Evaluating Rogue User Testing in Industry: an Experience Report

Sebastian Bauersfeld
Centro del PROS
Universitat Politècnica de València
Valencia, Spain
Email: sbauersfeld@pros.upv.es

Antonio de Rojas
Clave Informática S.L.
Alicante, Spain
Email: aderojas@clavei.es

Tanja E.J. Vos
Centro del PROS
Universitat Politècnica de València
Valencia, Spain
Email: tvos@pros.upv.es

***Abstract*—Testing applications with a graphical user interface (GUI) is an important, though challenging and time consuming task. The state of the art in the industry are still capture and replay tools, which may simplify the recording and execution of input sequences, but do not support the tester in finding fault-sensitive test cases and leads to a huge overhead on maintenance of the test cases when the GUI changes. In earlier works we presented the Rogue User Testing Tool, an automated approach to testing applications at the GUI level whose objective is to solve part of the maintenance problem by automatically generating test cases based on a structure that is automatically derived from the GUI. In this paper we report on our experiences obtained when implanting the Rogue User testing Tool with the Spanish software vendor *Clavei* who decided to apply the tool to stress test a component of one of their ERP applications. Our main goal was to identify potential problems that arise during the setup of the Rogue User. While carrying out our tests, we discovered critical and previously unknown faults in the application under test.**

I. Introduction

Testing software applications at the Graphical User Interface (GUI) level is a very important testing phase to ensure realistic tests because the GUI represents a central juncture in the application under test from where all the functionality is accessed. Contrary to unit or interface tests, where components are operated in isolation, GUI testing means operating the application as a whole, i.e. the system's components are tested in conjunction. This way, it is not only possible to discover flaws within single modules but also faults arising from erroneous or inefficient inter-component communication. However, it is difficult to test applications thoroughly through their GUI, especially because GUIs are designed to be operated by humans, not machines. Moreover, they are inherently non-static interfaces, subject to constant change caused by functionality updates, usability enhancements, changing requirements or altered contexts. This makes it very hard to develop and maintain test cases without resorting to time-consuming and expensive manual testing.

The current industrial state of the art in GUI level testing is Capture and Replay (CR) [1], [2]. The idea behind CR is that of a tester developing use cases and recording (capturing) the corresponding input sequences, i.e. sequences of actions like clicks, keystrokes, drag and drop operations. These sequences are then replayed on the UI to serve as regression tests for new product releases. A major problem with this approach is that the created sequences often break when the UI changes (e.g., controls are removed or repositioned). This has severe ramifications for the practice of testing: instead of creating new test cases to find new faults, testers struggle with repairing old ones, in order to maintain the test suite! For software applications, the UIs change all the time and hence make the CR method infeasible. Furthermore, new generation of applications are increasingly able to adapt their own layout to a target screen (e.g. small or large) and its user profile (e.g. normal or elderly). Consequently, CR tools are sometimes referred to as "Shelfware" and CR tool vendors are accused of trying to sell them as the silver bullet [3]. Due to this maintenance problem, companies return to manual regression testing which results in less testing being done and faults that still appear to the users.

In previous work we have presented an approach to testing at the GUI level [4], [5] whose objective is to solve part of the maintenance problem by automatically generating test cases based on a structure that is automatically derived from the GUI. The tool is called Rogue User and it uses the operating system's Accessibility API to recognize GUI controls and their properties and enables programmatic interaction with them. It derives sets of possible actions for each state that the GUI is in and automatically selects and executes appropriate ones in order to drive the GUI and eventually crash it. The tool uses a machine learning algorithm called Q-Learning to generate short, fault- effective and reproducible crash sequences, which a normal human user would not come up with.

This paper reports on the experience gained when implanting the Rogue User testing tool at a company called Clavei, a Spanish software vendor that develops the accounting software *ClaveiCon* that is part of their Enterprise Resource Planning (ERP) system (see Figure 1) and sold to companies within Spain. The product is in a mature state and has been tested and applied for many years. However, Clavei recognizes to have the regression testing maintenance problems when the GUI changes and hence their clients every-time find more faults which evidently is a very undesirable situation. The goal of the study was twofold. On the one hand Clavei was interested to see how the tool could be used within their context and on their software application. On the other hand, we were interested to see how easy of difficult is was to implant our academic prototype within an industrial setting.

II. The Context

Clavei is a private software vendor from Alicante, which has specialized for over 26 years in the development Enterprise Resource Planning (ERP) systems. for Small and Medium Sized (SME) companies. One of their main products is called *ClaveiCon* a software solution for SMEs for accounting and financing control.

Due to their many clients, it is of fundamental importance to Clavei to thoroughly test their application before releasing a new version. Currently, this is done manually. Due to the complexity and size of the application this is a time-consuming and daunting task, which is not always done well as is observed by the amount of faults that are communicated by the clients of the company.

Clavei is eager to investigate alternative, more automated approaches to reduce the testing burden for their employees and has been seeking information about Capture & Replay tools. After having attended a presentation at the Technical University of Valencia, the company expressed explicit interest in the Rogue User Tool and requested to carry out a trial period to investigate the applicability of the tool for testing their ERP products. The findings of which are presented in this paper.

Clavei's testing and development team creates test cases by relying on specified use cases. Each test case describes a sequence of the user interactions with the graphical interface and all of them are executed manually by the test engineers. If a failure occurs, the engineer reports it to a bug tracking system and assigns it to the developer in charge of the part affected by the failure. If necessary, this developer then discusses the issues with his team, fixes the fault and re-executes the test cases to ensure that the application now performs as expected.

The SUT in our investigation is *ClaveiCon* (Figure 1), an accounting software that belongs to a classic database-backed Enterprise Resource Planning system developed at Clavei.

The application is used to store data about product planning, cost, development and manufacturing. It provides a real-time view on a company's processes and enables controlling inventory management, shipping and payment as well as marketing and sales.

ClaveiCon is written in Visual Basic, makes use of the *Microsoft SQL Server 2008* database and targets the Windows operating systems. The application can be considered to be in a mature state, as it has been applied in companies all over Spain during more than a decade.

The fact that Clavei tests their application manually before each release, entails considerable human effort and consequently high costs, which makes it desirable to investigate automated alternatives.

