2016 IEEE International Conference on Software Testing, Verification and Validation

A Framework for Monkey GUI Testing

Thomas Wetzlmaier, Rudolf Ramler
Software Competence Center Hagenberg GmbH
Softwarepark 21, A-4232 Hagenberg, Austria
{thomas.wetzlmaier, rudolf.ramler}@scch.at

Werner Putschögl
Trumpf Maschinen Austria GmbH + Co. KG
Industriepark 24, 4061 Pasching, Austria
werner.putschoegl@at.trumpf.com

***Abstract*—Testing via graphical user interfaces (GUI) is a complex and labor-intensive task. Numerous techniques, tools and frameworks have been proposed for automating GUI testing. In many projects, however, the introduction of automated tests did not reduce the overall effort of testing but shifted it from manual test execution to test script development and maintenance. As a pragmatic solution, random testing approaches (aka "monkey testing") have been suggested for automated random exploration of the system under test via the GUI. This paper presents a versatile framework for monkey GUI testing. The framework provides reusable components and a predefined, generic workflow with extension points for developing custom-built test monkeys. It supports tailoring the monkey for a particular application scenario and the technical requirements imposed by the system under test. The paper describes the customization of test monkeys for an open source project and in an industry application, where the framework has been used for successfully transferring the idea of monkey testing into an industry solution.**

Keywords—GUI testing; random testing; test automation tool.

I. INTRODUCTION

Testing a complex software system via its graphical user interface (GUI) is a challenging endeavor. The GUI is intended to provide comfortable, interactive access to the system's functionality for human users. It introduces a layer of abstraction that hides the technical details of the underlying implementation while it increases the ease and flexibility in how users perform their tasks. Unfortunately, the additional abstraction and the increased flexibility make testing much more difficult. Moreover, the growing size and complexity of modern GUIs becomes a new source of critical defects that also has to be taken care of in testing.

GUI testing is usually a labor and resource intensive task as in practice a large part of this type of testing is still performed manually. In response, numerous techniques, tools and frameworks have been developed for automating GUI testing [1]. The primary goal of most of these tools lies in supporting the automated execution of GUI (regression) tests, which are typically created via recording or programming. Among the many technical features provided by these tools are the correct recognition of elements on the GUI and the interaction with these elements by sending appropriately timed events. The constantly evolving landscape of GUI technologies and widget libraries makes implementing and maintaining these features a continuous challenge for tool developers.

Ongoing development and continuous evolution of a software system under test (SUT) raises similar challenges for the developers of automated GUI tests. When a feature of the SUT is changed, the corresponding automated tests have to be changed as well. The maintenance problem in test automation is exacerbated in GUI (system) testing as these tests are affected by changes of almost any aspect of the system ranging from its functionality to the layout of the GUI. In many projects, thus, test automation did not reduce the overall effort of testing; it has merely been shifted from executing manual tests to developing and maintaining automated tests.

Random testing approaches [2] have been suggested as a way to support testing in such adverse situations. In context of GUI testing, the term "monkey testing" has been coined by Nyman [3], who described a practical approach for automated random exploration of GUI applications with test monkeys as an aid for black-box system testing. Further success stories have been reported by Fodeh [4], Hofer et al. [5], Forrester and Miller [6] as well as others[1]. Their results confirm the applicability of monkey testing in practice and encourage its transfer to industry scenarios.

The body of literature also shows a diversity of solutions for automated GUI testing due to the fundamental differences in the functionality, architecture and technical implementation of GUI applications. This observation matches our experiences in developing support for test automation as documented, e.g., in [7] and [8]. In almost all cases the implementation of testware has been found to be closely related to the implementation characteristics of the corresponding SUT.

In this paper, thus, we present a widely adaptable framework for monkey GUI testing. The framework provides a set of reusable components and a predefined, generic workflow with several extension points for the development of different monkey test applications. Hence, individual test monkeys can easily be tailored to meet the requirements of a particular application context and the SUT's technical peculiarities. Section II of the paper provides a detailed overview of the monkey framework. Concrete applications of monkey GUI testing including customized test monkeys are shown in Section III (experimental application) and IV (industry application), which correspond to consecutive stages of transferring the idea of monkey testing into an industry solution. The paper concludes with a discussion of related work (Section V) and a summary of our findings and planned future work (Section IV).

[1] http://staq.dsic.upv.es/sbauersfeld/papers/inprogress.pdf

978-1-5090-1827-7/16 $31.00 © 2016 IEEE
DOI 10.1109/ICST.2016.51

II. FRAMEWORK OVERVIEW

The *Monkey Framework* has been designed to support the custom implementation of different monkey test applications ("test monkeys"), each tailored to the specific testing needs and technical characteristics of a particular SUT. Therefore the framework provides a set of reusable components that represent key features (see subsection A) required for monkey testing and a predefined, generic workflow (B) with several extension points to adapt and specialize the behavior with individual guide sequences and interaction strategies (C).

