GUI Savvy End-to-End Testing with Smart Monkeys

Birgit Hofer and Bernhard Peischl and Franz Wotawa*
Technische Universität Graz
Institute for Software Technology
Inffeldgasse 16b/2, A-8010 Graz, Austria
{hofer,peischl,wotawa}@ist.tugraz.at

Abstract

In this article we report on the development of a graphical user interface-savvy test monkey and its successful application to the Windows calculator. Our novel test monkey allows for a pragmatic approach in providing an abstract model of the GUI relevant behavior of the application under test and relies on a readily available GUI automation tool. Besides of outlining the employed test oracles we explain our novel decision-based state machine model, the associated language and the random test algorithm. Moreover we outline the pragmatic model creation concept and report on its concrete application in an end-to-end test setting with a Windows Vista front-end. Notably in this specific scenario, our novel monkey was able to identify a misbehavior in a well-established application and provided valuable insight for reproducing the detected fault.

1. Introduction

Most test automation approaches focus on systematic testing - repeating the same sequence of actions on the software under test to reveal unexpected behaviour. Despite many advantages, this traditional approach has a number of limitations and still misses serious customer-relevant defects in the software [11]. Moreover, today's test automation often access the application under test (AUT) via an application programming interface. As a result of that the code implementing the graphical user interface is rarely tested. However, to ensure that the application under test meets the customer's quality attributes an end-to-end test is highly desirable.

Applying test monkeys is a powerful supplement that can assist filling systematic (regression) test gaps and detecting the defects. Monkey testing refers to the process of randomly exercising a software program by means of an automated test tool. Unlike regression testing, test monkeys explore the software in a new way each time the test is run, consequently finding new defects. Notably test monkeys can be used early in the development process with low development and maintenance effort [7]. The authors of [11] report on successful applications and discuss the concrete application of dumb and smart monkeys as an orthogonal (and thus supplementing) techique to systematic testing.

The availability of tools for automated treatment of graphical user interfaces allows for recording user interface actions and to replay these actions afterwards. Together with the ability to capture user interface elements and events dynamically these tools provide a solid technical grounding for setting up end-to-end regression tests that help to save costs and allow for improved testing.

More recently, specifications of user interfaces have been used in order to generate test sequences and test data automatically. These test cases can be automatically applied using the replay (to provide the test input) and dynamic capture functionality (to implement the test oracle) of today's GUI automation tools.

However, almost all testing tools as well as the majority of systematic testing approaches rely on an implicit or explicit model. For example, the IDATG tool [3] allows for integrating GUI automation and model-based testing. However, even under presence of these sophisticated tools faults that occur because of unexpected interactions might go undetected.

Consider for example the Windows calculator application. Figure 1 shows the graphical user interface of the calculator in scientific mode of the German version of Windows Vista. The calculator application is not a very complicated application and should be right. Unfortunately, this is not the case. For example entering a number, e.g., `57` and divide it by `0`, leads to a correct response that a division by zero is not allowed. However, after calling the help application, the display of the result turns to `100`, which should not happen. Although it is impossible to use this number afterwards directly for computation, such a behavior appears to

*Authors are listed in alphabetical order. The research herein is partially conducted within the competence network Softnet Austria (www.soft-net.at) and funded by the Austrian Federal Ministry of Economics (bm:wa), the province of Styria, the Steirische Wirtschaftsförderungsgesellschaft mbH. (SFG), and the city of Vienna in terms of the center for innovation and technology (ZIT).

978-1-4244-3711-5/09/$25.00 © 2009 IEEE

Figure 1. The interface of the Windows calculator in scientific mode

be unintended and thus should be avoided. Hence, there is still an unwanted behavior. Moreover, such a fault can even be hardly found in manual testing.

What helps to detect such faults is random testing. And indeed the mentioned Windows calculator bug was found using our monkey testing tool we are going to present in this paper. What makes this bug specifically interesting is the fact that it can only be found when testing the user interface. Moreover, user interfaces account for about 45-60 % of the whole source code [9] but are mostly tested manually. Hence, automated user interface testing has a huge practical impact.

But what makes the automation of user interface testing so complicated? As reported in [14] there are several major factors including the large number of possible user interactions, the determination of coverage, the unexpected behavior in case of failures, which might prevent from further interactions with the application, and problems regarding the matching of user interface elements like text fields with their representations in the test case.