III. The Rogue User Testing Tool

Modern GUIs are large, complex and difficult to access programmatically which poses great challenges for their testability. The Rogue User (RU) is a technique that allows completely unattended testing of large and complex gui-based SUTs. Its basic sequence generation algorithm comprises the following steps:

1) Obtain the GUI's state (i.e. the visible widgets and their properties like position, size, focus ...).
2) Apply an oracle to check whether the state is valid. If it is invalid, stop sequence generation and save the suspicious sequence to a dedicated directory, for later replay.
3) Derive a set of sensible actions (clicks, text input, mouse gestures, ...).
4) Select and execute an action.
5) If the given amount of actions and sequences has been generated, stop sequence generation, else go to step 1.

In its default mode, the RU selects the actions to be executed at random. We believe that this is a straightforward and effective technique of provoking crashes and reported on its success in [5]. We were able to find 14 crash sequences for Microsoft Word while running the Rogue User during 48 hours[1].

In large SUTs with many – potentially deeply nested – dialogues and actions, it is unlikely that a random algorithm will sufficiently exercise most parts of the GUI within a reasonable amount of time. Certain actions are easier to access and will therefore be executed more often, while others might not be executed at all.

Therefore, in [4] and [5] we presented an algorithm whose idea it is to slightly change the probability distribution over the sequence space. This means that action selection will still be random, but seldom executed actions will be selected with a higher likelihood, with the intend to favour exploration of the GUI.

The strength of the approach is, that it works with large, native applications which it can drive using complex actions. Moreover, the technique does not modify nor require the SUT's source code, which makes it applicable to a wide range of programs. With a proper setup and a powerful oracle, the Rogue User can operate completely unattended, which saves human effort and consequently testing costs.

IV. The study

A. Objective - What to achieve?

The goal of the study is to find out how many previously unknown faults and problems the Rogue User Tool can reveal in a mature, thoroughly tested system. We will not use a System Under Test (SUT) with known or injected faults, but the current version of *ClaveiCon*, as it is shipped to clients of Clavei. This is a more realistic setting than injecting faults, since in production errors are unknown and there is usually no clear definition of what precisely is an error. Consequently, in this study, we will install the Rogue User in the context of the *ClaveiCon* runtime environment, design an oracle, configure the action set, run the tool and report about the problems encountered and lessons learned.

The investigation has been carried out in a fashion similar to the one applied in Iterative Software Development [6]. Since the goal is to develop a working set-up for the Rogue User,

[1] Videos of these crashes are available at http://www.youtube.com/watch?v=PBs9jF_pLCs

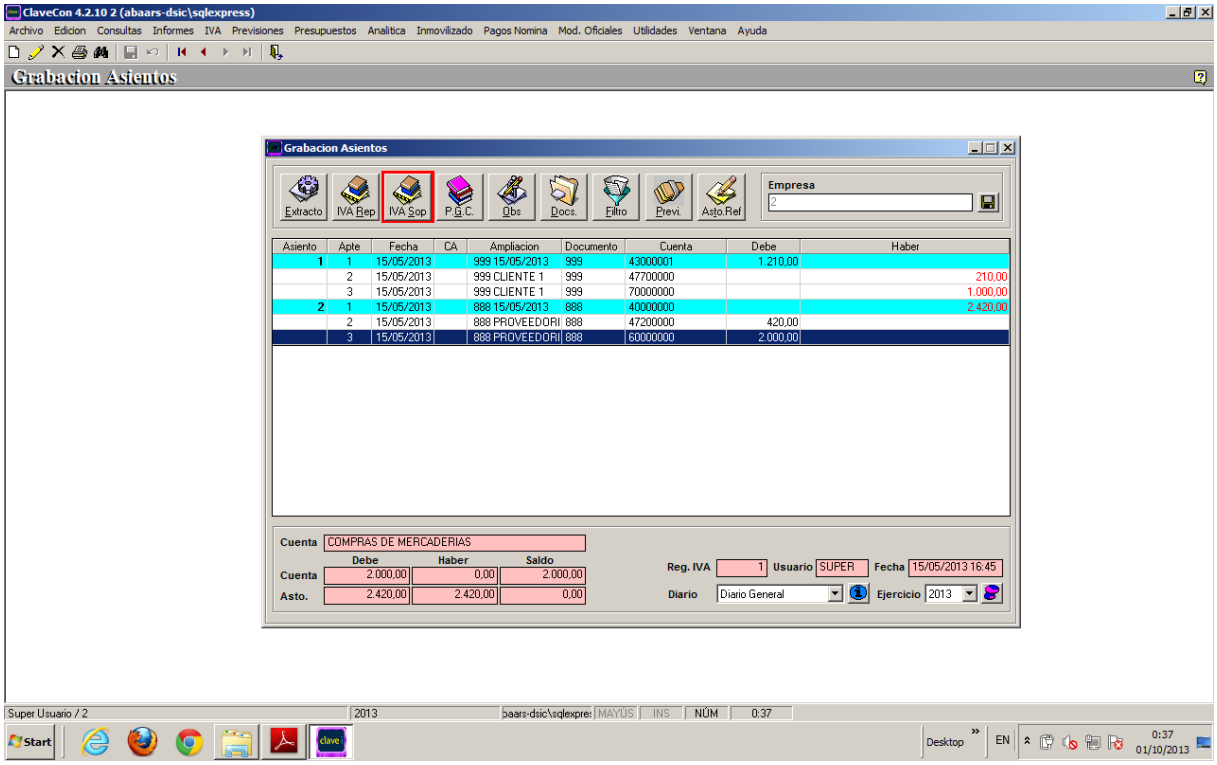

Fig. 1: The SUT: ClaveiCon

Analyze	The RU tool
For the purpose of	Investigating
With respect to	Effectiveness and Efficiency
From the viewpoint	Of the Testing Practitioner
In the context of	The Development of the ClaveiCon accounting system

we reasoned it is necessary to start with a simple base set-up and perform a stepwise refinement of each implementation aspect. Thus we will iteratively perform the phases of planning, implementation, testing and evaluation of the setup. In this iterative process we will:

- Try out different setups for the Rogue User Tool. The goal is to find a setup which allows as much automation as possible while detecting commonly encountered problems and enabling notification and reproduction thereof. In order to strike the balance between a powerful oracle and a truly automatic approach, one has to find a trade-off between an accurate oracle with good fault-detecting capabilities and one which generates few false positives.

