2014 IEEE International Symposium on Software Reliability Engineering Workshops

Record and Replay GUI-based Applications with Less Overhead

João Matos, Nuno Coração and João Garcia
INESC-ID / Instituto Superior Técnico, Universidade de Lisboa, Portugal
jmatos, ncoracao, jog@gsd.inesc-id.pt

***Abstract*— Debugging is, typically, a hard and time-consuming task. Fault-replication mechanisms facilitate the debugging process by providing software developers with an error's "steps-to-reproduce". The main challenge of fault-replication is the overhead imposed by recording all non-deterministic events of an execution, such as thread interleaving and the user interaction with the application. The overhead imposed by user input is especially significant for graphical-based applications. This paper proposes a new approach to record and replay user interactions with the GUI, which significantly reduces the amount of recorded information. We developed an open-source implementation of an execution-recording framework and evaluated it using a test bed that includes real bugs from well-known applications. We achieved average reductions of 3567 times fewer events recorded.**

***Keywords*-Software Bugs; Reliability; Performance; Error Reporting; Fault-Replication; Record and Replay; GUI**

I. Introduction

A considerable amount of resources is spent in testing and fixing of software errors, which represent several billion dollars per year worth of maintenance costs [1]. This is because debugging is a hard and time-consuming task.

Reproducing failures that occur on client devices is one of the most important steps to locate bugs in a timely manner. Fault replication systems (e.g. [2], [3], [4]) allow developers to do so, by replaying errors in a step-by-step fashion. In order to deterministically reproduce clients' faulty executions, these systems monitor a user execution and log its non-deterministic events, such as the interaction between the user and the application. The main challenge for fault-replication mechanisms is the overhead imposed by the recording of sources of non-determinism during execution, which often requires a prohibitive resource usage.

Many strategies and techniques to address the overhead problem of fault-replication, have been proposed. These solutions are dedicated at reducing the amount of information monitored and logged, respective to one or more sources of non-determinism, namely thread interleaving (e.g. [2], [3], [4], [5], [6]). One additional source of non-determinism, which also causes recording overhead, is user input. Considering that most application nowadays have graphical user interfaces [7], the interaction between the user and the GUI is a very relevant source of non-determinism. Graphical events such as mouse actions are numerous in a typical execution and therefore overhead concerns exist also for this matter and the existing solutions for record/replay of graphical events (e.g. [7], [8], [9], [10]), do not attend to these concerns.

This work was supported by national funds through FCT – Fundação para a Ciência e a Tecnologia, under project PEst-OE/EEI/LA0021/2013.

This paper presents a new approach for recording graphical applications with less overhead called REGALO. REGALO is an open-source fault-replication framework with which developers can choose what they consider to be relevant events for recording. We present an experimental analysis based on 6 publicly available applications, which includes popular, large-scale software projects.

This paper is organized as follows. In Sec. III we describe REGALO. We also present in that section a set of properties that we defined, to better characterize graphical components and help developers to decide on the relevance of events and widgets. Additionally we provide five sets of policies that balance the recording cost and certainty of failure reproduction, which we evaluate in Sec. IV, before presenting some concluding bookmarks in Sec. V.

II. State of the art and motivation

A. Testing Tools

Record and replay techniques are most popular for testing purposes. During the testing stages of an application, this type of techniques is useful to discover bugs before an application is released. Many testing tools have been proposed, for both graphical [7], [8], [9], [10] and non-graphical [11], [12], [13] applications. However, software testing is typically unable to detect all program flaws. Software errors often manifest themselves after the software is released and persist long after that [14]. In fact, it is common for more than half of the resources, in a typical development cycle, being invested in testing and bug fixing, which represent several billion dollars per year worth of maintenance costs [1]. Currently, the most popular way to provide developers with information about a program failure is through traditional error reporting tools.

B. Traditional Error Reporting Tools

Initial error reporting mechanisms, such as Windows Error Reporting [15] and Mozilla Crash Report [16] involve mainly information collected at the end of a failed program execution. When an application crashes, the error reporting system gathers information with little or no criteria from the state of the process at the moment of the crash and submits it as an error report, if authorized by the user.