The latter issue is very important because regression testing requires a stable mapping but changes in the user interface, which often occur, might change this mapping. Because of the mentioned challenges simple Capture-Replay tools often fail to meet their expectations in practice. One solution of the mapping problem is to make this explicit and couple Capture-Replay tools with a model of the user interface from which the test cases can be derived directly. IDATG [3] is one example of such a tool.

Our work follows the idea of using models in combination with GUI automation tools. Instead of deterministically generated test sequences we rely on random sequences of interactions. Random or stochastic testing has been developed because of the fact that exhaustive testing is rarely possible for real-world applications (see [12]). Monkey testing can be seen as an instance of random testing. In [7] monkey testing is defined as the process of randomly executing a program. It is important to note that each test run is completely non-deterministic and therefore each time a test is invoked the software is explored in a new way. This is the reason why test monkeys often detect bugs that are not found when using deterministic test approaches.

In literature two types of test monkeys are distinguished: dumb monkeys and smart monkeys (see [1]). Dumb monkeys have no glue about the application. They are able to detect faults related to application crashes, memory leaks, or access violations and are inexpensive to build. Smart monkeys know the application to some extent. They can differentiate between valid inputs and outputs. They know about consequences of interaction and know how to access the applications user interface. Smart monkeys require a (not necessary complete) model of the application but therefore they are more expensive to construct and maintain.

The main advantage of smart monkeys is that they may find costumer-relevant bugs. Because of this we focus in our work on smart monkeys. In particular it is important to come up with a modeling language that is expressive enough but easy to handle in order to keep modeling costs as low as possible. In this paper, we introduce a modeling language that is an extension of finite state machines but avoids some problems like state explosion. The language is very well adapted to the domain of user interface testing. We further demonstrate the capabilities of the language using the Windows Calculator application and show the results obtained using our prototype implementation.

2. Modeling the Monkey's Behavior

Smart monkeys require a model from which they derive the events and inputs to be sent to the AUT. It has been argued [11] that smart monkeys are expensive to develop and that the cost-benefit ratio might be rather low. In general, this argumentation is traceable as modeling always demands for resources. However, the costs typically depend on the expertise of the engineers, the modeling paradigm employed and the language being used. In using an appropriate modeling language and additional tool support that is close to the application domain, the additional costs might be quite reasonable, thus leading to an acceptable cost-benefit ratio.

Moreover - compared with dumb monkeys - smart monkeys have stronger fault detection capabilities. While dumb monkeys can only be used to detect faults that cause an application to crash or deadlock, smart monkeys are able to find bugs by using the output and the user-interface behavior of an application. A comparison of smart monkeys with other testing approaches leads to similar observations. Hence, smart monkeys complement other testing approaches and developing a modeling language that is close to the specific application domain is desired in order to keep additional costs for modeling as low as possible.

The modeling language we introduce in this section is

well adapted to system testing where the application communicates with a user via a graphical user interface. The language allows specifying possible sequences of interactions where the real sequence is selected randomly by the smart monkey. Moreover, the language can be used to specify an oracle that answers the question whether an application behaves correctly after one interaction with the monkey. The literature discusses lots of different modeling paradigms varying from finite state machines [6] to decision trees [2, 5] or event flow graphs [10] and grammars [13]. Our modeling language is based on a finite state machine extended with elements from UML and state charts. The goal was to develop a language that is as close to graphical user interface testing as possible. The language we used for modeling is based on a decision based state machine (DBSM).

Definition 1 (DBSM) *A decision based state machine (DBSM) over a language* $\mathbf{L}$ *is a tuple* (S, s_0, ρ, T, E, V) *where:*

- $S = \Sigma \cup D$ *is a set of states comprising standard states* Σ *and decision states* D.
- $s_0 \in \Sigma$ *is the initial state.*
- $\rho : \Sigma \mapsto \mathbf{L}$ *is a function mapping standard states to properties, which are formulated in* $\mathbf{L}$.
- $T = T_E \cup T_C$ *is the set of transitions comprising event transitions* $(s, e, c, s') \in T_E$ *and conditional transitions* $(s_C, co, s') \in T_C$ *where* $s \in \Sigma$, $s' \in S$, $s_C \in D$, $e \in E$, $c \in \mathbf{L}$ *is a program possible changing variables, and* $co \in \mathbf{L}$ *is a boolean condition.*
- E *is a set of events.*
- V *is a set of variables.*

In the definition all programs that correspond to a transition are expected to use only variables from V. We do not formally define the semantics of the language $\mathbf{L}$. Any language under given restrictions, which we define in this article whenever necessary, is sufficient. Furthermore, we assume that the semantics of $\mathbf{L}$ is defined over variable environments. A variable environment is a function env mapping variables (from V) to their values. The semantics function $eval$ maps a program $\Pi \in \mathbf{L}$ and an input environment to a new environment accordingly to the semantics of the language. Furthermore, we assume that for every expression $e \in \mathbf{L}$ there is an evaluation function $eval_E$ mapping expressions and variable environments to values.