- Apply the developed set-up and determine its efficiency and effectiveness. We will record the difficulties that arise during application and find out how much manual labour is actually still necessary during the test.

After having developed the set-up and applied the tool to the SUT, we will have gained valuable insight into specific challenges and problems encountered during a real-world test. This insight will be the basis for future research.

B. *Subjects - Who applies the techniques?*

Since our investigation involves the application of action research, the researchers of the Polytechnic University of Valencia as well as Clavei's developers will collaborate in a joint effort to setup the Rogue User Tool and to test Clavei's accounting system. The subjects are a researcher with practical testing experience and two Clavei test practitioners who worked in the industry for many years.

All three testers will work both, on-site at Clavei as well as communicate over Skype to collaborate and exchange information. The researcher has extensive domain-knowledge about the Rogue User Tool and will thus lead the development of the setup. The two industrial Clavei testers, on the other hand, are familiar with the internals of *ClaveiCon*, so that all three will complement each other.

C. *What will be measured?*

During the investigation we will measure certain aspects of the prototype setup development in order to evaluate the Rogue User Solution. The following aspects were measured in order to obtain indicator values:

1) **Effectiveness**:
   a) Number of failures observed after executing the Rogue User Test on *ClaveiCon*.
   b) Percentage of reproducible failures.
   c) Number of false positives.
2) **Efficiency**:
   a) Time needed to set-up the test environment and get everything running

b) Lines Of Code (LOC) and time needed for defining error definition, oracle design, action definition and design of stopping criteria.
c) Time for running the RU tool
d) Time needed for manual labour after the Rogue User Test has been started. Here we want to find out how much manual work is actually necessary during a full test. This includes adjustments that needed to be made, the evaluation of run results and reproduction of potential faults as well as other manual activities.

D. Protocol for the Set-up Development

Our study has been carried out in a fashion that allowed us to perform iterative development of the Rogue User Setup. This means that we performed a set of ordered steps in a loop. We think that the natural way of setting up our tool involves a mode of operation similar to the one applied in Iterative Software Development [6], which usually repeats the phases of planning, implementation, testing and evaluation in order to achieve increasingly better results. The process included the following steps which were repeated several times to yield the final Rogue User Setup:

1) Planning Phase:
   a) Implementation of Test Environment: Plan and implement the technical details of the test environment for the Rogue User.
   b) Error Definition: Anticipate and identify potential fault patterns.
2) Implementation Phase:
   a) Oracle Implementation: Implement the detection of the errors defined in the previous step.
   b) Action Definition Implementation: Implement the action set, which defines the behaviour of the Rogue User.
   c) Implementation of stopping criteria: These criteria determine when sufficient testing has been done by the Rogue User.
3) Testing Phase:
   a) Run the test.
4) Evaluation Phase:
   a) Identify the most severe problems encountered during the run and reproduce potentially erroneous sequences. The collected information will be used for the refinement of the setup during the next iteration.

V. Results

The development took place over a period of 2 weeks in which the participants performed the activities outlined in Section IV-D. As mentioned earlier, our strategy was to carry out an iterative process, in which we repeated the phases planning, implementation, testing and evaluation until we obtained a viable Rogue User setup. Generally, the first step in setting up a RU Test is to enable the RU to start and stop the SUT. In the case of *ClaveiCon* this simply amounts to specifying the executable. The tool can then execute the program before generating a sequence and kill its process after sequence generation has finished. Here it is important to make sure that the SUT always starts in the same initial state, in order to enable seamless replay of recorded sequences. Therefore, it is necessary to delete potential configuration files which have been generated during previous runs and could potentially restore previous state.

Our next step was to anticipate and define certain error patterns that could occur during the test. For our initial setup we only considered crashes and non-responsiveness, but during later iterations we wrote more fine-grained error definitions which exploited information obtained during previous runs. After we defined our faults, we settled out to implement the Rogue User's oracle. The complexity of the implementation depended on the type of the errors to be detected and increased over time.

Before we were able to run our RU setup, we had to define the actions that the RU would execute. The action definitions determine what controls the RU clicks on, where it types text in and where it performs more complex input, such as drag and drop operations. Those definitions, along with the previously mentioned oracle, can be encoded in the tool's customizable protocol and visualized during runtime for debugging purposes. Figure 2 shows the actions detected by the Rogue User, based on the initial version of action definitions.

This initial version implemented only clicks on most elements. In later versions we included text input and drag and drop operations.

The final ingredient for a finished setup is the stopping criterion which determines when the Rogue User will cease to generate sequences and will finish the test. Our initial setup used a time-based approach which stopped the test after a particular amount of minutes had passed by. Later on we used a combination of several criteria, such as time and amount of generated sequences.

After our first run we already detected a severe fault, in which the SUT refused to respond to any user input. Figure 3 shows this first error, which was the first of two of this kind.

Whenever the Rogue User detected such a fault it logged the information, saved the corresponding sequence and continued with the generation of the next sequence. After the test had been finished, the tester read the logs where she was presented the final results. In case of an error the RU yielded an output such as the one in Figure 4 which corresponds with the error mentioned above. The log told the tester which actions had been executed, where the fault occurred and what the cause was (in this case an unresponsive system).

During a few runs we encountered difficulties with our current setup, such as non-reproducible sequences, unexpected foreground processes or "stalemate" situations in which the RU did not have sufficiently detailed action definitions to proceed the sequence generation. These problematic situations usually manifested themselves within the log files (too few actions, actions sometimes failed since the SUT was blocked by another process, etc.). After each test execution we inspected the logs, the encountered errors and the problems if some occurred. We then continued with the next iteration of development, in order to adjust and improve our test environment, the error