978-1-4799-7377-4/14 $31.00 © 2014 IEEE
DOI 10.1109/ISSREW.2014.31

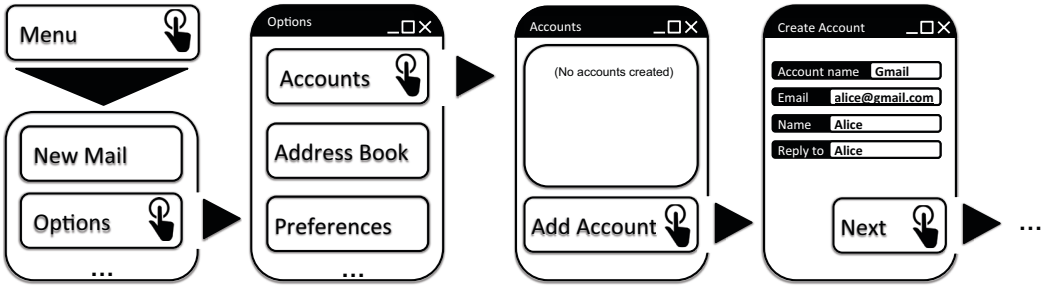

Fig. 1: Motivational example: an email client.

C. Fault Replication tools

Error reporting tools are not necessarily meant to provide the developers with the steps to reproduce the failure. On the other hand, fault-replication tools give historical information on how the error was reached, guiding the programmer (step by step) to the failure that occurred in the client device. Plenty of fault-replication techniques were published ([2], [3], [4] to name a few). In order to replicate the original execution, these systems need to log all its' non-deterministic events, such as user input, network packets and thread interleaving (amongst others). The two main challenges of fault replication are:

- Recording all sources of non-determinism (in order to achieve deterministic replay), often imposes prohibitive time and memory overheads on the user execution.
- Addressing the privacy concerns raised by recording sensitive sources of non-determinism such as user input.

The most expensive source of non-determinism to record is thread interleaving. Nevertheless, other sources of non-determinism may impose a high cost and although their cost is not as high as the cost of thread interleaving, it is often significant. *This work is focused on the overhead caused by the interaction between the user and the graphical user interface of the application*. To record and replay thread interleaving, dedicated solutions should be employed (e.g. [2], [3], [4], [5], [6], [17]). Furthermore, to enhance privacy within fault-replication, the following solutions should be considered [18], [19], [20], [21], [22], [23].

Recording overhead by user input. Even during a short program execution, solutions that capture all graphical events (e.g. [8]) will record hundreds of thousands of events corresponding to mouse movements alone [11]. Furthermore, it has been estimated in [24] that, in a typical execution, there are thousands of clicks and keystrokes per hour. Therefore, reducing the amount of recorded graphical events is important as it may impose a considerable cost. This work proposes an approach to reduce the overhead imposed by the user interaction with GUI.

Figure 1 exemplifies an interaction between the user and an email client. The simplicity of the example depicted in Fig. 1 allows us to explain, more clearly, the intuition behind our proposed framework. Our objective is to reduce the amount of information recorded and still be able to replay the failure. We can observe in Fig. 1 that, i) some events/actions such as mouse movement, resize/maximize the window or clicking the window border would have no relevance, ii) not all button clicks trigger the execution of application-specific code and iii) if the user acted on the *Create Account* window, then all (prior) actions from the click on the Menu button until the click on the "Add Account" button, could be deduced deterministically (e.g. to open the *Create Account* one has to click the *Add Account* button). Our proposed framework REGALO helps developers adjust the execution's recording overhead, by allowing them to specify which type of events they consider to be relevant and necessary for reproducing the failure.

III. REGALO

This section presents an overview of our framework. Our objective is to provide the means for developers to achieve a good balance between recording overhead and certainty of failure reproduction, in graphical-based applications, as depicted in Fig. 2. In order to guarantee failure reproduction while imposing the least possible overhead, one has to record all relevant events and dismiss all irrelevant events, achieving what we call as the *lower overhead bound*. The challenge is to determine what is relevant and what is not, which is far from being trivial. It is a fair assumption that, typically, it is not possible to predict what is strictly necessary to record, in order to later reproduce a previously unknown error and therefore the *lower overhead bound* can only be achieved in practice for very simple scenarios. Our REGALO framework allows the developer to specify what she considers to be relevant for recording, in order to achieve a good balance between cost and certainty of reproduction.

