using the QF-Test. When a bug was found, participants had to write down the bug id (which was shown in the failure message dialog) and reproduction steps according to their opinion. No time limit for finding the bugs was imposed on participants. Because of the purpose of this case study part was to collect failure-inducing interaction traces, we helped participants with hints when they didn't find a bug after five minutes. Table I shows for each participant the bug ids found, the time taken to find a bug, and the number of interactions till the bug was found. All participants found all bugs which is not surprising as we helped them with hints. On average, participants took 1 min 55 sec to find a bug and performed 88 interactions in that time. We used this dataset of failure-inducing interaction traces to run our extraction algorithms.

Table I
OVERVIEW OF DATA COLLECTED DURING CASE STUDY

Part. ID	Bug Type	Time (mm:ss)	Number of Interactions
1	B	00:20	22
1	D	01:30	88
1	A	00:10	16
1	C	04:24	109
2	B	00:10	16
2	C	02:04	31
2	A	01:10	78
2	D	06:25	223
3	B	00:10	12
3	A	01:00	24
3	C	04:50	168
3	D	03:20	86
4	B	00:36	28
4	D	01:20	54
4	A	00:25	60
4	C	04:20	256
5	B	00:35	28
5	A	03:00	310
5	D	00:10	18
5	C	02:30	111

B. *Case Study Results*

Table II shows the results of running the extraction algorithms on interaction traces collected from study participants. For each interaction trace, i.e. each combination of participant and bug, the table presents the length of the captured interaction trace, the length of its reproduction steps (i.e. the minimal, failure-inducing subsequence), and whether the interaction trace was replayable with QF-Test. For each of extraction algorithm, the table shows the length of the output sequence, the fraction of unnecessary interactions removed by the prototype, whether the output sequence is minimal and failure-inducing, and the execution time. Execution time was measured by running the prototypes on a MacBook Pro (i7 quad-core processor @2.5 GHz, 8 GB RAM). The execution times are given to enable relative comparison of execution times of different extraction algorithms.

Only 25% of interaction traces (5/20) were replayable directly. We had to manually inspect and manipulate the remaining interaction traces to make them replayable. Manipulations included removing window resizing actions, adding close dialog-actions when they were not captured successfully, adding selection events of GUI elements before subsequent actions could target them, removing events which were captured multiple times. Those manipulations enable replayability but do not change the user's interaction flow. We managed to make 14 interaction traces (70%) replayable by the manipulations (marked "after manipulation (AM)" in Table II). One interaction trace was not replayable even after modifications.

Bugs A and B can be triggered only by a single sequence of interaction, i.e. they have unique reproduction steps. In contrast, bugs C and D can be triggered in multiple ways due to the fact that text can be entered in different ways, e.g. by typing or copying.

DD was applicable to 95% of interaction traces. One interaction trace (participant 2, bug C) was not replayable and hence DD could not be applied. For 95% of interaction traces (18/19), DD identified the minimal, failure-inducing subsequence. For one interaction trace (participant 5, bug A), DD returned a 1-minimal, failure-inducing sequence which was not minimal. On average, DD filtered out 99.9% of interactions unnecessary for failure reproduction. The cost for such good performance was payed in execution time ranging from 4 minutes to 9 hours 36 minutes.

Because SPM needs a set of interaction traces triggering the same failure as input, we used captured interaction traces of participants per bug as input for SPM, i.e. one SPM run was performed for each bug type. SPM was applicable to 75% of interaction traces and 75% of bug types. It was not applicable to interaction traces of bug D because each participant performed different interactions triggering bug D. Hence, SPM cannot identify a common subsequence which is failure-inducing. The same situation holds for bug C where participants had to enter different special characters to trigger the failure. To make SPM applicable, we preprocessed interaction traces of bug C and replaced the entered special character with a common special character. The execution time of the SPM ranged from 12 seconds to 25 seconds. SPM did not find the minimal, failure-inducing sequence for any interaction sequence. Nevertheless, all SPM output sequences were failure-inducing and SPM managed to filter out 93% of interactions unnecessary for failure reproduction. Because the SPM output sequences were only 1.8 times longer than the minimal, failure-inducing sequence, we hypothesize that developers can manually analyze SPM output sequences and extract reproductions steps.