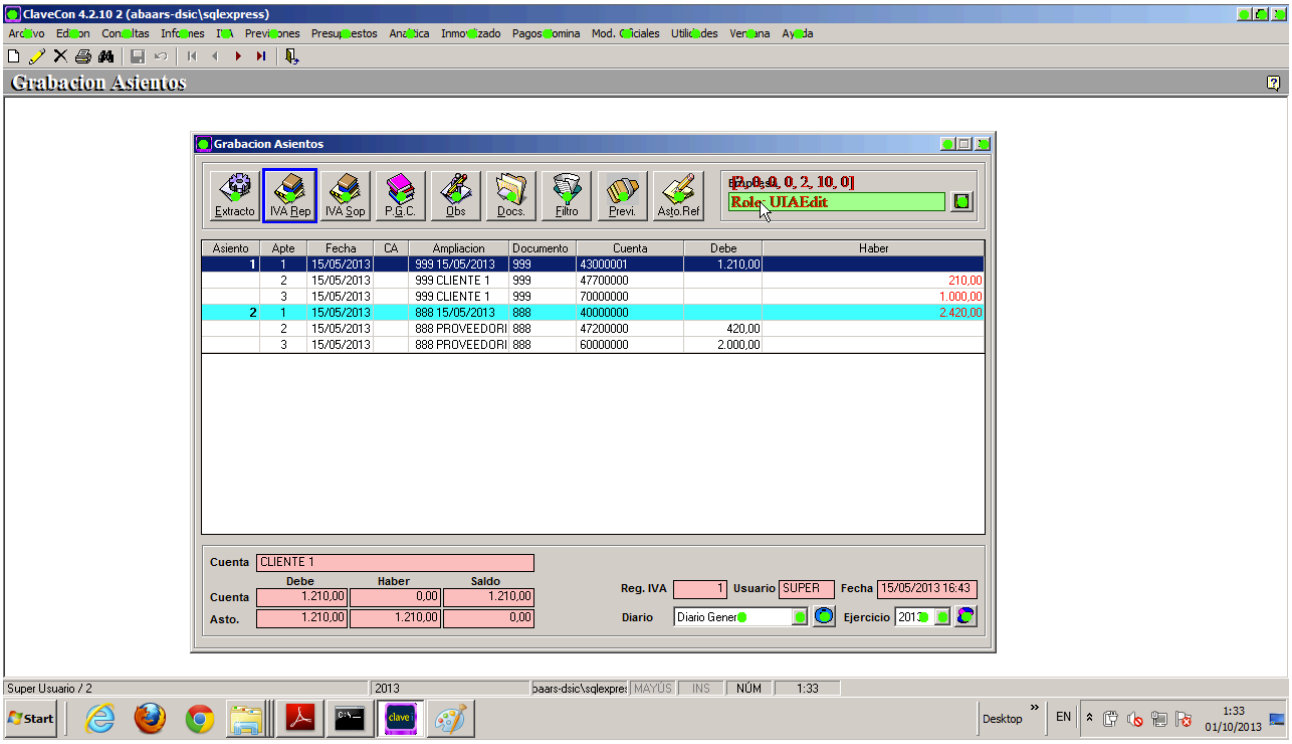

Fig. 2: Action definitions.

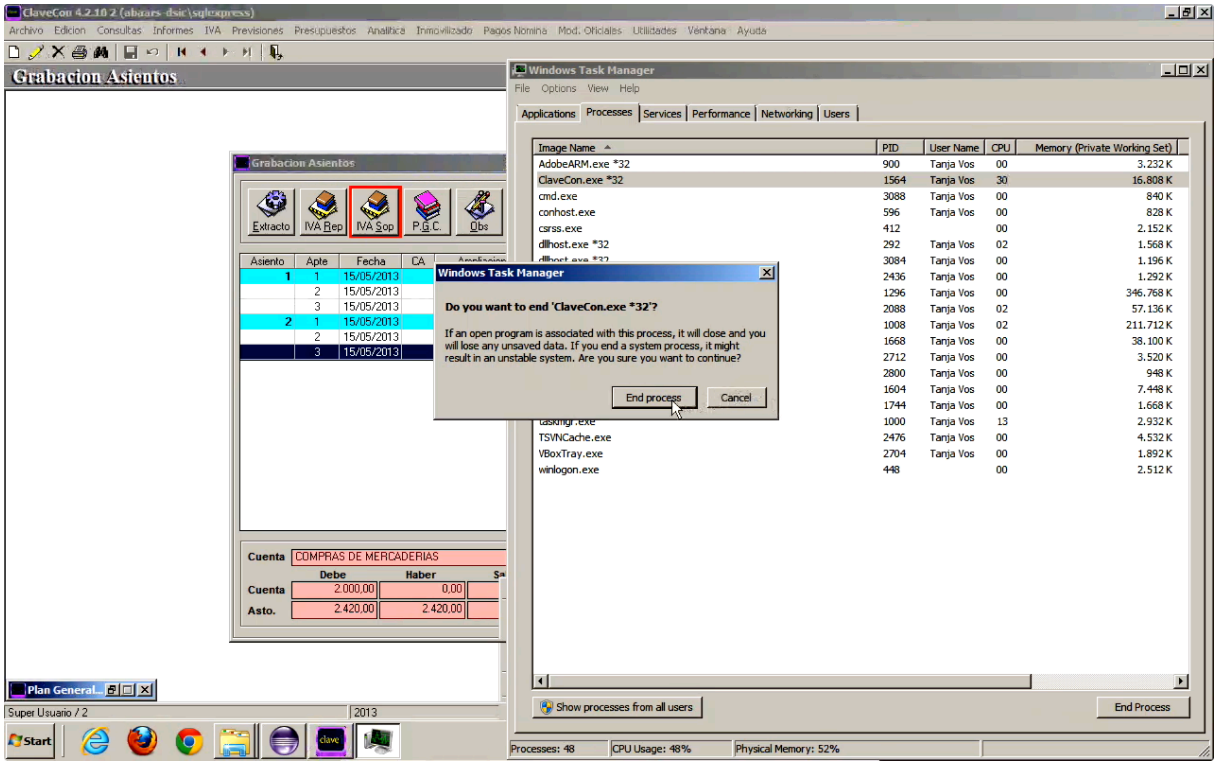

Fig. 3: Easy to detect fault. The SUT froze and did not respond to user input.

```
25.mayo.2013 08:04:27
Hello, I'm the Rogue User!

25.mayo.2013 08:04:28 Starting system...
Starting sequence 0
'Generate' mode active.
Executed (0): Left Click at 'Utilidades'...
Executed (1): Left Click at 'Previsiones'...
Executed (2): Left Click at 'P.G.C.'...
Executed (3): Type 'Test ...' into '...
Executed (4): Left Click at ''...
Executed (5): Left Click at ''...
Executed (6): Left Click at ''...
Executed (7): Left Click at 'Periodo Desde'...
Executed (8): Left Click at ''...
Executed (9): Left Click at ''...
Executed (10): Left Click at 'Minimize'...
Detected fault: severity: 0.8 info: System is unresponsive! I assume something is wrong!
Sequence 0 finished.
Sequence contained problems!
Copying generated sequence ("./output\sequence0") to output directory...
Shutting down system...
```

Fig. 4: The Rogue User's log file for the error in Figure 3.

definitions, the oracle, the action definitions and the stopping criteria. In the following paragraphs we will describe these steps in more detail.