In the context of monkey testing the language $\mathbf{L}$ is assumed to comprise all necessary predicates that allow for testing the current state of the graphical user interface elements of the AUT. This includes asking for specific values held by text fields or testing the liveness of the application. We come later to this issue again. But before that we discuss the semantics of DBSM.

In our case we are interested in the states that can be reached from the starting state and the corresponding events. The events in our application domain correspond to actions performed at the side of the AUT. This includes inserting a value to a text field or pressing a button.

The following definition describes the semantics of a DBSM in terms of traces where a trace is a sequence of events. We assume a concatenation function $\oplus$ for sequences, and that ϵ represents the empty sequence.

Definition 2 (Trace) *A trace is a sequence of events that can be reached from the starting state and an empty environment using either event or conditional transitions. Conditional transitions can only be taken if the condition evaluates to* $TRUE$ *under the given variable environment.*

The following algorithm **trace** takes a state $s \in S$ and a variable environment env as inputs, and returns a trace.

Algorithm trace (s,env)

1. If s is a *standard state*, i.e., $s \in \Sigma$, then do the following:
   (a) If the properties of the state cannot be fulfilled, i.e, $eval(\rho(s), env) = FALSE$, then return $\langle$ **FAIL** $\rangle$.
   (b) If there are event transitions from state s, then randomly select one transition $(s, e, c, s') \in T$. Call **trace**$(s', eval(c, env))$ and store the result in t'. Return the trace $\langle e \rangle \oplus t'$ as result.
2. If s is a *decision states*, i.e., $s \in D$, and there exists a conditional transition $(s, co, s') \in T$, which evaluates to true $eval(co, env) = TRUE$, then call **trace**(s', env) and return the result.
3. In cases where no other rule apply return the empty trace ϵ.

The algorithm **trace** is non deterministic and might return different traces whenever called. Moreover, there is the underlying assumption that for decision states there is only one conditional transition where the corresponding condition evaluates to $TRUE$ under a given variable environment. Note that for monkey testing there is another dimension of randomness. The used values of test fields should also be selected randomly. We assume that randomness can be expressed in the language $\mathbf{L}$. Hence, when setting a value of a text field we use $\mathbf{L}$.

In order to use DBSM as modeling paradigm for smart monkeys we have to ensure that $\mathbf{L}$ comprises certain functions and predicates used for communication with the AUT. To conclude this section we therefore introduce the most important functions and predicates. Note that we also assume that every element of the graphical user interface that

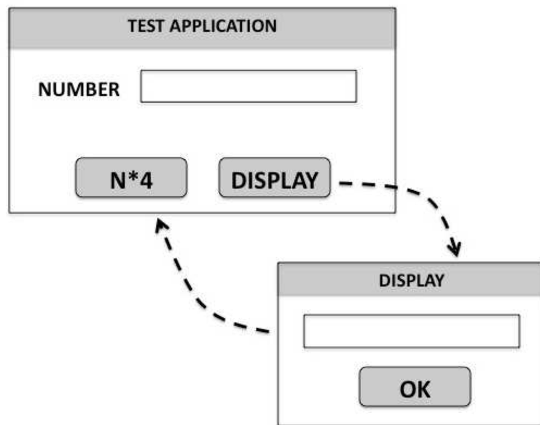


Figure 2. A simple graphical user interface to be used as example

is going to be used during testing has an unique identifier which can be accessed by the test monkey. We start with a discussion of predicates used as properties assigned to a state. Those predicates are used to implement an oracle that allows to distinguish incorrect from correct behavior.

NotIsHungProperty This predicate returns $TRUE$ if the AUT is still alive, and $FALSE$, otherwise.

WindowCaptionProperty This predicates takes an windows identifier as argument and returns $TRUE$ if the active window is the given one, and $FALSE$, otherwise.

AllClickEventsAvailableProperty This predicate checks whether all clickable events are available in the current state. The clickable events are those events specified for all leaving transitions.