SPM + DD was applicable to 75% of all interaction traces and found the minimal, failure-inducing sequence for all of these interaction traces, i.e. it filtered out 100% of interactions unnecessary for failure reproduction. The execution time of SPM + DD ranged from 3 minutes 44 seconds to 18 minutes 47 seconds. This constitutes a performance improvement to DD alone in three ways: First, even for the interaction sequence which was not replayable but contained the failure-inducing sequence, the minimal, failure-inducing sequence was found. Second, SPM + DD found the minimal, failure-inducing sequence for the interaction trace for which DD produced a 1-minimal but not minimal sequence. Third, the execution time was reduced dramatically, in one instance from 8 h to 18 min. Similarly, the performance of SPM + DD improved compared to SPM alone as SPM + DD returned minimal sequences for all interactions. This improvement in filter quality has to be paid with a higher execution time.

VI. DISCUSSION

In this section we discuss the results of the evaluation case study as well as general aspects of our approach.

A. *Case Study Discussion*

This section discusses results and limitations of the case study.

Table II
Overview of Case Study Results
Filter Quality: Fraction of irrelevant interactions removed by algorithm (in %)
Failing?: Denotes whether resulting interaction sequence is failure-inducing
Minimal?: Denotes whether resulting interaction sequence is minimal

Part. ID	Bug ID	Trace Len.	Re-play-able? (RQ1)	Len. Repr. Steps	Delta Debugging (DD)				Sequential Pattern Mining (SPM)					SPM + DD				
					Out-put Len.	Mini-mal? (RQ3)	Filter Qual-ity (RQ2)	Exec. Time h:mm:ss	Out-put Len.	Mini-mal? (RQ3)	Fail-ing? (RQ3)	Filter Qual-ity (RQ2)	Exec. Time h:mm:ss	Out-put Len.	Mini-mal? (RQ3)	Fail-ing? (RQ3)	Filter Qual-ity (RQ2)	Exec. Time h:mm:ss
1	A	16	Yes		7	Yes	100%	0:30:43				89%					100%	
2	A	78	AM		7	Yes	100%	0:49:54				99%					100%	
3	A	24	AM	7	7	Yes	100%	0:29:39	8	No	Yes	94%	0:00:25	7	Yes	Yes	100%	0:18:47
4	A	60	AM		7	Yes	100%	0:29:19				98%					100%	
5	A	310	AM		13	No	98%	7:57:12				100%					100%	
1	B	22	Yes		1	Yes	100%	0:06:18				91%					100%	
2	B	16	AM		1	Yes	100%	0:05:29				87%					100%	
3	B	14	Yes	1	1	Yes	100%	0:04:37	3	No	Yes	82%	0:00:12	1	Yes	Yes	100%	0:03:30
4	B	28	AM		1	Yes	100%	0:05:22				93%					100%	
5	B	28	AM		1	Yes	100%	0:24:08				93%					100%	
1	C	109	AM		2	Yes	100%	0:13:32				96%					100%	
2	C	29	No		29	No	0%	0:10:56				85%					100%	
3	C	168	Yes	2	2	Yes	100%	0:14:52	6	No	Yes	98%	0:00:15	2	Yes	Yes	100%	0:07:44
4	C	256	AM		2	Yes	100%	0:26:35				98%					100%	
5	C	111	Yes		2	Yes	100%	0:09:48				96%					100%	
1	D	88	AM	10	10	Yes	100%	2:27:43										
2	D	223	AM	6	6	Yes	100%	1:03:13										
3	D	86	AM	12	12	Yes	100%	9:36:32	-	-	-	-	-	-	-	-	-	-
4	D	62	AM	6	6	Yes	100%	1:38:28										
5	D	18	AM	6	6	Yes	100%	1:35:19										

1) Discussion of Case Study Results: In the case study, five of 20 interaction traces captured with the capture/replay tool could be replayed directly, 14 traces needed manual manipulation to make them replayable, and one trace could not be replayed. This observation corresponds with experiences of our industry partner reporting that almost no captured interaction trace can be replayed directly and that they usually have to manipulate captured traces to make them replayable. This observation constitutes a problem for DD which requires a replayable interaction sequence to operate. Therefore, we suggest future research to study the state of the practice of capture/replay tools, i.e. how capture/replay tools are used in practice and identify problems and improvements when dealing with interactions of real users.

The DD algorithm identified the minimal, failure-inducing sub-sequence of an interaction trace in 18 of 20 cases. Because of its construction, it is suitable for failures with non-unique reproduction steps and its output sequences are always failure-inducing. On the downside, it required a lot of execution time, e.g. 4,5 minutes for an interaction trace of 12 interactions and up to 8 h for an interaction sequence of 210 interactions. When studying reasons behind this observation, we found that most of the time was spent to control the capture/replay tool, i.e. to start replays. Because the execution time depends on several factors, it is difficult to generalize but we conclude that the filter quality of the Delta Debugging algorithm has to be paid by investing execution time. Furthermore, the Delta Debugging algorithm requires a replayable interaction sequence which is not always available.