1) Test Environment: The test environment is the skeleton of the Rogue User and guarantees a seamless execution of the generated input sequences without interruption by other sources (e.g. the SUT is always the foreground process, it can be started automatically, it can be stopped reliably in case of a severe error, recorded sequences can be reproduced reliably, ...). The main challenges that we encountered during the development of the test environment were the following:

- Restoring a dedicated start state: This is important in order to make sure that previously recorded sequences can be replayed reliably. If the SUT performs bookkeeping on things like window positions or saves and restores settings of previous runs, then the starting states of the system will vary significantly, rendering sequence replication difficult or impossible. In the case of *ClaveiCon* we discovered several configuration files for various settings. However, it can be non-trivial to find all these files and it took 3 iterations to identify most of them. Moreover, ClaveiCon uses a database to store customer data. This database changed during sequence generation (tables or rows got added, modified or deleted) and it was necessary to restore its original state everytime a new sequence got generated. We implemented our test environment such that the database would be repopulated according to a particular schema. However, this incurred a delay of more than 10 seconds before each sequence, which increased the testing time.
  Other factors were related to the system's clock time, the processor load, memory consumption and thread scheduling, which varied for each sequence and which could not be controlled. In one case we were not able to reliably replay an erroneous crash sequence. Sometimes the sequence failed, and at other times it went through without problems. This "indeterminism" sometimes impeded the testing process.
- Unexpected third-party processes. ClaveiCon is a standalone application and usually includes everything that is needed to work with it. However, it has functionality that invokes external applications, such as word processors to view exported files or a help system. Since these are not part of the process under test, the Rogue User will ignore them. Unfortunately, these processes can occur in the foreground and thus block access to the SUT, preventing the RU from properly generating sequences. During the development we encountered several such situations and implemented functionality to terminate the applications when encountered.

In summary, the development of a stable test environment is not complicated but requires a certain amount of trial-and-error and some knowledge about the SUT. A proper replay of each recorded sequence cannot be guaranteed, but the more robust the environment is, the better the likelihood of a replay. Later in the text we will see that almost all erroneous sequences, but one, were reproducible.

2) Error Definition: Before we started to implement the actual oracle, we brainstormed on potential errors that could occur within the SUT. The following list describes the ones that we came up with during the development:

1) Crashes: If the process associated with the SUT suddenly stopped, we considered it to be a crash. A problem that can occur with this definition is, that certain actions that the Rogue User executes will inevitably shut down the SUT, e.g. the "Exit" menu item. Obviously, those are no crashes. Our approach was to consider these cases in the action definition of

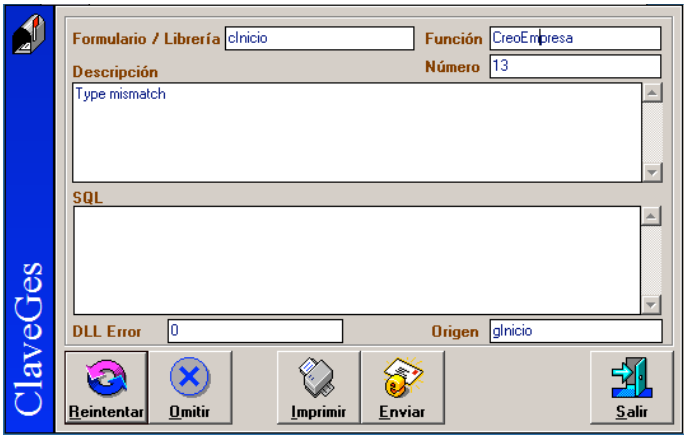

Fig. 5: Dialog triggered by an exception with easy to detect error message.

the Rogue User. We simply disallowed actions that would terminate the SUT.

2) Freeze (unresponsiveness): If the SUT did not respond for a specific amount of time, we considered it to be dead-locked.
3) Exception: *ClaveiCon* was programmed in such a way that, whenever an exception is thrown that "bubbles up" to the system's main function, it will generate an error dialog with the error string, an error number and the name of the function that caused the exception. Figure 5 shows such a dialog. As described in [4] and [5] the Rogue User creates a so-called widget tree for each state that the SUT is in. This tree contains information about the SUT's current screen layout, the coordinates of each control element and its properties, such as its name or text content. This makes it possible to detect when a dialog such as in Figure 5 appears and to report an erroneous sequence. We simply applied regular expressions that searched for particular error strings within the widget tree. However, we encountered several situations where this triggered a false positive, since other control elements had names that corresponded with the error strings, e.g. a button called "Report Error". Therefore, we had to implement certain rules and first analyze the type of control element that included the error string.
4) Layout Error: Another error type which we detected during later iterations were layout errors. Figure 8 shows a clipping error within *ClaveiCon* where two dialogs fight over the drawing order and leave an abnormal representation on the screen. These types of errors can be detected by checking the value for the z-orders of the dialog elements. If two dialogs exhibited the same z-order value, this often resulted in this visible defect.

It is generally difficult to anticipate and properly define many errors since one does not know what to look for. Crashes and freezes are easy to detect, but the exception and layout errors we noticed during test runs and later implemented their detection. A thorough knowledge of the SUT can help and so the fact that two of the testers were also developers of ClaveiCon certainly helped. However, due to the oracle problem [7] it is hard to define many error patterns thoroughly and ahead of time.

