OULU 2019
C 724

ACTA UNIVERSITATIS OULUENSIS

Pekka Aho

AUTOMATED STATE MODEL EXTRACTION, TESTING AND CHANGE DETECTION THROUGH GRAPHICAL USER INTERFACE

UNIVERSITY OF OULU GRADUATE SCHOOL;
UNIVERSITY OF OULU,
FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING;
OPEN UNIVERSITY OF THE NETHERLANDS

ACTA UNIVERSITATIS OULUENSIS
C Technica 724

PEKKA AHO

AUTOMATED STATE MODEL EXTRACTION, TESTING AND CHANGE DETECTION THROUGH GRAPHICAL USER INTERFACE

Academic dissertation to be presented with the assent of the Doctoral Training Committee of Information Technology and Electrical Engineering of the University of Oulu for public defence in Martti Ahtisaari -sali (L2), Linnanmaa, on 13 November 2019, at 12 noon

UNIVERSITY OF OULU, OULU 2019

Acta Univ. Oul. C 724, 2019

Supervised by
Professor Juha Röning
Professor Tanja E. J. Vos

Reviewed by
Professor Ana C. R. Paiva
Professor Anna Rita Fasolino

Opponent
Professor Serge Demeyer

ISBN 978-952-62-2405-3 (Paperback)
ISBN 978-952-62-2406-0 (PDF)

ISSN 0355-3213 (Printed)
ISSN 1796-2226 (Online)

Cover Design
Raimo Ahonen

JUVENES PRINT
TAMPERE 2019

Aho, Pekka, Automated state model extraction, testing and change detection through graphical user interface.
University of Oulu Graduate School; University of Oulu, Faculty of Information Technology and Electrical Engineering; Open University of the Netherlands
Acta Univ. Oul. C 724, 2019
University of Oulu, P.O. Box 8000, FI-90014 University of Oulu, Finland

Abstract

Testing is an important part of quality assurance, and the use of agile processes, continuous integration and DevOps is increasing the pressure for automating all aspects of testing. Testing through graphical user interfaces (GUIs) is commonly automated by scripts that are captured or manually created with a script editor, automating the execution of test cases. A major challenge with script-based GUI test automation is the manual effort required for maintaining the scripts when the GUI changes. Model-based testing (MBT) is an approach for automating also the design of test cases. Traditionally, models for MBT are designed manually with a modelling tool, and an MBT tool is used for generating abstract test cases from the model. Then, an adapter is implemented to translate the abstract test cases into concrete test cases that can be executed on system under test (SUT). When the GUI changes, the model has to be updated and the test cases can be generated from the updated model, reducing the maintenance effort. However, designing models and implementing adapters requires effort and specialized expertise.

The main research questions of this thesis are 1) how to automatically extract state-based models of software systems with GUI, and 2) how to use the extracted models to automate testing. Our focus is on using dynamic analysis through the GUI during automated exploration of the system, and we concentrate on desktop applications. Our results show that extracting state models through GUI is possible and the models can be used to generate regression test cases, but a more promising approach is to use model comparison on extracted models of consequent system versions to automatically detect changes between the versions.

Keywords: dynamic analysis, model inference, model-based testing, software engineering, test automation

Aho, Pekka, Automaattinen tila-mallin luominen, testaaminen ja muutosten havaitseminen graafisen käyttöliittymän kautta.
Oulun yliopiston tutkijakoulu; Oulun yliopisto, Tieto- ja sähkötekniikan tiedekunta; Open University of the Netherlands
Acta Univ. Oul. C 724, 2019
Oulun yliopisto, PL 8000, 90014 Oulun yliopisto