When DD was applied to the interaction trace of participant 5 and bug A, it filtered out most of the unnecessary interactions (from 310 interaction down to 14 interactions) but not all of them. When investigating this observation, we found the reason in the 1-minimality guarantee of DD: The DD algorithm guarantees that its result is 1-minimal [22], i.e. that it is not possible to remove one interaction from its result and maintain the failure-induction. But DD does not guarantee that a subsequence of its result sequence exists which is shorter by two or more interactions and still induces the failure. This was the case for interaction trace of participant 5 and bug A. In the case study, this situation occurred in one of 20 interaction traces. Future work has to investigate the impact of this limitation and, if necessary, identify improvements of the Delta Debugging algorithm.

The Sequential Pattern Mining algorithm was applicable to 15 of 20 interaction traces in the case study. For those interaction traces, it filtered out 93% of the unnecessary interactions and all its resulting interaction sequences were failure inducing. These results show that the Sequential Pattern Mining algorithm does not guarantee minimality as the Delta Debugging algorithm does. But the resulting interaction sequences were on average 1.8 times longer than the minimal, failure-inducing sequence and contained maximal eight interactions. Hence, we hypothesize that manual inspection of the resulting interaction sequence becomes possible. Furthermore, the resulting interaction sequence of SPM is failure-inducing for failures which have unique reproduction steps because those reproduction steps have to be contained in every interaction trace. The execution time of SPM algorithm in the order of seconds was rather short, especially when comparing it to the execution time of the DD algorithm processing the same data. While it is difficult to generalize the execution time, these results show that SPM can quickly filter out the majority of unnecessary interactions.

The SPM algorithm had difficulties when processing interaction traces of bugs C and D: it was applicable to interaction traces of bug C after manual preprocessing and it was not applicable to interaction traces of bug D. To trigger those bugs, users had to enter text which fulfilled a certain criterium, for bug C entering a special character and for bug D entering text that exceeded a certain length. To fulfill each criterium, users can execute several different interaction sequences, i.e. those bugs to not have unique reproduction steps. Consequently, the interaction traces contain different failure-inducing sequences and SPM is not able to recognize a common, failure-inducing subsequence. We conclude that the current version of the SPM algorithm is not applicable to failures with non-unique reproduction steps. In the following we discuss two ways to overcome this limitation: trace preprocessing and an adaptation of the SPM algorithm. Preprocessing was demonstrated for interaction traces of bug C in the case study: before the SPM algorithm was run, the interaction traces were preprocessed and all special characters were substituted by a shared special character. The SPM results for interaction traces of bug C indicate that preprocessing makes SPM applicable, but previous knowledge is necessary to determine which preprocessing operations to perform. Another approach to make SPM applicable to failures with non-unique reproduction steps is to adapt the SPM algorithm: instead of running SPM once with a support of 100%, SPM can be run multiple times with decreasing support values (e.g. 100%, 90%, 80%, ...) until one or more interaction sequences are found. We hypothesize that such an approach is able to identify different interaction sequences triggering the same failure. Such information could be used by developers to accelerate cause identification and detection of duplicate bug reports. We leave the implementation and evaluation of this adaptation as future work.

The case study also investigates the combination of SPM and DD to improve performance. Intuitively, the SPM algorithm with its short execution time filters out most unnecessary interactions before the DD algorithm identifies the minimal sequence in the remaining interaction sequence. This combination requires multiple interaction traces triggering the same failure and a failure with unique reproduction steps. In that case, SPM produces an interaction sequence which is failure-inducing and therefore can be further minimized by DD.

The combination of SPM and DD was applicable to 15 of 20 interaction traces and three of four bugs. All its results were minimal and failure-inducing. The SPM+DD combination performance nice: it was able to correctly identify reproductions steps of two cases which DD alone could not handle (interaction traces of (participant 5, bug A) and (participant

2, bug C)). Also, the execution time was drastically reduced, e.g. from 8 h to 18 seconds for bug A. We conclude that the SPM+DD combination improves performance compared to DD and SPM alone but its application is limited compared to DD.

2) Discussion of Case Study Limitations: As with every case study, our case study design has some limitations which we discuss in the following as starting points for future work. The case study was conducted with a single, basic application. As Simple Mail is an Eclipse RCP application, it represents state of the art implementation technology. But the generalizability of the approach to other, more complex applications has to be investigated by future work. We remark at this point that the transferability depends on the capture/replay tool used for capturing and replaying user interactions. We hypothesize that our approach can be applied to applications for which a capture/replay tool exists which can reliably capture and replay interaction of real users.