The framework is the basis for building a specific *Monkey Test Application.* The typical usage context of the resulting application is shown in Fig. 1. It is called by a *Test Runner* and uses a *GUI Testing Tool* to exercise the *SUT.*

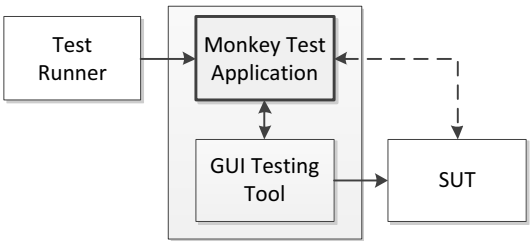

Fig. 1. Typical usage context of a Monkey Test Application.

A. Key Features

Monkey test applications built with the monkey framework inherit several key features that support automated testing of the SUT via randomly exploring its GUI.

Accessing GUI Elements: Currently a wide range of different GUI technologies are available and new ones are frequently introduced. A large share of the effort invested in the development of GUI testing tools is consumed by keeping up with the fast-paced development of GUI technologies. Today many open source and commercial GUI testing tools provide specialized support for one or more GUI technologies. Therefore, instead of creating yet another technology specific implementation, the monkey framework has been designed to harness available GUI testing tools for accessing and controlling GUI elements (Fig. 1). Currently the framework includes an adapter for the test automation tool Ranorex[2] that provides up-to-date support for all major industry-relevant GUI technologies.

Multi-level Setup and Teardown: For exercising a SUT via automated GUI tests, the SUT as well as its runtime environment have to be setup into a predefined state. A custom-built setup procedure (e.g., for setting configuration files, initializing the test database, or starting up services) is provided. It is executed prior to the launch of the SUT. A corresponding teardown procedure can be implemented for cleaning up after the test run to roll-back any changes. The framework implements standard operations for process control such as starting and stopping the SUT. Setup and teardown procedures are also well supported by most freely available unit testing frameworks (e.g., NUnit, MbUnit). For realizing complex, multi-level setup and teardown scenarios, the monkey test application can therefore be combined with a *Test Runner* from on one of these frameworks (see Fig. 1), which also allows easy integration in a build or continuous integration environment.

Predefined Workflow and Extension Points: The monkey framework implements a predefined workflow that includes the setup and startup the SUT, a basic operation cycle for randomly interacting with the GUI, continuous monitoring of the SUT, and a final teardown and cleanup in case a failure has been detected or the interaction limit has been reached. The predefined workflow contains several extension points for customizing the monkey test application to a specific SUT. A typical example is the need for executing a predefined interaction sequence when a particular situation is encountered, e.g., a login dialog requesting a password. Hence, instead of generating random input for the password, a predefined value is used. A detailed description of the workflow and its extension points can be found in the next subsections.

Failure Detection: Observing and interpreting the behavior of the SUT is important for detecting failures. The monkey framework implements various generic checks for detecting application failures such as crashes, unhandled exceptions or violations of assertion indicated by a modal exception dialog. The monkey framework is capable of detecting and handling such dialogs. Exceptions may also be logged to the standard error output stream or to application specific log files. The framework supports monitoring append operations to error streams and log files. Furthermore, a failure of the SUT may also be detected by monitoring of system performance measures as explained in the next section.

System Monitoring and Metrics: In parallel to exploring the GUI of the SUT, the monkey framework monitors the "health" and performance of the SUT and the underlying operating system. An observed excessive resource usage may be an indicator for a failure of the SUT or it may be the explanation for an unexpected behavior of the SUT if caused by an interfering application (e.g., automatic updates). The monkey framework supports monitoring performance measures for *process*, *system*, *processor* and *memory* (e.g., processor queue length, user time, thread count) to reveal well-known bottlenecks[3]. The monitored measures are logged for a later trend analysis and for analyzing the root cause of detected failures. In addition, a detailed log about the interaction of the monkey test application with the SUT is kept together with screenshots captured from the dialogs accessed during the exploration of the GUI. Besides being a debugging aid, these recordings are used to compute reliability metrics such as "number of random interactions between failures" suggested by Fodeh [4].

B. Generic Workflow

The framework defines a generic workflow implementing a basic operation cycle for randomly interacting with the SUT until a failure has been detected or the interaction limit has been reached. Fig. 1 provides an overview of the workflow and the involved activities, which are described in the following.