Tiivistelmä

Testaaminen on tärkeä osa laadun varmistusta. Ketterät kehitysprosessit ja jatkuva integrointi lisäävät tarvetta automatisoida kaikki testauksen osa-alueet. Testaus graafisten käyttöliittymien kautta automatisoidaan yleensä skripteinä, jotka luodaan joko tallentamalla manuaalista testausta tai kirjoittamalla käyttäen skriptieditoria. Tällöin scriptit automatisoivat testitapausten suorittamista. Muutokset graafisessa käyttöliittymässä vaativat scriptien päivittämistä ja scriptien ylläpitoon kuluva työmäärä on iso ongelma. Mallipohjaisessa testauksessa automatisoidaan testien suorittamisen lisäksi myös testitapausten suunnittelu. Perinteisesti mallipohjaisessa testauksessa mallit suunnitellaan manuaalisesti käyttämällä mallinnustyökalua, ja mallista luodaan abstrakteja testitapauksia automaattisesti mallipohjaisen testauksen työkalun avulla. Sen jälkeen implementoidaan adapteri, joka muuttaa abstraktit testitapaukset konkreettisiksi, jotta ne voidaan suorittaa testattavassa järjestelmässä. Kun testattava graafinen käyttöliittymä muuttuu, vain mallia täytyy päivittää ja testitapaukset voidaan luoda automaattisesti uudelleen, vähentäen ylläpitoon käytettävää työmäärää. Mallien suunnittelu ja adapterien implementointi vaatii kuitenkin huomattavan työmäärän ja erikoisosaamista.

Tämä väitöskirja tutkii 1) voidaanko tilamalleja luoda automaattisesti järjestelmistä, joissa on graafinen käyttöliittymä, ja 2) voidaanko automaattisesti luotuja tilamalleja käyttää testauksen automatisointiin. Tutkimus keskittyy työpöytäsovelluksiin ja dynaamisen analyysin käyttämiseen graafisen käyttöliittymän kautta järjestelmän automatisoidun läpikäynnin aikana. Tutkimustulokset osoittavat, että tilamallien automaattinen luominen graafisen käyttöliittymän kautta on mahdollista, ja malleja voidaan käyttää testitapausten generointiin regressiotestauksessa. Lupaavampi lähestymistapa on kuitenkin vertailla malleja, jotka on luotu järjestelmän peräkkäisistä versioista, ja havaita versioiden väliset muutokset automaattisesti.

Asiasanat: dynaaminen analyysi, mallin generointi, mallipohjainen testaus, ohjelmistokehitys, testiautomaatio

Acknowledgments

I would like to thank my first supervisor prof. Juha Röning for the patience during this long process, and my second supervisor prof. Tanja Vos for providing me the opportunity to finish this thesis. I would like to thank the co-authors of my publications for the help and contributions, especially Matias Suarez at F-Secure Ltd for the fruitful collaboration on Murphy tools. I would like to thank my family, friends, and colleagues at VTT Technical Research Centre of Finland, Fraunhofer FOKUS in Germany, University of Maryland in the US, and Open University of the Netherlands for the support. Special thanks to prof. Ina Schieferdecker and prof. Atif Memon for allowing me to be part of your research teams during my research exchange visitations. Also, I would like to thank Business Finland for the funding in D-MINT and ATAC projects and Netherlands Enterprise Agency for the funding in TESTOMAT project that made this research possible.

27 September 2019 Pekka Aho

Abbreviations

AI	artificial intelligence
AIF	application independent functionalities
AJAX	asynchronous JavaScript and XML
API	application programming interface
AUT	application under test
C&R	capture & replay
CI	continuous integration
e.g.	exempli gratia
EFG	event-flow graph
etc.	et cetera
FSM	finite state machine
GUI	graphical user interface
HFSM	hierarchical finite state machines
i.e.	id est
MBGT	model-based GUI testing
MBT	model-based testing
ML	machine learning
OS	operating system
QA	quality assurance
RIA	rich internet application
ROI	return of investment
SUT	system under test
SW	software
UI	user interface
VGT	visual GUI testing
XML	extensible markup language

Original publications

This thesis is based on the following publications, which are referred throughout the text by their Roman numerals:

I Aho, P., Menz, N., Räty, T., & Schieferdecker, I. (2011). Automated Java GUI modeling for model-based testing purposes. *Proceedings of the 2011 Eighth International Conference on Information Technology: New Generations (ITNG)*, 268–273. 11-13 Apr 2011, Las Vegas, USA. doi:10.1109/ITNG.2011.54

II Aho, P., Menz, N., & Räty, T. (2011). Enhancing generated Java GUI models with valid test data. *Proceedings of the 2011 IEEE Conference on Open Systems (ICOS)*, 310–315. 25-28 Sep 2011, Langkawi, Malaysia. doi:10.1109/ICOS.2011.6079253

III Aho, P., Räty, T., & Menz, N. (2013). Dynamic reverse engineering of GUI models for testing. *Proceedings of the 2013 International Conference on Control, Decision and Information Technologies (CoDIT)*, 441–447. 6-8 May 2013, Hammamet, Tunisia. doi:10.1109/CoDIT.2013.6689585

IV Aho, P., Suarez, M., Kanstrén, T., & Memon, A.M. (2013). Industrial adoption of automatically extracted GUI models for testing. *Proceedings of the 3rd International Workshop on Experiences and Empirical Studies in Software Modeling (EESSMod)*, 1078:49–54. 1 Oct 2013, Miami, Florida, USA.