The case study uses four bugs which were designed to mimic common failure situations such as out of memory exceptions or text input validation exceptions. We do not claim the bugs to be representative and future work should study other types of bugs and the generalizability of our approach.

The case study uses the capture/replay tool QF-Test to capture and replay user interactions. Before choosing QF-Test, we compared different available capture/replay tools and identified QF-Test as the most suitable tool. To generalize the results beyond QF-Test, future work should study other capture/replay tools.

B. General Discussion

Because of their requirements, DD and SPM are applicable in different contexts. DD requires a replayable interaction trace and might need considerable execution time. If these requirements are satisfied, DD is applicable and its strong guarantees for minimality and failure-induction can be exploited. In contexts where the interaction trace is not replayable (as it was the case for 15 interaction traces in the case study) DD is not applicable. SPM requires multiple interaction traces triggering the same failure. We hypothesize that such traces can be collected when an application is used by multiple users and those experience the same failures. As those interaction traces do not have to be replayable for SPM to operate, SPM can be used in the context of multiple, non-replayable interaction traces where DD can't be used.

Compared to most related work, our approach captures and processes traces on the level of user interactions instead of code execution. This has the advantage that it relieves users from manually describing reproduction steps when reporting failures and allows developers to discuss extracted reproduction steps with users. We hypothesize that such discussions provide developers with insights about user behavior preceding failures, helps to identify wrong developer assumptions, and complements runtime information on code execution level. The disadvantage of this approach is that capturing user interactions might capture sensitive data and violate user privacy. Hence, future work should investigate privacy issues, i.e. measures for protecting user privacy while enabling failure reproduction.

We envision two use cases for our approach: In the first use case, our approach is deployed in the field to document field failures experienced by users. In the second use case, our approach is deployed in-house during testing to document failures detected by human testers. In both cases, each failure and its reproduction steps are documented automatically, enabling developers to reproduce it and fix the corresponding bug.

The analysis of interaction traces is currently performed on the analysis server (see Figure 1). This design was chosen to introduce as less performance overhead as possible on the user device. However, if the analysis could be performed on the user device and only extracted reproduction steps were sent to developers, the amount of data sent to developers would be greatly reduced. We hypothesize that this reduction has a positive effect on both the data transfer as well as privacy concerns. Hence, future work should investigate if the analysis can be accomplished on the user device.

Because our approach concentrates on the extraction of reproduction steps from user interaction traces, it has some limitations. It focuses on user interactions as source of non-determinism while there are other sources of non-determinism such as e.g. network traffic. Further, it addresses capturing and reproduction of the failure environment only indirectly by using a capture/replay tool. Hence, our approach should be combined with other approaches which complement it and tackle these limitations.

VII. Conclusion

We presented an approach to extract failure reproduction steps from user interaction traces. It captures user-GUI interactions with a capture/replay tool and uses three different algorithms to extract reproduction steps: Delta Debugging (DD), Sequential Pattern Mining (SPM), and a combination of both. We found that only 25% of interaction traces captured with a state of the practice capture/replay tool were replayable directly, threatening the applicability of DD which requires replayable traces as input. After manual modifications, DD was applicable to 95% of all traces and identified reproduction steps in 90% of all cases. Its output sequences are guaranteed to be minimal and failure-inducing. Besides its dependency on replayable traces, the execution time of DD was rather long. Furthermore, we studied SPM as complementary algorithm overcoming the limitations of DD. SPM was directly applicable to 50% of all traces and after manual modifications to 75%. In these cases, SPM identified and removed on average 93% of unnecessary interactions, potentially enabling developers to manually analyze output sequences to extract reproduction steps. SPM does not guarantee minimality or failure-induction of its output sequences and the SPM version used in this paper is applicable only for failures with unique reproduction steps. Moreover, we found that combining SPM and DD improves performance in terms of execution time and filter quality. But the combination is applicable to a smaller amount of traces.

Overall, these advantages and disadvantages of each extraction algorithm should be considered when choosing among them.

Future work should investigate several directions. First, it should investigate the state of the practice of capture/replay tools and ensure that interactions of real users can be captured and replayed reliably. Second, it should further evaluate the presented approach to address the limitations of the case study. Third, it should investigate how the capture tool can be deployed on user devices. Fourth, it should improve the SPM algorithm to tackle failures with non-unique reproduction steps. And fifth, it should investigate privacy issues arising when capturing user interactions.