[2] http://www.ranorex.com

[3] https://technet.microsoft.com/en-us/magazine/2008.08.pulse.aspx

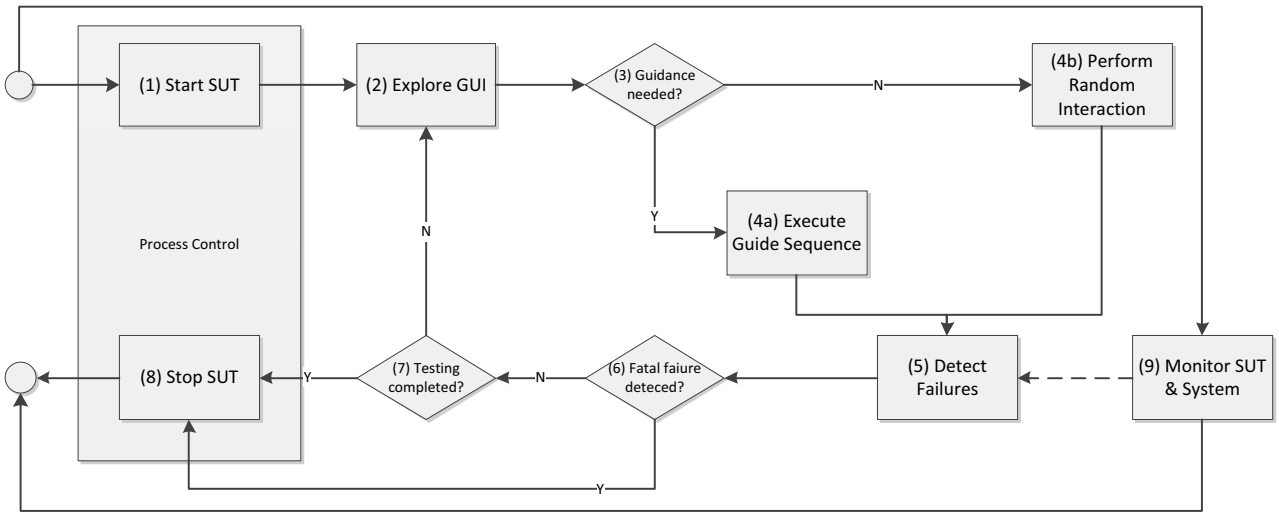

Fig. 1. Activities and decisions in the generic workflow of a monkey test application (straight arrows show control flow, dashed arrows show data flow).

(1) Start SUT: The first step is the setup and start of the SUT. As the SUT runs in a separate process, the monkey has to be synchronized with the completion of the SUT's startup, i.e., it has to wait until the SUT is ready for interaction. A first synchronization point is the appearance of the main window. However, many systems show a splash screen to signal the user that the application is still initializing. Usually the splash screen disappears and the actual main window is shown when the application is ready for use. In order to prevent interacting with the splash screen, the startup procedure needs to be customized, for example by specifying an additional delay.

(2) Explore GUI: The exploration of the GUI starts with enumerating all windows that belong to the SUT's processes and with determining the top-most window. This window is the starting point for GUI ripping [9], which traverses the hierarchy of the GUI elements and extracts information from element (e.g., widget type, name, ID, text, size and location on the screen). The information that can be extracted depends on the UI technology and the GUI element type (e.g., the window text attribute is only available for window elements). Irrelevant element types (e.g., panels) or attributes (e.g., window handles) can be excluded from ripping.

(3) Guidance needed? The extracted information is analyzed in order to identify situations that require a "guide", a predefined interaction sequence. For example, a login dialog requires that a valid username and password are entered in order to proceed. The presence of a login dialog is recognized by its name or the text it contains. In general, the trigger for applying a guide is specified as XPath expression, which resembles a search string matching one or more GUI elements. This step is a decision point in the overall workflow. If a match has been found, the corresponding guide is executed (step 4a). Otherwise the monkey continues with its default behavior, i.e., selecting and performing random interactions (step 4b).

(4a) Execute Guide Sequence: If a guide has been triggered in the exploration of the GUI, the custom code of the specific guide is executed. The framework does not have any restrictions about how the guide is implemented. Any executable program or script may be called, including recordings created with the underlying GUI testing tool. This flexibility allows handling all possible situations ranging from simply filling input fields with predefined values (e.g., username and password) to manipulating the SUT's environment (e.g., programmatically inserting records in the test database).

(4b) Perform Random Interaction: The default workflow of the monkey includes randomly interacting with the SUT via its GUI. An interaction can be sending keyboard events (e.g., key pressed), shortcuts (i.e., hotkeys), mouse events (e.g., click, double click) as well as touch events and gestures [8]. Furthermore, combinations of interactions can be specified in form of interaction strategies (see below). Most interactions need a specific GUI element to be performed on. Furthermore, these elements have to meet specific conditions in order to be usable for interaction (e.g., enabled, visible, valid screen rectangle). Modern GUIs often use several layers to realize highly-interactive GUI concepts such as scrollable side-panels and overlays. The elements placed on the different layers may all be accessible for the monkey. Determining which elements are actually usable (e.g., not covered by an overlay) and randomly selecting one of the available elements is the first part of this step. The second part is the random selection of one of the possible interactions that can be performed on the element. The monkey framework provides a set of predefined interactions for the most commonly-used elements (e.g., *Button*: press the button, *MenuBar*: select a random menu item, *Text*: enter a random number of random characters). However, the behavior can be tailored by adjusting the weights for selecting elements and by providing custom interaction strategies (e.g., to generate only numeric input for text fields).

(5) Detect Failures: After interacting with the SUT, the monkey analyzes the state of the SUT to determine if the interaction has triggered a failure such as a "crash", an unhandled exception, or a deadlock/timeout. The monkey framework provides several checks to detect failure situations. It is able identify exception dialogs opened by the application or the underlying runtime environment as well as assertion violations indicated by a special exception dialog when the SUT has been compiled as debug build. The framework supports monitoring append operations to error streams and log files, which can be filtered by regular expression pattern matching to determine failure events. A crash may result in a "hang", i.e., the SUT does not respond anymore and a specified timeout occurs. Similar failures can be detected by observing the system performance measures (e.g., CPU and memory consumption) exceeding thresholds. All detected failures are logged together with the available information about the performed interaction and the state of the GUI (e.g., screenshot).