V Aho, P., Suarez, M., Kanstrén, T., & Memon A.M. (2014). Murphy tools: Utilizing extracted GUI models for industrial software testing. *Proceedings of the 2014 IEEE 7th International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 343–348. 2014. doi:10.1109/ICSTW.2014.39

VI Aho, P., Kanstrén T., Räty, T., & Röning, J. (2014). Automated extraction of GUI models for testing. *Advances in Computers, 95:2*, 49–112. Academic Press, Elsevier, 2014.

VII Aho, P., Suarez, M., Memon, A.M., & Kanstrén, T. (2015). Making GUI testing practical: Bridging the gaps. *Proceedings of the 12th International Conference on Information Technology: New Generations (ITNG 2015)*, 439–444, 13-15 Apr 2015, Las Vegas, USA. doi: 10.1109/ITNG.2015.77

VIII Aho, P., Alégroth, E., Oliveira, R., & Vos, T. (2016). Evolution of automated regression testing of software systems through the graphical user interface. *Proceedings of the First International Conference on Advances in Computation, Communications and Services (ACCSE)*, 16–21, 22-26 May 2016, Valencia, Spain.

IX Aho, P. & Vos, T. (2018). Challenges in automated testing through graphical user interface. *Proceedings of the 2018 IEEE International Conference on Software Testing Verification and Validation Workshop, (ICSTW)*, 118–121, 9 Apr 2018, Västerås, Sweden. doi: 10.1109/ICSTW.2018.00038

In all the listed publications, the author of this thesis was the first author and the primary contributor to the research idea, analysis and reporting of the research work. In addition to writing, the author of this thesis implemented the GUI Driver tool for

model extraction of Java based GUI applications (Article I), extended the GUI Driver tool to allow user input during the model extraction (Article II), compared the performance of the GUI Driver tool to the GUITAR tool and proposed a widget classification to allow GUI exploration strategies for faster model extraction (Article III), interviewed Matias Suarez (the main developer of the Murphy tools at F-Secure Ltd) on implementation and functionality of Murphy tools (Article IV) and on industrial experiences on using Murphy tools at F-Secure (Article V), performed an extensive literature study on automated GUI testing (Article VI), draw conclusions based on the state-of-the-art study and industrial experiences of the previous papers (Article VII), defined an improved classification of automated testing methods (Article VIII), and summarized the challenges in GUI testing based on literature study and experiences on several tools (Article IX).

Contents

1 Introduction

1.1 Background and motivation

Our daily lives have become dependent on software functioning without errors. A defect in a smartphone or web application might just cause a nuisance for the end user. However, life and money are at stake when software is controlling the more critical parts of our world, such as airplanes, hospitals, banks, etc. In 2016, a software error in an automatic train control system at Denver airport caused passenger injuries requiring medical care in a hospital (National Transportation Safety Board, 2017), and a software bug in medical risk assessment software resulted some patients suffering the side effects of a medicine they did not need and others not getting the medicine they needed (Matthews-King, 2016). In 2017, there was a total chaos at British airports due to a server failure during maintenance at the data centre of British Airways. Software failures are having more and more impact, in 2017, the Software Fail Watch reported over 1.7 trillion USD losses caused by software failures in 2017, and getting more attention from the news (Tricentis, 2018). Assuring the correct and reliable behaviour of software systems is extremely important, and testing is a critical part of the quality assurance (QA).

The widespread use of iterative and incremental processes and continuous integration (CI) practices in software development has shortened the development cycles, drastically limiting the time for testing and QA of each release. Instead of months or weeks, the longest period for testing a release is over a weekend or a night. The results of an automated smoke test set are expected almost instantly or in a few minutes. In practice, the use of test automation is a requirement for a successful CI process.

At the same time the software systems are getting ever more complex, systems of systems with multitude of platforms and devices to support. The complexity of the systems being developed makes also software testing more difficult. A major part of test automation has been concentrating on automating the execution of test cases. The combination of shorter time for testing and more software to test adds pressure to develop testing methods and tools that are more intelligent and efficient, reducing the manual effort from all phases of the testing.