A. Overview

REGALO allows developers to choose what kind of graphical events are relevant for recording. Each input in a graphical interface is a (e, w) pair, corresponding to an event that occurred on a widget. The configuration phase of REGALO collects policy specifications of the type (Event<type>,Widget<type>), which the recorder enforces. The recorder filters every (e, w) pair that fits the policies devised by the developer. The developer is responsible for the specification of the policies

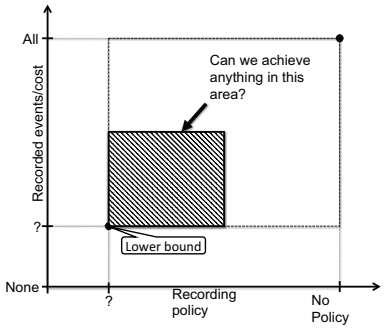

Fig. 2: Cost VS Policy example.

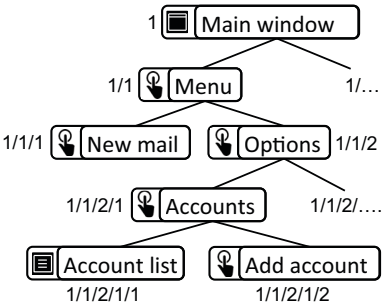

Fig. 3: Widget Identification.

and therefore is also responsible for the balance exemplified in Fig. 2. Considering that it is hard for the developer to anticipate which pairs (e, w) are relevant, we evaluate in this paper some policies that are capable of achieving a good balance.

1) Widget Identification: One important step in this work is to uniquely identify each graphical component. A graphical user interface is hierarchical [7]. Typically the main window is the root of the hierarchy tree and parents the graphical components visible in the window. We exemplify a hierarchy in Fig. 3, as well as our method employed to identify graphical components. Note that the hierarchy exemplified in Fig. 3 is associated with the example of Fig. 1. In our implementation of REGALO we decided to use the hierarchy to uniquely identify each graphical component: the identification of a component is a prefix of the identification of its children. If two components c_1 and c_2 open the same window w that contains component c_3, c_3 is identified differently for each situation as it inherits (as its prefix) the id of the component that opened w (either the id of c_1 or of c_2). In other words, our scheme treats c_3 when w is opened by c_1 as being a different component from c_3 when w is opened by c_2.

B. Design and Prototype

The operational workflow of REGALO is illustrated in Fig. 4. We implemented this framework in the Java programming language[1].

1) Pre-deployment: Ripping Handler: Before the application is released, REGALO resorts to GUI ripping [25] to obtain the hierarchy map of the application and to uniquely identify each graphical component. The ripping component runs the application and automatically extracts every graphical component of the GUI. The ripping handler outputs the hierarchy map of the application.

2) Pre-deployment: Policy Handler: In this phase, REGALO enforces the recording policies. The policy handler works in two levels, the event level and the component level. As mentioned, the policies are specified in the form (Event<type>, Widget<type>). If the developer intends to record every GUI mouse click, she will employ a (event<MOUSE_CLICKED>,-) policy and the policy handler will update the recording event mask of the *EventListener* interface (of the Java API) to filter *MouseEvent.MOUSE_CLICKED* events. On the other hand, if the developer wants to record at the component level, e.g. she wants to record clicks on all JButtons labeled as *cancel* (-,widget<class=JButton, label="cancel">), the policy handler uses the SOOT bytecode instrumentation tool [26] to instrument the *actionPerformed* method of every JButton labeled as *cancel*.

If the user intends to record at the component level, the policy handler traverses the hierarchy map and creates a set containing the ids (Fig. 3) of every component that matches the policies specified, which we simply call *target set*. This target set is provided to SOOT, indicating which listener methods should be instrumented. We acknowledge that we could access components at the event level too, but to verify, at the event level, if the component is within the target set, would impose additional overhead.

3) Deployment: Recorder: The recording of events is implemented using the *EventListener* interface of the Java API. The recording component is provided with the *recording mask* generated by the *policy handler*, filtering the events specified as relevant by the policies. Additionally, the methods that were instrumented will perform logging operations, to record the respective components.

4) Maintenance: Unfold Handler: The *unfold handler* attempts to generate a complete sequence of events if, during the recording stage, some relevant events were not recorded. It consults the hierarchy map generated by the ripping handler, to extract all unrecorded ancestors of each recorded component. This is achieved by locating the recorded component in the hierarchy tree and moving up, one ancestor at a time. Therefore, the unfold handler allows to further reduce recording, because we can choose not to record (some) potentially important actions, knowing the *unfold handler* will recover them later. For example, if the *recorder* logs the action at the *Add account* button of Figs. 1 and 3, the unfold handler is able to (later) deduce the preceding actions at the *Account*, *Options*, *Menu* and *Main Window* components, simply by moving up in the hierarchy structure obtained during the *ripping* phase.