(6) Fatal failure detected? If a failure has been detected, it has to be determined whether the failure is fatal and requires the SUT to stop. In response to a fatal failure the monkey quits testing and terminates the SUT (step 8). For example, the appearance of an exception dialog indicates a fatal failure (e.g., NullPointerException) as there is no way to handle the exceptional situation. If a failure resulted in an entry in an error log file, it may be counted as non-fatal and the monkey continues.

(7) Testing completed? As long as no fatal failures have been detected the monkey continues testing until a specified number of interactions has been reached. The current implementation of the monkey randomly exercises the SUT without defining a specific target function. Therefore the supported stop criteria are either detecting a fatal failure or the SUT has "survived" the specified number of interactions. When the monkey has finished testing, it stops the SUT (step 8).

(8) Stop SUT: The first attempt to stop the SUT is closing its main window. As it runs asynchronously, the monkey waits until the process has been stopped. If the SUT "hangs" (i.e., it does not stop within a specified timeout) the monkey tries to terminate the process of the SUT. Once the SUT has been stopped, the monkey performs the teardown procedure to clean up the remains of the test run (e.g., delete files, shutdown remote systems). The implemented process control mechanism also monitors the system process of the monkey itself to detect an unexpected termination of the monkey or its sub-processes (e.g., GUI testing tool) and to ensure that the monkey and all related sub-processes are finally stopped too.

(9) Monitor SUT & System: The system where the SUT runs and the SUT are continuously monitored and a range of "health" and performance measures are collected. The monkey writes a detailed log during the whole test process including information about each action performed in the workflow, the accessed GUI elements (i.e., XPath and screenshot) and the related timing (i.e., how long the interaction took). Each entry contains a timestamp so different logs can be correlated. In case of a failure the logs can provide valuable information for manually analyzing and reproducing the failure scenario in debugging. The analysis of the collected information is also an important source for adopting the monkey for a particular SUT (e.g., implementing custom guide sequences) and for fine-tuning the monkey to increase its fault detection capability (e.g., adding interaction strategies).

C. Guides Sequences and Interaction Strategies

All parts of the monkey framework and each activity of the generic workflow presented above are intended to be adaptable in order to build a specific, concrete monkey test application. Nevertheless, two extension points are provided for adding custom code that controls the interaction of the monkey with the SUT. First, in step 4a (see Fig. 1) individual *guide sequences* can be used and, second, in step 4b custom *interaction strategies* can be included. Although the implementation of the custom code may be similar for both extension points, each of them serves a different purpose.

Guide Sequences: The purpose of a guide sequence is to override the default random exploration approach and to enforce a predefined sequence of interactions to handle a specific situation. Thus, the "guide" directs the monkey safely through a critical passage. Typical situations that require guidance are, for example, password dialogs requesting a valid credential, system dialogs (e.g., for opening/saving files) or input fields that are validated against business rules. Guides may increase the GUI coverage as they enable the monkey to transition to previously inaccessible areas of the SUT. However, they may also block access to undesirable parts of the SUT (e.g., functions that are still under development and should be excluded from testing) or prevent interactions that call unsafe operations (e.g., deleting files, destroying system settings, or repeatedly trying incorrect username/password combinations and getting locked out). If a guide is applicable in a specific situation, its usage is mandatory.

Interaction Strategies: The purpose of implementing custom strategies is to improve and extend the monkey's random exploration capability by incorporating additional, SUT-specific knowledge (e.g., generate numerical input for text fields, perform drag and drop operations on a canvas, perform random clicks on unrecognized GUI elements). As a result, the monkey should perform better in testing as it reaches new, yet uncovered areas of the GUI and triggers new, so far unknown types of failures. A great source of inspiration for implementing interaction strategies that try to provoke failures are the seventeen "user interface attacks" described by Whittaker [10]. These strategies aim to "break" the SUT by forcing it into inconsistent or illegal states. Example attacks are "overflow input buffers", "explore allowable character sets and data types", "apply inputs using a variety of initial conditions", or "force a data structure to store too many/too few values".

III. Experimental Application

The first evaluation of the monkey framework was conducted in a lab setting using the open source system *KeePass* as representation of the SUT. The motivation for the initial experimental application was to receive early feedback about the feasibility and scalability of the approach and to catch implementation issues before transferring the solution to the industry partner.