3) Oracle Implementation: In order to implement the error detection as defined in the last paragraph, the Rogue User Tool provides a customizable protocol with hooks for specific tasks, such as error detection. The implementation has been done in the Java Programming language. Our oracle operated solely on the widget tree for each state, which was already provided by the RU Tool. The tree provides information about the SUT (such as whether it is running, etc.) and its current screen state (position and properties of all visible control elements). This allowed a straightforward implementation of the previously stated detection rules. When the oracle reported an error, it attached an error message and a priority value (see Figure 4). The error message then appeared in the error log and the priority value was used to order the detected erroneous sequences, so that we were able to inspect the most promising ones first. This turned out to be useful, since we encountered a few false positives during our tests, such as errors triggered by too general regular expressions for error strings. We gave crashes and freezes higher priority so that they were reported first.

After the error definition step, the implementation of the oracle was usually straightforward and free of any complications.

4) Implementation of Action Definitions: As with the oracle, the Rogue User Tool provides a hook for action definitions. The tool comes with a variety of predefined actions and allows to combine those to form more complex ones such as mouse gestures. The position and control type information within the widget tree allowed us to tailor action definitions to specific types of controls. During the first iteration we started off with clicks on enabled button and menu item controls. The tool provides a mode in which we were able to debug our definitions, by visualizing them. Figure 2 shows the definitions for our initial setup. In addition, the tool provides a so-called "spy mode" (see Figure 7) which allows to inspect the properties of specific control elements. This information has been valuable when we tried to exclude actions to specific elements such as clicks to the "Exit" menu item, which terminates the SUT prematurely.

The following is a list of challenges that we encountered when implementing the action definitions for the Rogue User:

1) Undetected control elements: Certain special or custom controls were not detected by the Rogue User. Consequently, these controls did not appear in the widget tree, which made it harder to write action definitions for them, due to e.g. unavailable positional information. This amounted to only a few control elements, such as the items in the tool bar below the main menu in Figure 2. However, we were able to write code that estimated the position of these items and thus able to generate actions for them.
2) Exclusion of unwanted actions: Certain actions caused potentially hazardous effects to the host computer. *ClaveiCon* has menu items that create files to export data, or open file dialogs in which one

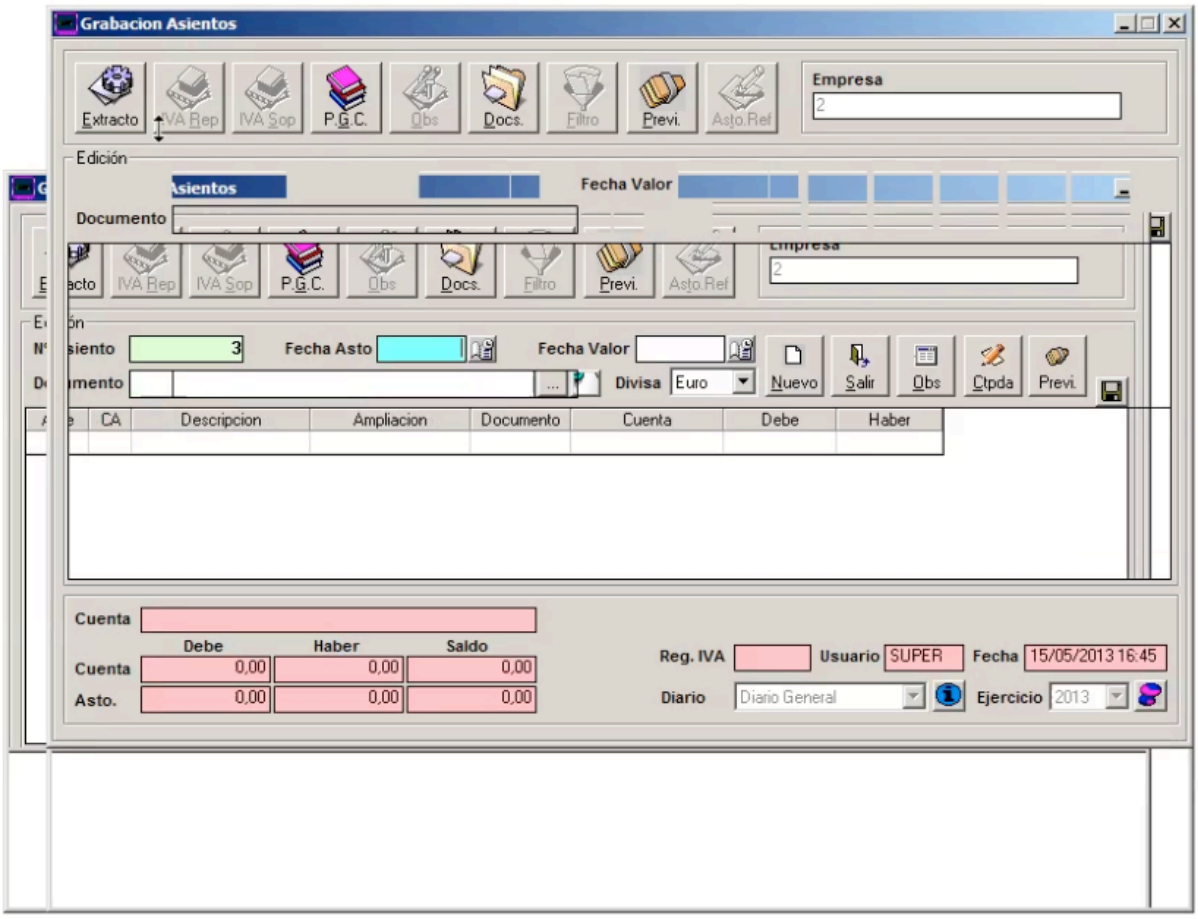

Fig. 6: Clipping error. Detectable through inconsistencies within the widget tree.

can move or delete files and directories. We first had to prohibit certain actions manually in order to guarantee the integrity of the hard disk. In later iterations we opted for another option: We ran our tests on a different system user account with restrictive directory rights. This reduced the complexity of our action definitions while guaranteeing the integrity of the machine.

3) Insufficient action choices: We observed sequences in which the Rogue User navigated into dialogs and was unable to leave those, due to a lack of available actions. We countered this by allowing the tool to perform certain actions such as hitting the escape key after specific time intervals.

5) Stopping Criteria: The stopping criterion determines when a test will finish, i.e. when the Rogue User will cease to generate new sequences. For our initial setup we applied a time-based approach with a maximum test time of 12 hours. For the final setup we used a combination of time and number of generated sequences. The reason for this is that certain sequences take longer to execute, so that the amount of generated sequences can vary largely. Since the tool uses a search-based algorithm which tries to explore the GUI of the SUT as thorough as possible, this can affect the test quality. Thus, we made sure that the Rogue User would generate a minimum of 200 sequences (with 200 actions each) per run, a setup which we successfully applied during earlier case studies.