5) Maintenance: Replay Handler: Our replayer takes the event sequence provided by the *unfold handler* and replays the sequence in a similar way as previous record/replay solutions for GUI applications (e.g. [8]). The visualization of the replay

[1]The prototype is open source at http://sourceforge.net/projects/fastfixrsm/

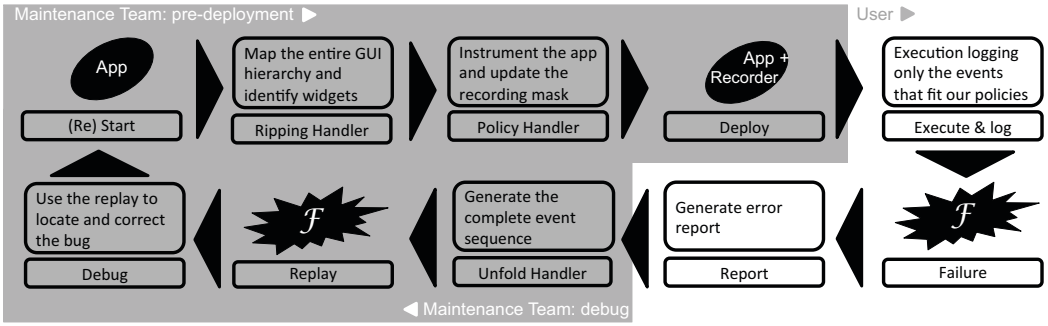

Fig. 4: Proposed work flow.

helps developers to locate the cause of failure. After the bug is corrected, the cycle depicted in Fig. 4 starts again.

C. Widget Properties

In order to configure REGALO, one needs to know which type of widgets to record and which to dismiss. We can discriminate between widgets by their native properties, such as type (button, combobox, window), border, color, label, amongst many others. In this work we devised three additional properties to characterize widgets, which facilitate the elaboration of polices. We describe these properties as follows.

1) Parenting: Following the same example of Figures 1 and 3, a click on the item *Options* automatically implies a click on the button *Menu*. Intuitively, the best strategy would be to record only the actions on graphical components with the highest depth and later use the *unfold handler* to determine what was not recorded. However we cannot predict up to what depth the user will keep on clicking. When a GUI window is opened it is likely to be eventually closed, typically by a click on an *ok* button, *cancel* button or the *close window* button. A widget that closes a window does not parent any component. We call a widget that is not a parent, a *childless* widget. Targeting *childless* widgets seems to be a good strategy for recording, because they are more likely to capture the last actions on a subsequence of events. According to this category a widget is either a *parent* widget (widget<p>) or a *childless* widget (widget<c>).

2) Activity: Typically only some of the widgets of a GUI invoke the application to execute. In this work we say that these widgets are *active* widgets, widget<$*$,a>. All other widgets are *static* widgets, widget<$*$,s>. The $*$ means that it does not matter whether the widget is childless or parent. Activity is an important property as well, because, in order to guarantee reproducibility, we need to execute the sequence of instructions of the original execution.

3) Value: Some widgets, such as text fields, text areas or a radio buttons (amongst many others) are what we call as *value* widgets, simply because they hold a value inserted by the user. This type of widgets is frequently of the utmost relevance for reproducing a failure, as it may be triggered by a value (a wrong value?) inserted by the user. We consider to be a good policy to log all *value* widgets, widget<$*$,$*$,v>, because we cannot guarantee the reproduction of the failure otherwise. We refer to all other widgets as *hollow* widgets, widget<$*$,$*$,h>.

D. Proposed policies

We implemented five policies with REGALO, which are evaluated in Sec. IV. Each policy is specified as a set of (**e**<type>,**w**<type>) pairs, which stands for event type and widget type. If we define (**e**<$\ldots$>,**-**) REGALO will record solely at the event level and if we define (**-**,<$\ldots$>) REGALO will record solely at the instrumentation level.

1) Complete recording (Basic): This scheme consists on running REGALO with a specified policy of (**e**<$*$>,**-**). It is equivalent to recording all interaction between the user and the GUI and is used as a baseline comparison between techniques that capture all graphical events (e.g. [8], [11]) and REGALO with the policies described next.