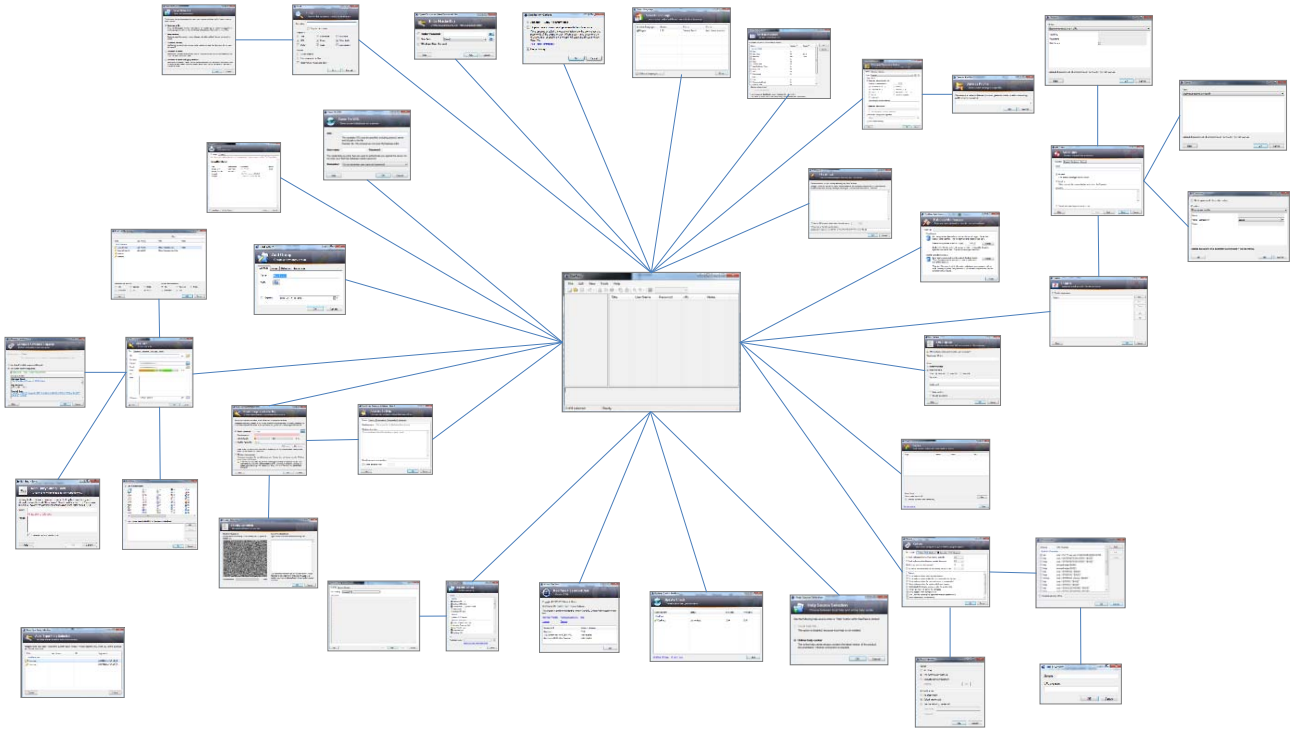

Fig. 2. Overview of the windows and navigation paths of the KeePass open source application.

A. System Under Test

KeePass[4] is an open source password manager. It provides a secure store for a user's passwords in one encrypted database locked with a master key. The application is implemented in C# and uses the Windows Forms (WinForms) GUI library part of Microsoft's .net framework. The GUI shows the typical layout of Windows desktop applications and contains a rich variety of GUI elements (e.g., menu bars, tabs, text areas, lists, context menus), which makes the application an interesting target for applying the monkey framework.

Fig. 2 shows the main window (central node) and the various other windows of the *KeePass* application. The arrows indicate navigation paths that connect the different windows. In a manual exploration a total of 39 different windows have been identified. 22 of these windows are directly accessible from the main window, 15 can only be reached indirectly via one or two other windows. Furthermore, 1 window (lower left in Fig. 2) is not linked with the others via a navigation path; if enabled, it can be opened in any context by pressing a globally available hotkey.

B. Application Scenario

The main window is the anchor point for the application's functionality. It contains 137 GUI elements accessible by the monkey of which 81 are menu items. However, their actual accessibility and, thus, the accessibility of the different linked windows are highly dependent on the application state.

KeePass shows an empty main window after the application is started. If no database has been loaded, only a few functions are enabled in this state. Access to most of the remaining functions requires an open database. For creating a new database a master password has to be specified. The same password has to be entered twice for safety reasons. It is nearly impossible for the monkey to pass this step by generating random input. Therefore a guide has been implemented for creating a new database. Additional guides were implemented for functions that use file dialogs (e.g., open database).

The monkey was configured to make 300 interactions per test run. After a few test runs it turned out that the monkey devoted too much attention to simple interactions (e.g., opening menus, scrolling, resizing) and a new database had been created in only about half of the runs. Hence, many functions that rely an open database have not been invoked by the monkey. Therefore the creation of a new database was included in the setup procedure, so an empty database was available to the monkey directly after the start of the application.

C. Results and Observations

A total of 20 test runs have been performed with *KeePass* compiled in debug mode and the monkey configured with a custom setup, custom guides and a limit of 300 interactions per run. All test runs together resulted in a total of 5,979 interactions. (One run stopped before reaching the interaction limit due to a fatal failure.) A single test run took about one hour (desktop PC with Intel Core i7-3770 CPU 3.4 GHz, 8 GB RAM). The most time-consuming step has been the GUI ripping, especially for complex windows containing a large num-

[4] http://keepass.info

ber of nested elements. However, time consumption was not considered a critical factor for scalability since the monkey tests were intended to run overnight as part of the nightly build.