TextfieldValueProperty This predicates takes the text field identifier and the expected value as inputs. If the current value of the specified text field and the expected value are equivalent, $TRUE$ is returned. Otherwise, $FALSE$ is given back as result.

In addition, we assume that **L** comprises the usual predicates, operators, and statements like a typical assignment language include functions **RANDOM_N** and **RANDOM_S** returning random values for numbers and strings respectively. These functions are used to randomly set the value of a text field in an user interface. Moreover, we also assume functions in **L** to access values of the user interface. In particular, **TextfieldValue** and **CheckboxValue** return the value of a specified text field or checkbox respectively.

The events used to communicate with the AUT comprises the following ones:

StartApplication This event is used to start the application to be tested.

CloseActiveWindow This event is for closing the active window via clicking the corresponding button.

GlobalKeyStroke Performs a keyboard input like `CTRL-ALT-DELETE`.

ComboBoxClick Opens the drop down list of a combo box and selects randomly one of the items.

KeyboardInput Fills the given input text field. The type of the value (alpha, numbers, or date) can be further specified.

MouseClick Access controls like radio buttons, check boxes, or push buttons. The corresponding identifier has to be provided as an argument.

MenuClick Access a menu item. The identifier of the menu has to be provide as argument.

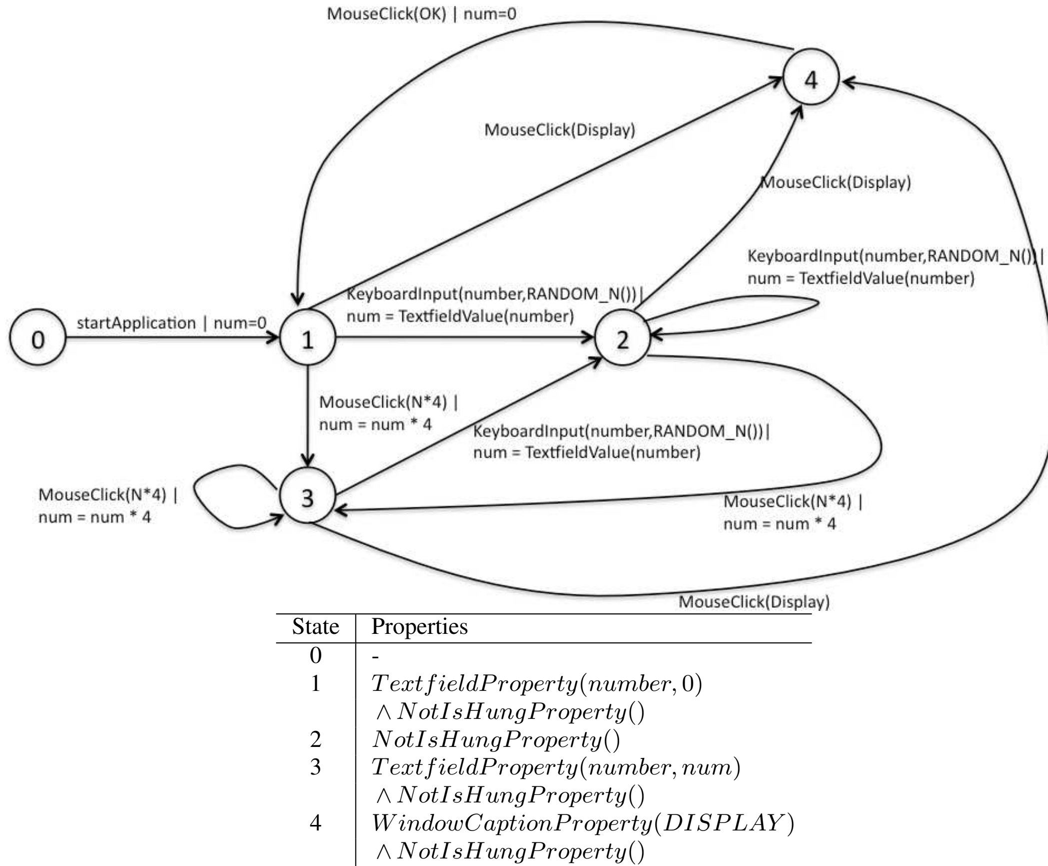
In the next section, we show how to use the DBSM and the given events, functions, and predicates to specify the model for a small example application.