2) Click recording (CLK): The intuition behind the elaboration of this policy is to record exclusively the type of events that most incite the application to execute and clicks are arguably the events that most trigger execution. This policy consists simply on recording clicks, and can be specified in REGALO as (**e**<$MouseEvent.MOUSE\_PRESSED$>,**-**).

3) Click recording and value widgets (CLKV): The values inserted by the user in widgets such as text areas, text fields, radio buttons, combo boxes, amongst many others, may be failure inducing. Therefore this scheme consists on: (**e**<$MouseEvent.MOUSE\_PRESSED$>,**-**) $\cup$ (**-**,**w**<$*$,$*$,v>).

4) Active widgets and Value widgets (AV): This set is based on the insight that the amount of clicks in a typical execution can be high [24] and a significant subset of this amount does not trigger any action of the application (e.g. a click on the window border). This scheme consists on (**-**,**w**<$*$,a,$*$>) $\cup$ (**-**,**w**<$*$,$*$,v>).

5) Childless widgets and Value widgets (CV): This scheme exploits the parenting property to reduce the amount of information recorded. It aims to record clicks on *childless* widgets and *value* widgets: (**-**,**w**<c,$*$,$*$>) $\cup$ (**-**,**w**<$*$,$*$,v>). Note that both this scheme and the previous one (AV) do not record anything at the event level. These two schemes make the most use of the *unfold handler*.

IV. Evaluation

In this section we evaluate REGALO using the sets of polices described in Sec. III-D, in terms of reproducibility and amount of information recorded. The experiments were conducted using a machine running the MacOS X Lion operating system, with a 2.5 GHz Intel Core i5 processor and 4GB of memory.

A. Subjects

This evaluation was performed using six publicly available applications chosen according to their popularity and availability of real bugs.

• *jEdit* [27] is a large and popular Java text editor (2366 classes, 115 kLOC). The bug considered in this evaluation was reported in [28] and is triggered when the user clicks on a button within the options menu.

• *TV-Browser* [29] is a large and popular TV guide (2491 classes, 110 kLOC), that allows the user to check what is scheduled in over 1000 channels and radio stations. The bug studied for this subject was reported in [30]. This bug manifests itself when the user attempts to delete a filter while providing no filter name.

• *Columba* [31] is an email client (3356 classes, 108 kLOC) that contains a fault in its address book component. This test case was used in the evaluation of the work in [20]. The contacts are stored in a CSV file and if the content does not meet the expected format, Columba crashes while loading it.

• *Pooka* [32] is also a Java mail client (854 classes, 48 kLOC). It contains a bug that crashes the application when the user attempts to create a new message without having created an account. This was reported in [33].

• *Lexi* [34] is a simple Java word processor (156 classes, 6901 LOC). The bug used in this subject was reported in [35] and consists simply of trying to access the *popup options* menu, which immediately crashes with a *NullPointerException*.

• *MyJPass* is a benchmark password manager (3 classes, 1225 LOC) that manages simple user accounts. The bug is artificial and consists of a *NullPointerException* thrown when the form values do not meet the expected format.

B. Reproducibility

All the applications were executed for approximately 10 minutes, and their GUIs were fully explored. The same execution is performed for all schemes. Once the error is triggered, REGALO generates an error report with the recorded log. We present the reproducibility results in two tables, i) table I if the recorded event sequence is sent directly to the replayer and ii) table II if the recorded event sequence is processed by the unfold handler before being replayed.

The CLK scheme did not succeed in the experiment with MyJPass, which suggests that recording clicks alone is not enough to assure error reproduction. Furthermore, if CLKV succeeded in the same experiment, it suggests that the (**a**<$TextEvent$>,**w**<$*$,$*$,v>) policy is important to ensure error reproduction. The AV and CV policies may not reproduce the error (except AV in jEdit experiment), unless the unfold handler is employed, which is an expected result considering that these polices were devised to dismiss information that can later be recreated by the unfold handler. The results in Table II indicate that the AV and CV schemes were successful in all experiments if the unfold handler were employed. The unfolding operations took merely a few seconds to finish, which is perfectly admissible considering that they are meant to be executed by the maintenance teams.

	scheme				
Test case	**Basic**	**CLK**	**CLKV**	**AV**	**CV**
jEdit	Yes	Yes	Yes	Yes	No
TVBrowser	Yes	Yes	Yes	No	No
Columba	Yes	Yes	Yes	No	No
Pooka	Yes	Yes	Yes	No	No
Lexi	Yes	Yes	Yes	No	No
MyJPass	Yes	No	Yes	No	No