The achieved "coverage" has been determined in terms of the number of windows the monkey was able to access. For each accessed window we also recorded the number of interactions. Fig. 3 shows the coverage after 20 test runs. The number of interactions per window is denoted by different shades of gray. The black box in the center represents the main window, which has been hit by 4,426 interactions (74% of all interactions). About one third of the remaining windows (13 out of 38) were covered as well; 5 of them received more than 200 interactions each (dark gray boxes) and the other 8 (light gray boxes) were only sporadically covered with an intensity of 1 to 24 interactions each over all 20 test runs.

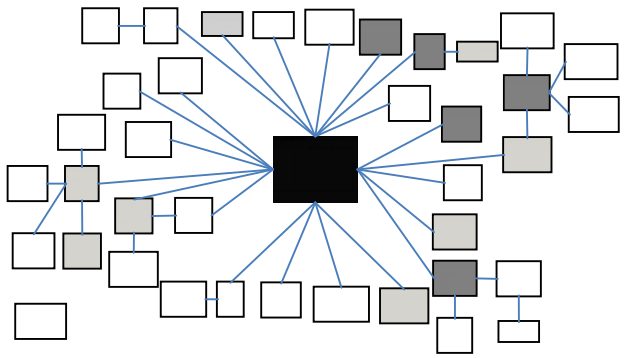

Fig. 3. Window coverage for KeePass after 20 test runs.

The coverage results clearly indicate that the random exploration strategy devotes most interactions to the immediately accessible GUI elements (located on the main window) and, thus, misses many "distant" windows. Adapting the monkey via additional guides, strategies and custom setup procedures is inevitable in order to achieve a least a minimum coverage of the basic functionality. Nevertheless, if the GUI structure is appropriately representing the application's actual usage profile, the monkey covers the most often used parts first and with higher intensity (e.g., the main window). From this perspective, complete coverage of the entire *KeePass* application was not defined as goal for testing and no investments were made in implementing further guides and strategies.

Finding defects was not considered as goal either, since *KeePass* is a very mature and widely-used application (active development since 6 years, 6.5 million downloads in 2015, ports and project forks for various different platforms). We did not expect to see any crashes or failures. To our surprise, the monkey was able to provoke an assertion exception that triggered a fatal failure. It has been caused by a problem in the GUI code, when calculating the position of a window splitter once the monkey had moved the splitter. This example demonstrates that GUI code is also a potential source of critical failures, especially in complex and highly-interactive GUI-based applications.

IV. Industry Application

The next stage after evaluating and improving the monkey framework in an experimental setting was its transfer to the context of the industry partner. The goal was to include the monkey in the nightly build of an ongoing software development project for random GUI testing. The transfer was conducted in two steps. First, the monkey was tailored according to the technical requirements imposed by industry partner's software system and a retrospective analysis using previous versions of the SUT was performed. Second, the monkey was integrated in the nightly build process at the site of the industry partner to provide feedback on the stability of the tested system on a day-to-day basis.

A. System Under Test and Project Context

The monkey was applied for testing the human machine interface (HMI) of the machine control software for high-tech press breaks. These machines are designed for complex bending processes and are characterized by a rich, versatile feature set as well as highest performance and reliability. The machine control software and its HMI have to reflect these quality properties. The software system runs on industrial PC hardware that is equipped with a keyboard, mouse and a touch screen supporting multi-touch gestures. The HMI is implemented with C# and Microsoft's .net framework plus Windows Presentation Foundation (WPF) for realizing the highly interactive GUI.

The machine control software has been under active development and it is still an ongoing project. Frequently, new versions are developed that introduce new features and changes that affect all parts of the system, e.g., the functionality, GUI, feature interactions, as well as base libraries and underlying technologies. Conventional GUI test automation does not work very well for a permanently changing system. Basically for every change of a feature covered by automated regression tests, the test have to be changed as well. Thus, even small crosscutting changes affected many automated GUI tests and blocked their execution due to the necessary rework. As a result, acceptance testing received unstable software versions that failed on simple interactions and frustrated manual testers.

Two main objectives were derived for the application of automated monkey testing of the HMI software. First, the monkey should be resilient enough not to be affected by the frequent changes of the SUT and should not cause additional maintenance effort. Second, the monkey should provide information about the stability and reliability of the SUT for deciding which versions are suitable for manual acceptance testing. The quality gate should open only if the SUT "survived" a predefined number of GUI interactions by the test monkey.

B. Customization and Retrospective Analysis

Customization of the monkey required finding the balance between optimally tailoring it to the technical and functional characteristics of the SUT on the one hand, and keeping the monkey as generic as possible on the other hand, since the entire software system had been under active development and all aspects of the GUI were subject to frequent changes.

Following adaptions were necessary to enable interacting with the SUT.