6) Test Execution: During each iteration, after adapting the Rogue User setup, we conducted an overnight test run. Most of the time this happened completely unattended. However, during the first 2 iterations we observed the test for a while in order to spot potential problems. During later runs we only inspected the tool's log file which records each executed action and other information, along with potential problems.

We also watched the sequence generation for a small amount of time in order to spot errors not detected by the current oracle or to get new ideas. This is how we found definitions for the encountered layout errors.

7) Evaluation: During the mornings we read the log files and inspected potentially erroneous sequences. Most of the time we spent reproducing those. If this was not possible, then the tool still provided the option to view an ordered set of screenshots of the sequence's actions, which enabled us to understand what happened.

During the first few iterations we encountered problems with unexpected foreground processes or non-reproducible sequences due to the fact that our test environment did not delete hidden configuration files. We learned from each iteration and

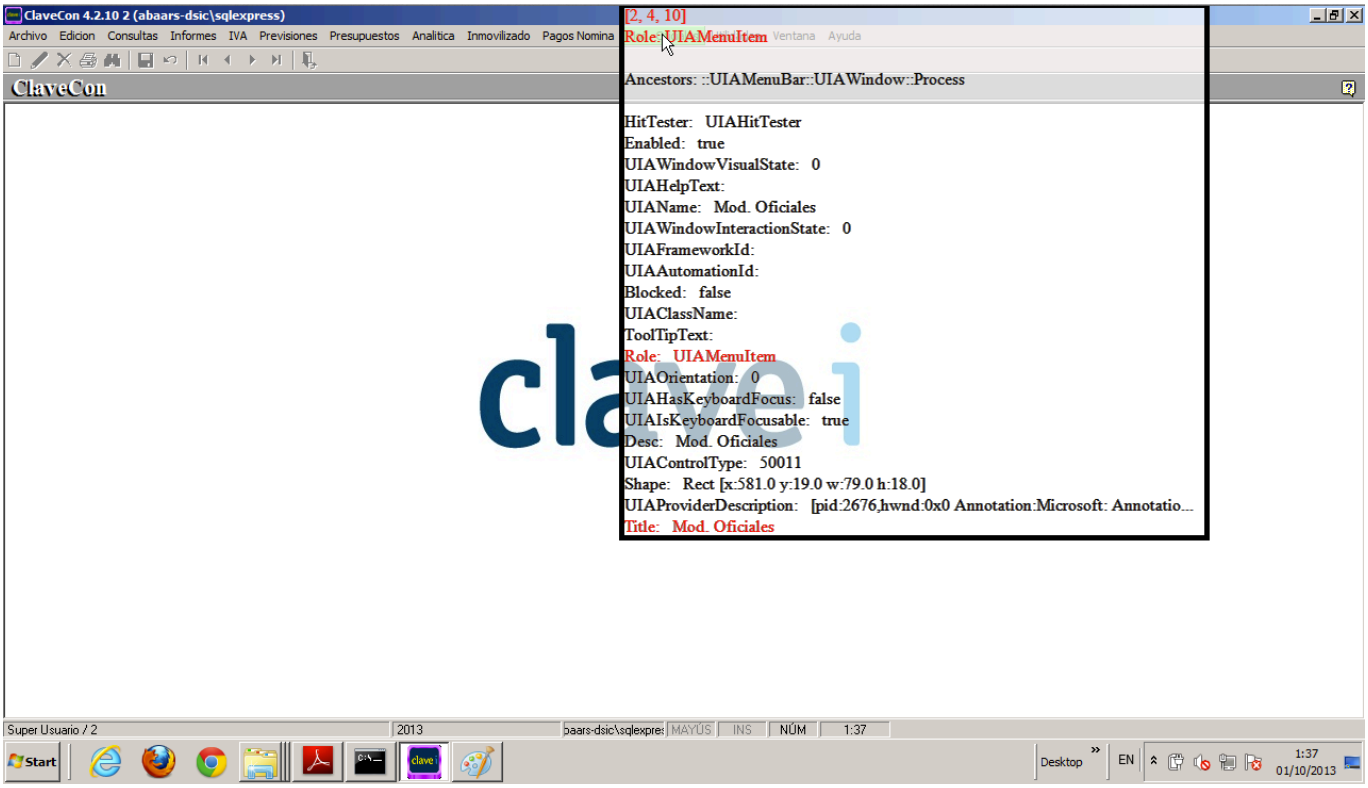

Fig. 7: Spy mode.

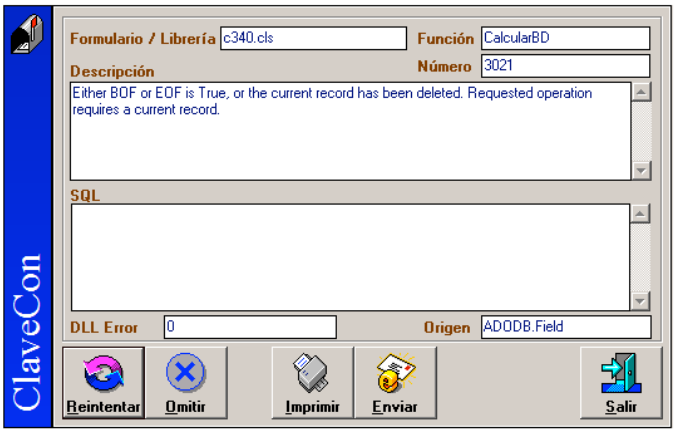

Fig. 8: Error dialog triggered by an internal exception.