3. Example

Figure 2 shows the interface of a small example application. The expected behavior of the application can be expressed as follows: After starting the application the main window *TEST APPLICATION* appears where a number can be inserted by the user. There are two buttons. The *N*4* button takes the inserted number and multiplies it with 4. The result is stored in the same text field. The *DISPLAY* buttons opens a new window *DISPLAY* where the current number is shown. After pressing the *OK* button the *DISPLAY* window is closed and the text field of the *TEST APPLICATION* window is reset to zero.

Using the DBSM we are able to specify the behavior of the example application formally. Figure 3 shows the graphical representation of the DBSM for the example application. The state 0 is the starting state from which the application is launched. Initially the value of the text field `number` is 0. When pressing the `N*4` button, we are going to state 3. A new random value for the `number` text field leads to state 2. And finally, if pressing `Display` the state 4 is reached. From state 4 we are only able to go back to state 1 after pressing the `OK` button of the `DISPLAY` window. In this case the text field `number` has to have the value 0 again.

In Figure 3 we also show the properties of each state. In any state the application has to be alive. Hence, the $NotIsHungProperty()$ has to be fulfilled. In State 1 the text field of the application should have the value `0`. State 2 represents a state where a new value for the text field is set



State	Properties
0	-
1	$TextfieldProperty(number, 0)$ $\wedge NotIsHungProperty()$
2	$NotIsHungProperty()$
3	$TextfieldProperty(number, num)$ $\wedge NotIsHungProperty()$
4	$WindowCaptionProperty(DISPLAY)$ $\wedge NotIsHungProperty()$

Figure 3. The DBSM of the example application from Fig. 2

randomly. Beside the $NotIsHungProperty()$ there are no other properties specified for this state. State 3 represents the application after clicking on the `N*4` button. In this state the text field value has to be the new one. Finally, in State 4 the `DISPLAY` window should be open and active to present the result. Other properties are possible like one to check the value displayed at the `DISPLAY` window and others. Note that - according to our assumptions in Section 2 - our language L contains all necessary predicates (like for example the predicate $NotIsHungProperty$) and thus we can compute the evaluation function $eval_E$ for every expression $e \in L$.

In order to show the general applicability of DBSM for a modeling monkey tester, we further demonstrate the corresponding model of a dumb monkey. Note that a dumb monkey in the original definition does not use a model of the system. Instead all controls and text fields are accessed randomly during testing simulating a tester with no knowledge about the AUT at all. When using DBSM we simulate this random behavior. Figure 4 shows the model. Obviously this model does restrict the interactions with the applications. However, it is easy to add new transitions, which allow for other user interactions like pressing keys on the keyboard and others. In Figure 4 the properties of the states are not shown. The starting state 0 has no properties. State 1 has the $NotIsHungProperty()$ property only. Note that the dumb monkey will take events that are not applicable at a certain time during running the application. For example, the `DISPLAY` window might not be open when the dumb monkey selects the $MouseClick(OK)$ event. Hence, we assume during execution that interface events will be ignored in cases where there is no corresponding interface element available.

In the next section we discuss the testing process from model creation to running the monkey when using a smart monkey testing tool.

4. The Monkey's Testing Process

In this section we discuss the process necessary to apply a smart monkey in practice. As motivated in the introduction a smart test monkey is required to recognize user interface elements dynamically as the dynamic behavior of the AUT is unforeseeable and a smart monkey has to be able to deal with this dynamics. Note that - to be able to deal with deadlocks and crashes - the test monkey as well as the AUT

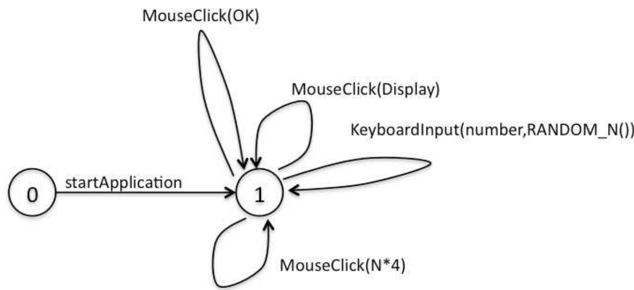


Figure 4. The DBSM of a dumb monkey for the application given in Fig. 2

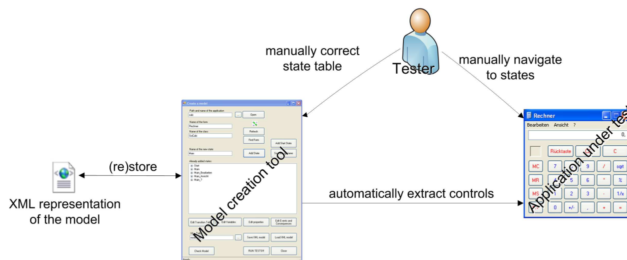


Figure 5. The model creation process

belong to different process spaces. Thus the GUI automation tool has to be able to dynamically identify the various user interface elements alongside with their actual state (e.g. whether an element can get the focus or is clickable) outside of the own process scope. In our case the Ranorex Automation Framework (http://www.ranorex.com) provided a solid technical underpinning for setting up our smart monkey with a reasonable effort.