ACKNOWLEDGEMENTS

We thank the participants of the case study for their time; Dr. Udo Hafermann, Alexandra Lungu, and Dr. Harald Schöning for their support; and Juan Haladjian as well as the anonymuous SANER reviewers for their comments. This research was funded by the German Federal Ministry of Education and Research (ReproFit project, Softwarecampus TU München, Förderkennzeichen 01IS12057).

REFERENCES

[1] S. Artzi, S. Kim, and M. D. Ernst. ReCrash: Making Software Failures Reproducible by Preserving Object States. In *European Conf. on Object-Oriented Programming*, volume 5142 of *LNCS*, pages 542–565. Springer, 2008.

[2] J. Bell, N. Sarda, and G. Kaiser. Chronicler: Lightweight Recording to Reproduce Field Failures. In *Proc. Int. Conf. on Software Eng.*, 2013.

[3] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst. Interactive record/replay for web application debugging. In *Proc. 26th ACM Symp. on User Interface Software and Technology*, pages 473–484, New York, New York, USA, 2013. ACM.

[4] M. Burger and A. Zeller. Minimizing reproduction of software failures. In *Proc. Int. Symp. on Software Testing and Analysis*, pages 221–231, New York, New York, USA, 2011. ACM.

[5] J. Clause and A. Orso. A Technique for Enabling and Supporting Debugging of Field Failures. In *Proc. 29th Int. Conf. on Software Eng.*, pages 261–270, 2007.

[6] Quality Software First. Description of the qf-test tool: http://www.qfs.de/en/qftest, Accessed: November 2014.

[7] 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. 22nd ACM Symp. on Operating Systems Principles*, pages 103–116. ACM, 2009.

[8] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. RERAN: Timing- and touch-sensitive record and replay for Android. In *Proc. 35th Int. Conf. on Software Eng.*, pages 72–81. IEEE, May 2013.

[9] 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.

[10] H. Hsu, J. A. Jones, and A. Orso. RAPID : Identifying Bug Signatures to Support Debugging Activities. In *Proc. of the 2008 23rd IEEE/ACM Int. Conf. on Automated Software Eng.*, pages 439–442. IEEE, 2008.

[11] W. Jin and A. Orso. BugRedux: Reproducing field failures for in-house debugging. In *Proc. Int. Conf. on Software Eng.*, pages 474–484. IEEE, 2012.

[12] W. Jin and A. Orso. F3: Fault Localization for Field Failures. In *Proc. Int. Symp. on Software Testing and Analysis*, page 213. ACM Press, 2013.

[13] E. I. Laukkanen and M. V. Mantyla. Survey Reproduction of Defect Reporting in Industrial Software Development. In *Int. Symp. on Empirical Software Eng. and Measurement*, pages 197–206. IEEE, 2011.

[14] W. Maalej, R. Tiarks, T. Roehm, and R. Koschke. On the Comprehension Of Program Comprehension. *ACM Trans. on Software Eng. and Methodology*, 2014.

[15] N. R. Mabroukeh and C. I. Ezeife. A taxonomy of sequential pattern mining algorithms. *ACM Computing Surveys*, 43(1):1–41, 2010.

[16] A. Orso. Monitoring, Analysis, and Testing of Deployed Software. In *Proc. FSE/SDP Workshop on Future of Software Eng. Research*, pages 263–267, 2010.

[17] T. Roehm, N. Gurbanova, B. Bruegge, C. Joubert, and W. Maalej. Monitoring User Interactions for Supporting Failure Reproduction. In *Proc. 21th IEEE Int. Conf. on Program Comprehension*, 2013.

[18] 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.

[19] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage : Diagnosing Production Run Failures at the User 's Site. In *Proc. 21st ACM SIGOPS Symp. on Operating Systems Principles*, 2007.

[20] J. Wang and J. Han. BIDE: efficient mining of frequent closed sequences. *Proc. 20th Int. Conf. on Data Eng.*, pages 79–90, 2004.

[21] A. Zeller. *Why Programs Fail*. Morgan Kaufmann, 2nd edition, 2009.

[22] A. Zeller and R. Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. on Software Eng.*, 28(2):183–200, 2002.

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

[24] D. Zuddas, W. Jin, F. Pastore, L. Mariani, and A. Orso. MIMIC: Locating and Understanding Bugs by Analyzing Mimicked Executions. In *Proc. 29th ACM/IEEE Int. Conf. on Automated Software Eng.*, pages 815–825. ACM, 2014.