The execution of unit tests is widely automated, and there are tools for unit test generation, such as EvoSuite[1] (Fraser & Arcuri, 2014), but system level testing is

[1] http://www.evosuite.org/

more difficult to automate, especially if the system includes graphical user interfaces (GUI) for the end users. Testing software applications through their GUI is important since it can reveal subtle and annoying bugs, but expensive (Pezzè, Rondena & Zuddas, 2018) and challenging, not only because of the combinatorial explosion in the number of event sequences, but also because of the difficulty to cover the large number of data values (Cheng, Chang, Yang & Wang, 2016).

Due to its repetitive nature, regression testing is usually one of the first areas of testing to be automated, in addition to unit testing. According to IEEE (IEEE, 1990, 61), regression testing is the "selective retesting of a system or component to verify that modifications have not caused unintended effects..." The idea is to show that software that previously passed the tests, still passes the same tests after changes. In practice, regression testing aims to discover changes in the behaviour of the system under testing (SUT). If the change was intentional, usually the regression test cases have to be updated to correspond to the new behaviour. If the change was unintentional, a regression bug was found.

1.1.1 Script-based GUI testing

Script-based GUI testing is a common industry practice to automate testing through GUI. By script-based GUI testing, we mean any kind of GUI test cases or sequences that have been manually created prior to test execution, basically automating only the execution of the test cases.

With capture & replay (C&R) tools, e.g., commercial tools Rapise[2], Squish[3], and Ranorex[4], the scripts are recorded during manual use of the SUT through its GUI. Then the recorded scripts can be automatically executed on another version of the same GUI to detect changes in the behaviour. The test oracle is the presumably correct behaviour of the recorded version. The advantage of C&R is that it does not require special skills to use. Recording of the test cases is very similar to manual GUI testing. C&R techniques differ mainly in the kinds of GUI technology they can handle (Pezzè et al., 2018), but more advanced C&R tools provide a graphical editor to make it easier to change and maintain the recorded scripts.

[2] https://www.inflectra.com/Rapise/
[3] https://www.froglogic.com/squish/
[4] https://www.ranorex.com/

With other script-based tools, the test steps of the test scripts are manually defined with some scripting language that can be textual, as with AutoIt[5], or the tool provides a graphical editor for scripting, as with SeleniumHQ[6], and Visual GUI Testing (VGT) tools SikuliX[7] and EyeAutomate[8].

The main challenge in industrial adoption of script-based GUI testing is the maintenance effort required when the GUI changes (Alégroth, Feldt & Kolström, 2016). With C&R tools any test sequence going through the changed parts of the GUI might have to be manually recorded again. With scripting languages, any script going through the changed parts might have to be manually updated. The more advanced tools are better resistant to small GUI changes, but a bigger change in the GUI will cause the test scripts to fail, resulting false positives until the scripts have been repaired. The maintenance effort greatly diminishes the return of investment (ROI) of script-based GUI test automation. Another challenge is the effort required to create the test scripts in the first place. Usually, the one-time investment in the beginning is not as big a problem as the continuous maintenance.

If the test scripts are manually defined or the recorded scripts are manually elaborated, it is possible to define test oracles that check the correctness of any value in any specific state of the GUI. However, each check has to be defined separately, and obviously, this increases the manually defined test artefacts that have to be maintained when the GUI changes. Most script-based GUI testing approaches check only if the most important data values are correct in a specific state of the GUI. Manually checking all the values of all the properties of all the widgets of each GUI state is not feasible even without the maintenance effort, but setting too few values may lead to ambiguous test verdicts (Article VI).

There are academic research approaches on automated GUI script repair to tackle this maintenance problem, e.g., VISTA (Stocco, Yandrapally & Mesbah, 2018), SITAR (Gao, Chen, Zou & Memon, 2016), but none of them have been widely adopted in the industry. Some commercial tools, e.g., Squish, claim to update the test scripts automatically when the GUI changes, but no details or data is given whether this works in practice and how.