Having a tool for accessing a user interface from outside, we are able to implement a smart monkey including a model creation, a model execution, and a result evaluation component. These three components are integral part of the overall test process when using a smart monkey.

Model creation is necessary for smart monkeys to provide enough knowledge in order to guide random testing to those parts of the application that are of customer relevance. From the perspective of tool support model creation is a combination of exploration and manual refinement.

In particular a typical Capture-Replay tool allows for extracting the elements of the application's GUI. This information can be used to create new states and transitions as well as properties. For example, transitions can be generated for every possible GUI element. The information regarding the GUI element can be directly used to create the events assigned to transitions. The refinement starts afterwards. New states can be generated and transitions can be adapted. The model creation process is depicted in Figure 5.

After creating the model the smart test monkey is executed. The execution starts with the start state of the model and ends whenever a property is violated. During the execution the test monkey writes the trace to a file. This file, i.e., the log file, is used afterwards to give information regarding the nature of the detected fault. It is important that the test monkey is designed such that the event of the selected transition is recorded in the log file before sending the event to the application. Otherwise, it might be not possible to record the event in case of a system crash caused by the event. There are other issue regarding execution in practice. For example, in some cases a user should be imitated as close as possible, which requires a delay between consecutive events to be send to the application.

The final step of monkey testing requires the analysis of the log file. This comprises an analysis of the reported fault such that the question whether the fault is caused by the application or the used model can be answered. In the latter case the model has to be adapted and the test has to be run again. If the application can be blamed, the log file should be minimized in order to extract the events necessary for reconstructing the faulty behavior. The extraction part is not automated at the moment. However, approaches like Delta Debugging [15] can be used for this purpose.

5. The Windows Calculator Case Study

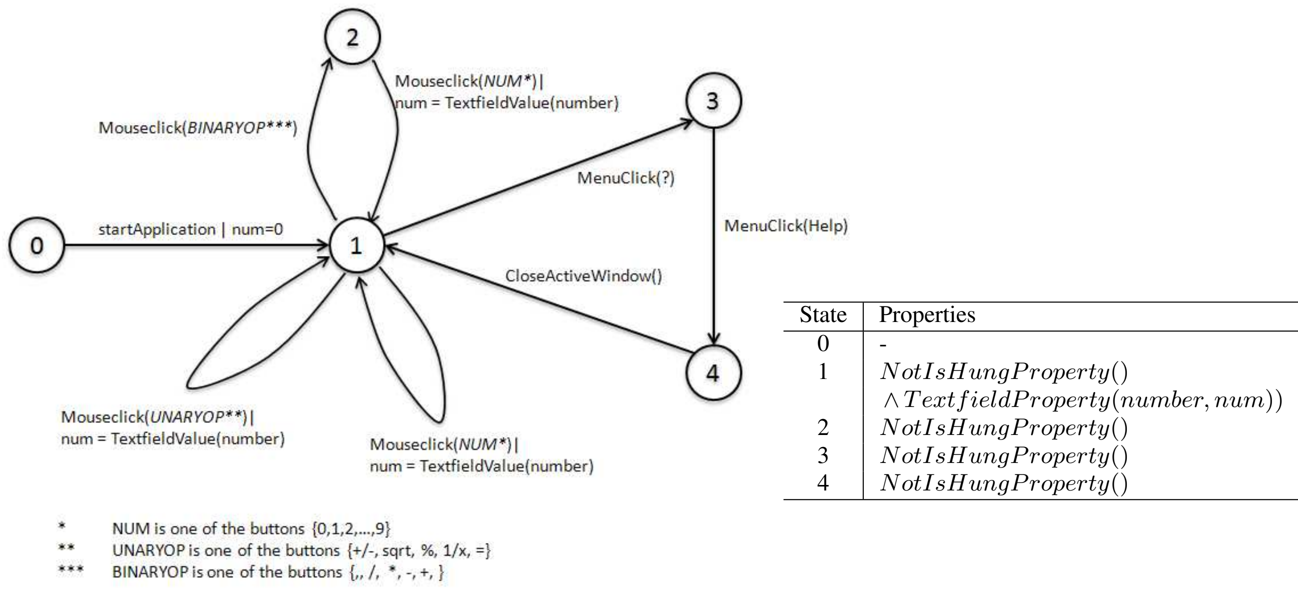
In order to give a proof of concept a prototype was developed in C# (.NET 3.5). The library Ranorex 1.5 (Professional edition)[1] was used in order to access the elements of graphical user interfaces.