TABLE I: Was the error reproduced? **No unfolding**

	scheme				
Test case	**Basic**	**CLK**	**CLKV**	**AV**	**CV**
jEdit	Yes	Yes	Yes	Yes	Yes
TVBrowser	Yes	Yes	Yes	Yes	Yes
Columba	Yes	Yes	Yes	Yes	Yes
Pooka	Yes	Yes	Yes	Yes	Yes
Lexi	Yes	Yes	Yes	Yes	Yes
MyJPass	Yes	No	Yes	Yes	Yes

TABLE II: Was the error reproduced? **With unfolding**

C. Recording Overhead

We measured the amount of events recorded by each of the proposed schemes. Figure 5 presents the results obtained in this part of the evaluation. It presents the recorded event count by each of the schemes in the y-axis, and, on top of each box, the improvement comparing to the basic scheme. All presented schemes achieved a massive reduction comparing to solutions that capture everything, with an average improvement of 1875 times less events recorded and up to 7552 times less. The most filtered event in our tests was mouse movement. In this evaluation, REGALO recorded hundreds of events (212 on average, and the best result was 19 events) from a total of hundreds of thousands (298k events on average). These results suggest that REGALO, with a proper set of policies that sidesteps some event types, that we consider to be unnecessary for error reproduction (e.g. mouse movement), is able to achieve error reproduction of GUI-based applications with significantly less overhead. Figure. 5 also suggests that REGALO has the potential to achieve better improvements, compared to solutions that log all events, for applications with more complex GUIs: the improvement was significantly lower, (for all schemes) for the Lexi and MyJPass subjects, which are by far the smallest applications, with the simplest GUIs in our test bed.

Comparing the four proposed schemes, AV and especially CV were significantly better than CLK and CLKV. The CV scheme was the best policy in this evaluation, with an average reduction of 3567 times less events, 4.84 times better than CLKV and 1.49 times better than AV. This suggests that the

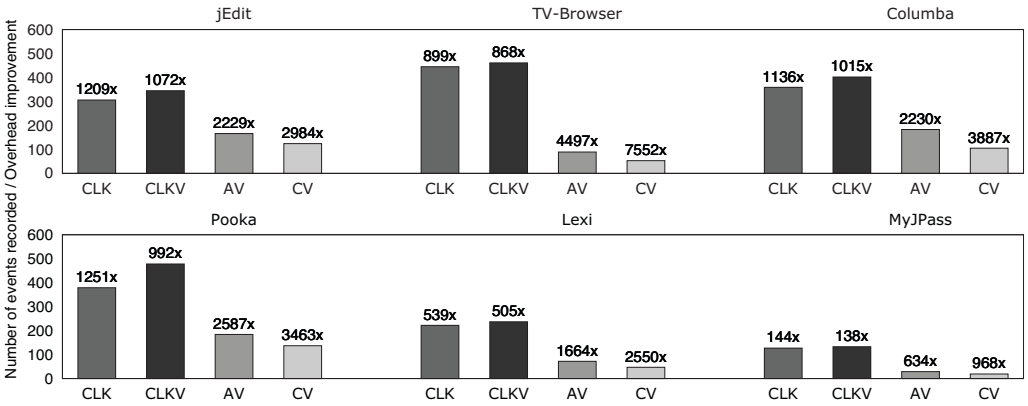

Fig. 5: Number of events recorded / Overhead improvement (×times less) comparing to basic sheme.

parenting property is useful to reduce recording overhead. It also suggests that significantly better results can be achieved by well-devised policy schemes.

V. Conclusions and Future Work

This paper presented REGALO, a framework that tackles recording overhead imposed by graphical events in fault replication record-and-replay tools. REGALO advances the state of the art of record and replay graphical-based applications, by reducing the amount of information recorded. The framework presented allows the developer to specify what she considers to be important events for recording. This paper also presented four recording schemes, of which three were able to reproduce the error in all of our tests. The proposed schemes achieved very large reductions in recorded information: an average of 1875 times less events recorded in all experiments. The most successful scheme (CV) achieved an average of 3567 times less events recorded. As future work we intend to develop an interactive approach that gradually reduces the number of events that need to be saved, as information about the execution becomes available.