[5] https://www.autoitscript.com/site/
[6] https://www.seleniumhq.org/
[7] http://sikulix.com/
[8] http://www.eyeautomate.com/

1.1.2 Model-based GUI testing

There are various model-based GUI testing (MBGT) approaches, e.g., using Spec Explorer in a tool chain (Silva, Campos & Paiva, 2007), NModel tool (Chinnapongse, Lee, Sokolsky, Wang & Jones, 2009), and Pattern Based GUI Testing (PBGT) (Moreira, Paiva, Nabuco & Memon, 2017), trying to reduce both the initial effort in creating the test scripts and the maintenance effort required after each change in the GUI. The difference to the script-based testing is that in model-based approaches the scripts are automatically generated. There are also various kind of models used for MBGT, e.g., state-based models like finite state machines (FSMs) (Miao & Yang, 2010), and event-based models like event-flow graphs (EFGs) (Memon, Banerjee & Nagarajan, 2003a).

In traditional MBGT, a model of the expected behaviour of the GUI is manually designed with a modelling tool. Then, a test case generation tool is used to generate either abstract or executable test cases from the model. The advantage of such manually created models is that the expected behaviour captured in the model enables generating system-specific automated test oracles.

The challenge in MBGT is the specialized formal expertise and effort required for creating the models that allow automated test case generation and execution on real-life GUI applications (Bertolino, Polini, Inverardi & Muccini, 2004). Designing a test model on a suitable level of abstraction, and following the exact modelling syntax supported by the test case generation tool is not a trivial task.

GUI ripping (Memon, et al., 2003a) and other GUI model extraction or inference approaches, e.g., GUI Driver (Article I) and GuiTam (Miao & Yang, 2010), try to help in creating the GUI models for testing. There have been some static approaches based on source code analysis, but it is difficult to capture the dynamic behaviour of the GUI without executing it. Most dynamic approaches, e.g., GUI Driver (Article I) and Extended Ripper (Amalfitano, Fasolino, Tramontana & Amatucci, 2013), analyse the behaviour of the GUI during run-time while emulating the end user by automatically interacting with the GUI widgets to traverse through the GUI. During model extraction, it is possible to use generic checks to detect failures, such as unhandled exceptions and crashes.

Many approaches, e.g., GuiTam (Miao & Yang, 2010), GUITAR[9] (Memon, Banerjee, Nguyen & Robbins, 2013), and GUI Driver (Article I), have used the extracted GUI models for generating test sequences and executing the test

[9] https://sourceforge.net/projects/guitar/

sequences on another version of the same GUI. With the generated scripts, it is possible to detect changes and regression bugs between the SUT versions, in addition to generic failure checks. When generating test cases from a model extracted from previous GUI version and executing them on the new version, the challenge is that the test cases will not test or notice any new parts of the GUI. The new parts were not in the extracted model, so they cannot be in the generated test cases either. Test cases fail when something included in the test cases is removed or changed, as illustrated in Fig. 1. Usually it is easy to extract a new model from the new version, but then we lose the reference behaviour that is used to discover the changes.

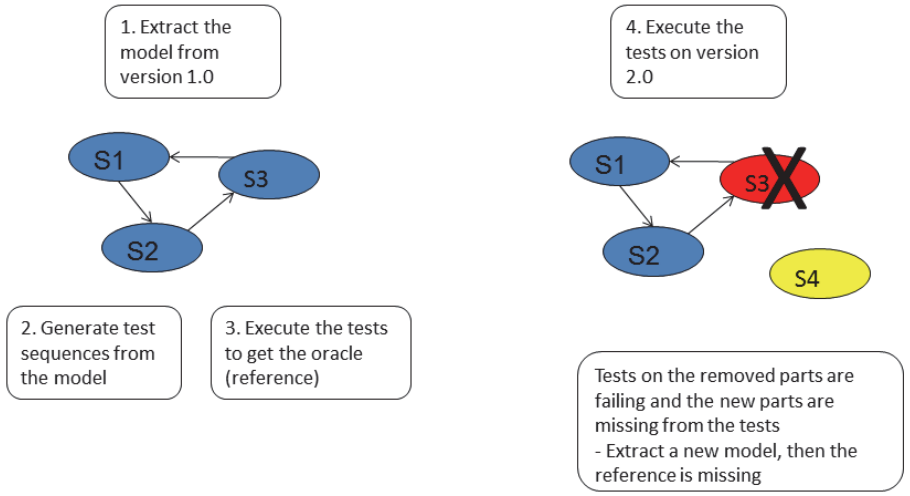

Fig. 1. Challenges in using extracted GUI models for generating regression test cases.

Another challenge is that GUI models extracted from the implementation do not capture expected behaviour. Instead, they capture implemented behaviour that might contain errors, so using them for testing is not so easy or effective. It is difficult to generate good test oracles from extracted models, unless the models have been manually elaborated to include expected behaviour too.