- *Strategy for identifying GUI elements:* The HMI was based on a GUI library developed in-house, which provided custom GUI elements optimized for touch screens and a generic navigation concept. Custom strategies were required to identify the custom GUI elements relevant for performing interactions. The implementation of these strategies required 44 net lines of code (without class or method declarations).
- *Login guide:* Users have to login to access the machine control software. A custom guide was implemented that provides valid credentials and directions for passing the login procedure (25 net lines of code).
- *File dialog guides:* The HMI allows saving and loading of configuration files. Custom guides (3 net lines of code) were implemented for saving (e.g., a file with a random name is created) and for loading (e.g., a file is randomly selected from a list of available files).
- *Failure detection support:* The HMI uses a custom exception handling strategy including a proprietary exception dialog. In order to recognize application specific exceptions dialogs, the monkey's error detection approach had to be extended with a custom implementation (45 net lines of code for three strategies).

The customized monkey has been evaluated by a retrospective analysis on previous versions of the HMI. To create a representative setting that covers various types of changes, the monkey was run on the successful weekly builds from over one year of development. Fig. 4 shows the results from the 52 runs. Each bar represents one test run comprising 300 GUI interactions. Blue bars indicate successful runs (i.e., the interaction limit was reached without detecting a fatal failure), red or missing bars indicate failed runs. The length of the bars shows the number of interactions performed in the particular run.

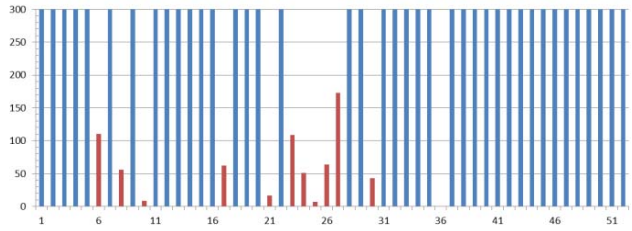

Fig. 4. Monkey test results from retrospective analysis of 52 versions of the SUT (blue bars indicate sucess, red/missing bars show failed runs)

Weekly builds need to have a basic level of stability and robustness, as they are deployed to test machines. It seems surprising that 12 out 52 test runs resulted in a fatal failure, which is almost one quarter (23%). However, according to the developers, most failures (e.g., the frequently encountered *ArgumentOutOfRangeException*) were caused by inconsistent or missing input validation. This problem was resolved in the later versions, which can be observed by the sequence of consistently successful runs towards the end of the analysis.

C. Nightly Build Integration

For applying the monkey as part of the industry partner's development and test process, it was integrated into the nightly build of the HMI project. After the successful build of the HMI software, the binaries were deployed to a test system and the monkey was started in the configuration described above.

The results collected from a one month pilot phase (34 nightly test runs) are shown in Fig. 5. The blue bars show the successfully runs accomplished when the SUT "survived" 300 GUI interactions performed by the monkey. Only 14 of the 34 test runs (41%) were completed successfully. An analysis of the recurring failures revealed that they were usually caused by the same defects. As these defects were often not fixed immediately, they made the test runs to fail several times in a row. The resolution of defects was deferred when, for example, the defect was located in a library developed by a different team or when it did not affected the functionality of the next release.

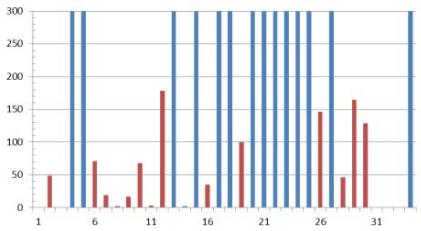

Fig. 5. Monkey test results from nightly builds over one month (blue bars indicate sucess, red/missing bars show failed runs)

D. Results and Observations

Tailoring and extending the monkey required some effort. The most time-consuming part was the development of new strategies for custom GUI components. Even though the employed commercial GUI testing tool provides excellent support for accessing all kind of GUI technologies, writing custom code for proprietary GUI components is still an indispensable and often underestimated task. Such testability issues impact monkey testing in the same was any other GUI test automation.

The monkey test runs were simple "smoke tests" of basic HMI features. However, it turned out that such tests still have their merits as they were able to reveal new defects – including critical ones – that were missed by other testing techniques.

The downside is that the discovered defects were often not fixed immediately, even when they were rated as critical. Thus, although developers already knew a defect, the monkey did not and kept finding the same defect again and again. Furthermore, in case of easily detectable defects, the test runs failed after a few interactions. If not fixed immediately, such "obvious" defects masked other defects in more remote locations.

V. Related Work

Numerous tools for automated GUI testing have been described in the research literature [1]. Among those related to the monkey framework presented in this paper, the most prominent example is GUITAR [11], a model-based test automation framework for GUI applications. Its innovation lies in the

architecture, which uses plug-ins to support flexibility and extensibility. Designed as a framework it has been the basis for the development of customized tools for testing[5].

A few related approaches to the implementation of our custom "guides" and "strategies" for monkey testing have also been proposed in the literature. *Moreira et al.* introduced a pattern-based approach for GUI testing ("UI Test Pattern") [12]. Their approach is based on the assumption that "GUIs that are similar in design, i.e., based on the same UI pattern, should share a common testing strategy". They came up with several generic GUI test patterns such as "Input, Login, Master/Detail, Find, Sort and Call". The aim of these patterns is to create commonly applicable solutions for testing GUIs, whereas the purpose of implementing strategies suggested in our work is to adjust the generic random exploration to the specifics of a concrete SUT. *Bauersfeld and Vos* presented the tool TESTAR that implements a model-based approach to automate testing GUI applications [13]. They also recognized the need to customize their tool by restricting the search space to "interesting" actions that are likely to trigger faults. They used a Prolog engine for specifying relevant custom actions. However, the described customization does not distinguish between the requirements to restrict the random exploration to a predefined interaction sequence in one case ("guides") and to make the exploration more versatile in other cases ("strategies").