References

[1] Cambridge University: Cambridge University Study States Software Bugs Cost Economy $312 Billion Per Year (2013)

[2] Altekar, G., Stoica, I.: Odr: output-deterministic replay for multicore debugging. In: SOSP. (2009)

[3] Huang, J., Liu, P., Zhang, C.: LEAP: lightweight deterministic multi-processor replay of concurrent java programs. In: FSE. (2010)

[4] Park, S., Zhou, Y., Xiong, W., Yin, Z., Kaushik, R., Lee, K.H., Lu, S.: Pres: probabilistic replay with execution sketching on multiprocessors. In: SOSP. (2009)

[5] VMware: The Amazing VM Record/Replay Feature in VMware Workstation 6: http://blogs.vmware.com/vmtn/2007/04/recordreplay_in.html (2011)

[6] N. Machado, P.R., Rodrigues, L.: Lightweight cooperative logging for fault replication in concurrent programs. In: DSN. (2012)

[7] Memon, A.: An event-flow model of gui-based applications for testing: Research articles. Wiley. Software Testing Verification and Reliability (2007)

[8] Steven, J., Chandra, P., Fleck, B., Podgurski, A.: jRapture: A Capture/Replay Tool for Observation-based Testing. In: ISSTA. (2000)

[9] Ganov, S.R., Killmar, C., Khurshid, S., Perry, D.E.: Test generation for graphical user interfaces based on symbolic execution. In: AST. (2008)

[10] Miura, M., Tanaka, J.: A framework for event-driven demonstration based on the java toolkit. In: APCHI. (1998)

[11] Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Transactions on Software Engineering (2002)

[12] Whalley, D.B.: Automatic isolation of compiler errors. ACM Transactions on Programming Languages and Systems (1994)

[13] Agrawal, H., Horgan, J.R.: Dynamic program slicing. SIGPLAN (1990)

[14] Zamfir, C., Candea, G.: Execution synthesis: A technique for automated software debugging. In: EuroSys. (2010)

[15] Microsoft Corporation: Windows error reporting: http://msdn.microsoft.com/en-us/library/bb513641 (2007)

[16] Mozilla Foundation: GNOME bug tracking. http://bugzilla.gnome.org/ (1998)

[17] Fonseca, P., Li, C., Rodrigues, R.: Finding complex concurrency bugs in large multi-threaded applications. In: EuroSys. (2011)

[18] Wang, R., Wang, X., Li, Z.: Panalyst: privacy-aware remote error analysis on commodity software. In: Usenix SS. (2008)

[19] Castro, M., Costa, M., Martin, J.P.: Better bug reporting with better privacy. ASPLOS (2008)

[20] Clause, J., Orso, A.: Camouflage: Automated Sanitization of Field Data. In: ICSE. (2011)

[21] Louro, P., Garcia, J., Romano, P.: MultiPathPrivacy: Enhanced privacy in fault replication. In: EDCC. (2012)

[22] Andrica, S., Candea, G.: Mitigating Anonymity Concerns in Self-testing and Self-debugging Programs. In: ICAC. (2013)

[23] Matos, J., Garcia, J., Romano, P.: Reap: Reporting errors using alternative paths. In: Programming Languages and Systems. Springer (2014)

[24] Wellnomics: An analysis of computer use across 95 organisations in europe, north america and australasia: http://www.wellnomics.com/ (2007)

[25] Memon, A., Banerjee, I., Nagarajan, A.: GUI ripping: reverse engineering of graphical user interfaces for testing. In: WCRE. (2003)

[26] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot - a java bytecode optimization framework. In: CASCON. (1999)

[27] jEdit: jEdit. http://www.jedit.org (1999)

[28] jEdit: jEdit bug report 3776. http://sourceforge.net/p/jedit/bugs/3776/jedit (2013)

[29] TV-Browser: TV-Browser. http://www.tvbrowser.org/ (2002)

[30] TV-Browser: TV-Browser bug report TVB-48 . http://tvbrowser.org:8080/jira/browse/TVB-48 (2007)

[31] Columba Team: Columba (2005) http://sourceforge.net/projects/columba/.

[32] Suberic: Pooka Mail Client. http://www.suberic.net/pooka/ (2005)

[33] Pooka: Pooka bug report 33. http://sourceforge.net/p/pooka/bugs/33/ (2003)

[34] Lexi: Lexi Java Word Processor. http://sourceforge.net/projects/lexi/ (2005)

[35] Lexi: Lexi bug report 13. http://sourceforge.net/p/lexi/bugs/13/ (2005)