1.1.3 Scriptless GUI testing

Random testing or monkey testing is a scriptless approach for automated testing through GUI. Usually, it is a very cheap investment to take it into use and it requires very little if any effort in maintenance. Random testing can be surprisingly effective

(Böhme & Paul, 2016), especially in finding failures with "improbable" test sequences that were not specified or properly implemented, or just long sequences finding failures for example related to memory management. However, random testing requires a lot of execution time. Random testing can be executed in parallel, so with enough resources it is possible to get the results faster.

A bigger challenge with random testing is the test oracle. Most random testing tools use only generic checks to find exceptions and crashes or getting the GUI into unresponsive state, not supporting manually defined application specific test oracles or regression testing between SUT versions. TESTAR[10] tool (Vos, Kruse, Condori-Fernández, Bauersfeld, & Wegener, 2015) provides an option to define application specific oracles and instructions for the GUI traversal.

1.1.4 Automated GUI change analysis

In this thesis, we propose using automated GUI model extraction for automated change analysis between consequent versions of the SUT. Instead of using the extracted model of version A for generating test scripts, and running the scripts on version B, we propose to extract a new GUI model from each version of the SUT, and compare the models of consequent versions to detect changes, as illustrated in Fig. 2. This way we have the reference behaviour in the previous model and we will not lose any information due to an incomplete test generation. Our approach aims to reduce the need for manually creating models or test scripts for regression testing through GUI.

[10] https://testar.org/

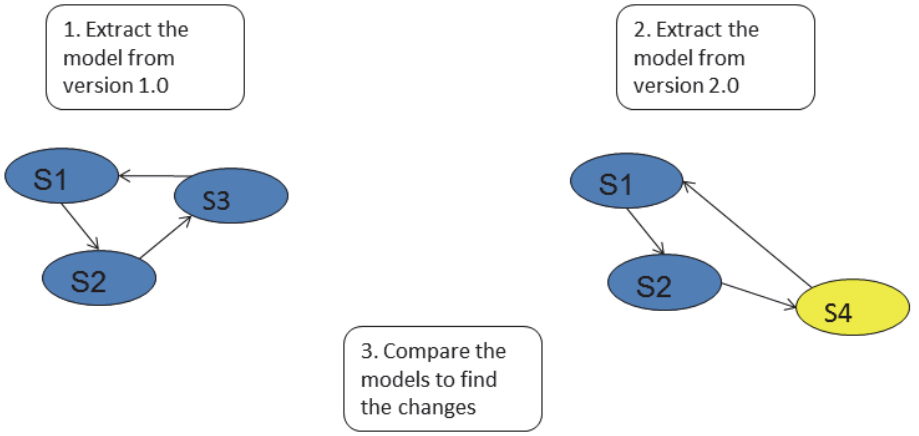

Fig. 2. Automated GUI change analysis based on extracted models.

During the model extraction, SUT can be tested against the general test oracles, such as crashes, unresponsiveness, and unhandled exceptions, as presented in Article I. It would be possible to specify application specific test oracles too, as in TESTAR tool (Vos et al., 2015).

The proposed approach requires a way to automatically extract a behavioural model of the GUI, on a level of abstraction that allows detecting changes in the GUI but restricts the state space to a reasonable level that allows recognizing the similarities between the versions. The challenges of GUI model extraction and GUI state abstraction, stated in Research Question 1 in the next chapter, are addressed mainly in Articles I and III of this thesis.

Another challenge in GUI model extraction is automatically reaching all parts of the GUI to capture sufficient coverage of the GUI, or limiting the model to the relevant parts of the GUI. Usually, some application specific instructions for the model extraction tools are required, e.g., providing a valid username and password, limiting the parts of the application to be modelled, or providing instructions to execute specific actions in a specific state of the GUI. This challenge is addressed mainly in Articles II, III and IV of this thesis.

The proposed approach for automated change detection through GUI is presented in Article IV. Murphy tools[11] are presented and evaluated in industrial setting with continuous integration process in Article V. Article VII analyses and generalizes the challenges being addressed by the proposed approach.