We constructed two models of the Windows calculator. One model represents the behavior when the calculator is in the scientific mode and a second model, which does not include the scientific part, i.e., it focuses on the standard view of the calculator only. Another simplification is made by excluding the working memory of the calculator (the buttons `MC`, `MR`, `MS` and `M+`) from the model.

Figure 6 shows a part of the model of the calculator's standard mode. State 0 is the starting state from which the application will be launched. State 1 allows pressing any button or to open the `?`-menu. The calculator allows the user to submit input via the keyboard. These inputs are not included in Figure 6, but they are included in the model of the calculator that was used for the monkey test. In addition the complete model includes the menu items `Edit` (with the sub menu item `Copy` and `Paste`), `View` (with the sub menu items `Standard` and `Grouping`) and the sub menu item `?/Info`.

Figure 6 includes a table with the properties which must be fulfilled in the different states. All states except state 0 require a running application. Thus the *NotIsHungProperty()* has to be fulfilled in these states. State 1 has one addi-

[1] see http://www.ranorex.com/



State	Properties
0	-
1	$NotIsHungProperty()$
	$\wedge TextfieldProperty(number, num))$
2	$NotIsHungProperty()$
3	$NotIsHungProperty()$
4	$NotIsHungProperty()$

Figure 6. The simplified model of the calculator

tional property: The value of the text field *number* must be equal to the value of the variable *num*. This property is the one which is not fulfilled when executing the events of the example from the introduction. The creation of the model of the calculator's standard view using our smart monkey tool requires about half an our.

The model of the scientific part of the calculator is more complex and requires the handling of different states that correspond with the numeric systems of the calculator. Note that in the scientific mode the numbers can be expressed using the hexadecimal, the binary, the octal, or the decimal number system. These different representations can be easily modeled using decision states. Figure 7 shows a simplified model with decision states. Beside the bug mentioned in the introduction of this paper, we were able to detect another misbehavior using the model of the calculator. If the monkey causes an error message like *'Division by 0 not possible'* or *'Undefined output'*, then it is not possible to open the `Info` window from the `'?'` menu.

The case study demonstrated that writing a smart monkey using the developed tool is efficient but requires additional time. However, monkeys detect faults that can hardly be revealed when using more traditional testing techniques. Even in the case of applications like the Windows calculator that are often used there are still bugs that can be detected when interacting with the GUI. Hence, the additional effort for writing a model is a good investment for making applications more reliable.

6. Discussion and Related Work

Following the strategy proposed herein, we build up the model by relying on an existing implementation of the GUI. Afterwards we tailor this model to specific needs, for example, we specify where exactly to check the test oracle. As the requirements often reason about the user interface, one could probably build up a model by focusing on the requirements - however, it is an unresolved issue how to generate DBSM models out of requirements documents.

To our best knowledge, smart monkey testing has only been a minor topic in test automation research. The authors of [11] report on conflicting opinions about the efficacy of monkey test tools. Boris Beizer suggests in Black Box Testing [4] that test monkeys are not very useful for testing todays professionally created software. His analysis concludes that the use of good testing practices will find more bugs than dumb monkey testing. This article also mentions that James Tierney, former director of testing at Microsoft, has reported in internal presentations that some Microsoft applications groups have found ten to twenty percent of the bugs in their projects using monkey test tools. John Fodeh [7] reports on the specific features of a test monkey employed at B-K Medical alongside with a proposal for a metric that allows for deciding when to enter and stop systematic testing by relying on a monkey test tool.