VI. Summary and Future Work

In this paper we presented a framework for random (monkey) GUI testing that supports the development of custom-built test monkeys tailored to the requirements of testing a concrete SUT. Based on the framework, custom test monkeys have been built and evaluated on an open source GUI application (*KeePass*) and in context of testing the nightly builds of an industrial application (HMI of a machine control software). The application scenarios demonstrated the benefit of the reusable components and the predefined, generic workflow provided by the framework. In both scenarios it has been inevitable to adapt and extend the monkey to match the peculiarities of the corresponding SUT in order to become effective. In the industry application the monkey would not even be able to interact with the SUT without an appropriate adaptation. For both applications the most important adaptations had to be made due to technical issues introduced by the SUT. With the adaptions in place, the monkey was able to automatically explore the main parts of the SUT's GUI and – in both scenarios – it was able to reveal new faults.

The idea of monkey GUI testing has been successfully transferred into a valuable aid for the industry partner and is used to complement the test automation strategy, e.g., it has been integrated as part of the automated build process. The observed benefits of the monkey testing approach were (1) its ability to automatically create arbitrary interactions sequences that "cover" the main parts of the HMI, (2) its ability to reveal new defects due to unusual interaction sequences and by forcing the application in exceptional states, (3) its contribution to quick estimates of the SUT's reliability for consecutive manual testing by providing information about the stability and robustness, and (4) its resilience to a frequently and rigorously changing GUI of a system under active development.

The application of the test monkey in an ongoing real-world project as part of the nightly build revealed several new requirements, which we plan to address in future work. Monkey testing has advantages if the GUI is frequently changed and refactored. However, its usefulness suffers if the GUI contains areas "under development" that contain "known defects" which are not immediately resolved. Thus, without knowledge about these existing defects, the monkey is likely running into the same failure over and over again creating – from the viewpoint of the developers – "false alarms". Furthermore, if these defects are easily detectable by the monkey, they will dominate the results of the test runs and mask other defects that are harder to find. We therefore consider including information from previous test runs when exploring the SUT so the monkey can avoid paths that are likely to trigger already know defects.

[5] http://guitar.sourceforge.net

Acknowledgment

The research reported in this paper has been supported by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry of Science, Research and Economy, and the Province of Upper Austria in the frame of the COMET center SCCH.

References

[1] I. Banerjee, B. Nguyen, V. Garousi, and A. Memon, "Graphical user interface (GUI) testing: Systematic mapping and repository," Information and Software Technology, 55(10), 1679-1694, Oct. 2013.

[2] T. Y. Chen, F. C. Kuo, R. G. Merkel, and T. H. Tse, "Adaptive random testing: The art of test case diversity," Journal of Systems and Software, 83(1), 60-66, January 2010.

[3] N. Nyman, "Using monkey test tools," Software Testing and Quality Engineering, 29(2), 18-23, January 2000.

[4] J. Fodeh, "Adventures with test monkeys," in Experiences of test automation: case studies of software test automation, D. Graham and M. Fewster, Eds. Addison-Wesley Professional, 2012.

[5] B. Hofer, B. Peischl, and F. Wotawa, "GUI savvy end-to-end testing with smart monkeys," Automation of Software Test (AST), ICSE 2009.

[6] J. E. Forrester, and B. P. Miller, "An empirical study of the robustness of Windows NT applications using random testing," 4th USENIX Windows Systems Symposium, Vol. 4, WSS 2000.

[7] R. Ramler, and W. Putschogl, "A retrospection on building a custom tool for automated system testing," 37th Annual Computer Software and Applications Conference, COMPSAC 2013.

[8] T. Wetzlmaier, and M.. Winterer, "Test automation for multi-touch user interfaces of industrial applications," 8th Int Conference on Software Testing, Verification and Validation Workshops, TAIC PART 2015.

[9] A. Memon, I. Banerjee, A. Nagarajan, "GUI ripping: reverse engineering of graphical user interfaces for testing," 10th Working Conference on Reverse Engineering, WCRE 2003.

[10] J. A. Whittaker, "How to break software," Addison-Wesley, 2003.

[11] B. N. Nguyen, B. Robbins, I. Banerjee, A. Memon, "GUITAR: an innovative tool for automated testing of GUI-driven software," Automated Software Engineering, 21(1), 65-105, March 2014.

[12] R. Moreira, A. Paiva, A. Memon, "A pattern-based approach for GUI modeling and testing," 24th International Symposium on Software Reliability Engineering, ISSRE 2013.

[13] S. Bauersfeld, T.E.J. Vos, "User interface level testing with TESTAR; what about more sophisticated action specification and selection?," 7th Seminar Series on Advanced Techniques & Tools for Software Evolution (SATToSE), CEUR 2014.