[11] https://github.com/F-Secure/murphy

1.1.5 Open issues and challenges in automated GUI testing

As stated in Article IX:

> Even if the test execution is automated, testing through GUI is significantly slower than for example traditional unit testing. The main reason is that you have to wait for the GUI to update after each test step or GUI interaction.
>
> GUI state comprises of all the screens visible for the user, all the widgets of each screen, and all the properties and values of each widget. In addition, the previous states of the GUI might affect the behaviour and therefore might be relevant for deducing the current GUI state. State space explosion is a challenge for modelling or extracting GUI models for testing.
>
> Another challenge in automating GUI testing is the test oracle (Barr, Harman, McMinn, Shahbaz & Yoo, 2015). Complexity of the state of the GUI makes it difficult to deduce whether the state of the GUI is correct, and even more difficult task to automate. Usually a GUI test case contains a sequence of events or interactions, and the oracle invocation has to be interleaved with the test case execution, because an incorrect GUI state in the middle of the sequence may lead to an unexpected screen, making further test case execution useless (Memon, 2002). Also, pinpointing the actual cause of an error may otherwise be difficult, especially when the final output is correct but the intermediate outputs are incorrect (Memon, 2002).
>
> One of the general challenges in testing is to objectively justify the decision when to stop testing (Belli, 2001). It is challenging to define and implement coverage criteria for not measuring only code coverage of the test suite, but also coverage of all the possible GUI interactions and the meaningful combinations of interactions (Yang, 2011), (Yuan, Cohen & Memon, 2011). On the other hand, a good test suite is the one that is able to detect the most faults. A “good test suite” can refer also to other qualities, e.g., requiring a short time or otherwise few resources to create or execute, or being easy to maintain (Strecker & Memon, 2008).
>
> Manual GUI testing, for example during exploratory testing, can give a good overview and gut-feeling of the usability and overall quality of the system. This part of manual GUI testing is very difficult to automate. With end user GUI applications, the customer might choose the product just by the look-and-feel of the GUI.

1.2 Scope, research questions, objectives, and approach

1.2.1 Scope

The focus of this research, and the topic of the first research question, is on automated state model extraction using dynamic analysis through GUI. Although our long term goal is to develop a generic, platform-independent solution, this thesis focuses on desktop applications. We limit our scope by not concentrating on GUIs of web or mobile applications, nor static source code analysis.

As the second research question, we research how to use the extracted GUI state models for automating testing. First we look into generating test cases from the extracted models, but as explained in Chapter 1.1.2, that approach has some shortcomings and restrictions. Our second approach is using model comparison to discover changes between SUT versions, and that appears to be more promising way to use extracted GUI state models for automating testing.

1.2.2 Research questions and objectives

The research questions of this thesis have been divided between two main parts: The first part studies automated extraction of state-based models of GUI applications and related challenges. The second part studies how to exploit extracted GUI models to automate testing of the modelled applications through their GUI. The research questions and the related objectives are listed below.

RQ1: How to automatically extract state-based models of GUI applications?

- RQ1.1: How to deal with the state-space explosion?
  - The objective is scaling up to non-trivial systems while maintaining sufficient accuracy in extracted models.
- RQ1.2: How to reach all parts of the GUI?
  - The objective is reaching a sufficient coverage of the GUI in a reasonable time used for model extraction.
- RQ1.3: How to restrict the modelling to the interesting parts of the GUI?

  - The objective is to include only the interesting parts of the system to the extracted models, for example excluding a web page opened through a Help-menu.
- RQ1.4: How to extract GUI state information in a generic way?
  - The objective is to be able to extract GUI state information without depending on a specific implementation language, operating system or platform.
- RQ1.5: How to validate that the extracted GUI state models conform to the expected behaviour?
  - The objective is to use extracted models in testing, and the observed behaviour of the SUT might include errors, so the models have to be validated in some way.

RQ2: How to use extracted GUI models to automate testing?

- RQ2.1: How to generate test cases from extracted GUI models?
  - The objective is to reduce manual modelling effort by using extracted GUI models for model-based GUI testing.
- RQ2.2: How to automatically detect changes in the GUI based on comparing extracted models of consequent GUI versions?
  - The objective is to replace part of the regression tests, reducing the maintenance effort of GUI testing in continuous integration process.

1.2.3 Approach

The first part of this research was conducted in an iterative and incremental way, involving proof-of-concept tool implementation, evaluating and validating the tool implementation by using it in case studies of open source GUI applications, and analysis and planning the next iteration. The tool implementation was also compared with a related academic tool GUITAR. The GUI Driver tool and Articles I, II and III were the results of this first part.