7. Conclusion

In this article we report on the development of a graphical user interface-savvy test monkey and its successful application to the Windows calculator. Our novel test monkey

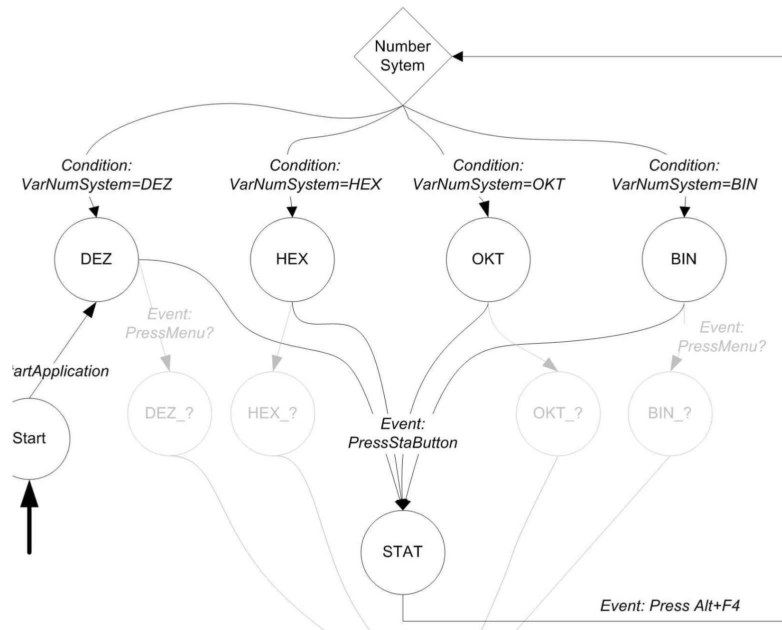


Figure 7. Part of the model of the calculator in scientific mode

allows for a pragmatic approach in providing an abstract model of the GUI relevant behavior of the AUT and relies on readily available GUI automation tools.

Moreover, we outline a novel decision-based state machine model that enriches the well-known finite state machine paradigm with elements from UML and allows for specifying possible interaction sequences and abstract oracles. The introduced algorithm performs a random walk among these specified sequences of events (traces) and thus - in contrast to a classical dumb monkey - our smart monkey solely considers meaningful interaction sequences. Furthermore we report on a case study, an end-to-end test of a well-established Windows Vista mainstream application. Notably in this specific scenario, our novel monkey was able to identify a misbehavior and provided valuable insights for reproducing the detected fault. In a different context - namely the evaluation of robustness of operating systems tools - the literature reports from random testing with randomly choosen input (e.g. in [8]).

References

[1] T. R. Arnold. *Visual Test 6 Bible*. IDG Books Worldwide, Inc., Foster City, CA, USA, 1998.

[2] R. Baker. A decision table based methodology for the analysis of complex conditional actions. *Methods & Tools, Volume 12 - number 3*, Fall 2004.

[3] A. Beer, S. Mohacsi, and C. Stary. IDATG: an open tool for automated testing of interactive software. In *Proceedings of the 22nd International Computer Software and Applications Conference (COMPSAC)*, pages 470–475, August 1998.

[4] B. Beizer. *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[5] A. M. Davis. A comparison of techniques for the specification of external system behavior. *Commun. ACM*, 31(9):1098–1115, 1988.

[6] I. K. El-Far and J. A. Whittaker. Model-based software testing. *Encyclopedia on Software Engineering*, 2001.

[7] J. A. Fodeh. Test monkeys - the new members of your team? Presented at the SIGiST Conference 'The Followship of the Test', February 11th, 2003, British Computer Society, UK.

[8] J. E. Forrester and B. P. Miller. An empirical study of the robustness of windows nt applications using random testing. In *In Proceedings of the 4th USENIX Windows System Symposium*, pages 59–68, 2000.

[9] A. M. Memon. *A comprehensive framework for testing graphical user interfaces*. PhD thesis, University of Pittsburgh, 2001. Adviser-Mary Lou Soffa.

[10] A. M. Memon. An event-flow model of GUI-based applications for testing. *Software Testing, Verification and Reliability*, 17(3):137–157, 2007.

[11] N. Nyman. Using monkey test tools. *Software Testing & Quality Enineering Magazine*, January/February 2000.

[12] J. A. Whittaker. Stochastic software testing. *Ann. Softw. Eng.*, 4:115–131, 1997.

[13] J. A. Whittaker. What is software testing? And why is it so difficult. *IEEE Software*, 17(1):70–79, January/February 2000.

[14] Q. Xie. Developing cost-effective model-based techniques for GUI testing. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 997–1000, New York, NY, USA, 2006. ACM.

[15] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, February 2002.