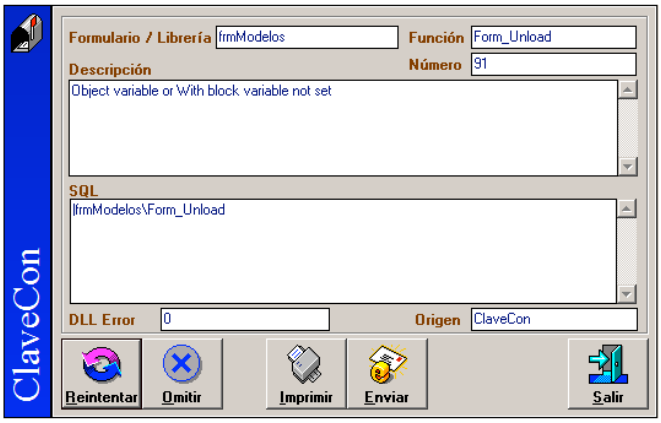

Fig. 9: Error dialog triggered by an internal exception.

improved the test environment as well as the other parts of the RU's setup. We recorded the time we spent doing manual work during and after each test. The last row of Table II shows this relatively low value.

A. *Final Results*

Table I lists the values for the effectivity indicators, i.e. the amounts of errors of different type that we encountered during all of our test runs. According to the Clavei developers, all of these were triggered by different code segments within *ClaveiCon*'s source code. With one exception we were able to reproduce each of the failures. The one problematic sequence can be replayed, but the fault will not be triggered. We currently do not know the reason for this, but we suspect it to be related to the thread-scheduling of the system and an erroneous piece of multi-threaded code. In summary, the RU Tool was able to detect 10 previously unknown faults, which is an encouraging result. The high number of false positives can be explained due to problems with the initial oracle in the first two development iterations, where 7 of the false positives where triggered due to a too general regular expression for the exception dialogs.

Error Type	Amount	Reproducible?
freeze	2	1/1
crash	1	1/1
exception	6	5/6
layout	2	3/3
false positives	8	

TABLE I: Indicator values for effectivity (list of encountered errors).

Activities	Indicator	Value
LOC for the RU Setup	2a)	1002
Implementation of test environment	2b)	340
Error Definition	2b)	140
Oracle Implementation	2b)	490
Action Definition Implementation	2b)	560
Implementation of Stopping Criteria	2b)	40
Total Development Time	2b)	1570
Test Duration	2c)	5490
Manual intervention during and after Test Runs	2d)	100

TABLE II: Indicator values for efficiency (over all iterations)

Table II shows the results for the efficiency indicators. Accumulated over all 5 iterations it took approximately 26.2 hours of development time to yield the final setup for the Rogue User. The majority of the time was spent on the oracle and action definition implementation. However, the final setup could be replayed over longer periods of time and could thus reveal more faults with only minimally more human intervention. In a possible scenario one would run the setup during weekday nights and check the RU Tools' log file during the following mornings. The low value of indicator 2d) indicates that the tool needs only very little human attention which in our case amounted to looking for potential problems in the log file and reproducing the faulty sequences. The tool cannot detect every error type (at least not with reasonable development effort), but it can detect certain critical errors with very low effort. The initial effort in developing the test setup pays off as testing time increases, as it can be applied arbitrary amounts of time.

VI. Conclusions and Lessons Learned

In this document we presented the results of an investigation that we carried out together with Clavei, a software vendor located in Alicante, Spain. We settled out to perform a real-world test with a previously unknown SUT. Our goal was to obtain knowledge about the challenges encountered in setting up such a test and to gather fundamental information for more detailed future research.

We performed the development of the setup in an iterative fashion, since we think this is the traditional way to gain feedback about its quality during each iteration and enables the testers to continuously improve the test environment, incorporate new ideas and fix previous problems.

One of the challenges that we encountered was the problem of reproducing (erroneous) sequences. It requires a thorough test environment with ideally identical conditions during a sequence's recording and replay time. Unfortunately, most complex SUTs are stateful and save this state within databases, configuration files or environment variables, which complicates the development of a test environment that guarantees traceable and deterministic sequence generation and replay. An interesting starting point would be to execute the SUT in a virtual machine environment, which would allow to restore most of the environmental conditions. One would have to deal with larger memory requirements and a time-overhead for loading the VM snapshots, but today's large and fast hard disks might make this problem tractable. However, for more distributed SUTs whose components live on multiple machines, this might not be a viable option and would call for additional solutions.

Another challenge was the development of a sufficiently powerful oracle. We started off completely blind without any ideas for potential errors. Our ideas developed during later iterations and with greater knowledge about *ClaveiCon*. However, we think that the types of errors we found and the error definitions we used, might be applicable to other SUTs as well. An idea could be a collection of "canned" error patterns that a tester who uses the Rogue User could start off with and refine.

To sum up, the development of an effective and efficient setup for the Rogue User takes some initial effort (in our case approximately 26 man hours) but will pay off the more often the test is run. The manual labor associated with a test breaks down to the inspection of log files, reproduction and comprehension of errors and makes only a tiny fraction of the overall testing time (we spent around 100 minutes of manual intervention during and after tests, compared to over 91 hours of actual unattended testing). This, combined with the fact that the Rogue User detected 10 previously unknown critical faults, makes for a surprisingly positive result and animates us to do more thorough case studies to evaluate that the technique is a valuable and resource-efficient supplement for a manual test suite.

Acknowledgment

This work was financed by the FITTEST project, ICT-2009.1.2 no 257574.

References

[1] Z. U. Singhera, E. Horowitz, and A. A. Shah, "A graphical user interface (gui) testing methodology," *IJITWE*, vol. 3, no. 2, pp. 1–18, 2008.

[2] B. N. Nguyen, B. Robbins, I. Banerjee, and A. M. Memon, "Guitar: an innovative tool for automated testing of gui-driven software," *Autom. Softw. Eng.*, vol. 21, no. 1, pp. 65–105, 2014.

[3] C. Kaner, "Avoiding shelfware: A managers' view of automated gui testing." www.kaner.com/pdfs/shelfwar.pdf, 2002.

[4] S. Bauersfeld and T. Vos, "A reinforcement learning approach to automated gui robustness testing," in *In Fast Abstracts of the 4th Symposium on Search-Based Software Engineering (SSBSE 2012)*. IEEE, 2012, pp. 7–12.

[5] S. Bauersfeld and T. E. J. Vos, "Guitest: a java library for fully automated gui robustness testing," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012. New York, NY, USA: ACM, 2012, pp. 330–333. [Online]. Available: http://doi.acm.org/10.1145/2351676.2351739

[6] C. Larman and V. Basili, "Iterative and incremental developments. a brief history," *Computer*, vol. 36, no. 6, pp. 47–56, 2003.

[7] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2008.