The second part of the research was about Murphy tools, and involved qualitative methods through interviews and collaboration with practitioners in the industry, especially at F-Secure Ltd. The case studies involved commercial GUI

applications during their development, in a highly automated continuous integration process. Articles IV and V were the results of this collaboration.

Interleaved with both parts, a comprehensive state-of-the-art literature study on automated GUI testing, reverse engineering and specification mining, and using extracted model to automate GUI testing was conducted and published as Article VI. The results of the literature study and the experiences from the industrial collaboration around Murphy tools were combined and generalized in Article VII. In Article VIII, a new classification of automated GUI testing approaches was proposed based on the literature study. Article IX summarizes the solutions and remaining challenges of the research topic.

1.3 Mapping the contributions of the articles into the research questions

The following Table 1 maps the contributions of the articles to the research questions specified in Chapter 1.2.2. After the table, more detailed explanations are given in Tables 2-10, one table for each article.

Table 1. Mapping between the articles and the research questions.

Research Questions:	GUI Driver Articles I, II, III	Murphy tools Articles IV, V	State-of-the-art and other generic Articles VI, VII, VIII, IX
RQ1	I, II, III	IV, V	VI, VII, VIII, X cover all the research questions at a more generic level
RQ1.1	I: state abstraction, II: abstract data values	IV: state abstraction	
RQ1.2	II: iterative process for input, III: widget classification, comparison with GUITAR	IV: SUT-specific instructions	
RQ1.3		IV: boundary nodes	
RQ1.4		IV: 3 drivers, image recognition is the most generic	
RQ1.5		V: visual model inspection	

Research Questions:	GUI Driver Articles I, II, III	Murphy tools Articles IV, V	State-of-the-art and other generic Articles VI, VII, VIII, IX
RQ2	I, II, III	V	
RQ2.1	I: tool chain for MBGT, II: errors found, III: comparison with GUITAR	V: using model to define and generate test cases	
RQ2.2		V: automated change detection by model comparison	

Table 2. Article I contributions to the research questions.

Research Question	Contribution
RQ1	Presenting an approach for automatically extracting state-based GUI models and implementation of a proof-of-concept tool GUI Driver.
RQ1.1	Presenting an approach for state abstraction based primarily on the enabled GUI actions and secondarily on the structural state information.
RQ2 & RQ2.1	Presenting a tool chain to generate test sequences from extracted GUI models, and executing the sequences with GUI Driver to generate test reports. Reporting various types of errors and issues that were find with the approach.

Table 3. Article II contributions to the research questions.

Research Question	Contribution
RQ1.1	More details on the proposed state-based GUI modelling approach, state abstraction method, and GUI Driver tool. To fight the state space explosion, the data values of the GUI are coupled with actions (state transitions) instead of states.
RQ1.2	Extending GUI Driver tool and presenting an iterative process to 1) automatically exploring the GUI and extracting models, and 2) manually providing valid input into the GUI to reach new parts of the GUI.
RQ2.1	Presenting a few examples of errors found with the approach.

Table 4. Article III contributions to the research questions.

Research Question	Contribution
RQ1	Presenting a comparison of GUI model extraction techniques and experimental comparison between GUITAR and GUI Driver tools.

RQ1.2	Proposing a classification of GUI components that can be used for improved action selection strategies. Discusses the possible strategies for reaching as many states as possible with as few actions as possible. Comparing GUI Driver screen coverage with GUITAR.
RQ2.1	Comparing test case generation with GUITAR.

Table 5. Article IV contributions to the research questions.

Research Question	Contribution
RQ1	Presenting another GUI model extraction approach and Murphy tools.
RQ1.1	Murphy tools abstract away data values from the state information to reduce the number of possible nodes in the state model.
RQ1.2	Murphy tools introduce an invocation script to separate application-specific modelling instructions from generic GUI modelling library.
RQ1.3	Murphy tools provide boundary nodes as a means to restrict the modelling to the interesting areas of the GUI application.
RQ1.4	Murphy tools provide a plugin architecture for implementing the extraction of GUI state information and implements various methods, including Windows specific, application specific, and image recognition based methods.

Table 6. Article V contributions to the research questions.

Research Question	Contribution
RQ1	Presenting results from a long term industrial experiment, showing that state-based GUI model extraction scales to commercial GUI applications and can be used as a part of a continuous integration process.
RQ1.5	Murphy tools provide an abstracted, human-readable graphical presentation of the extracted GUI model, allowing visual inspection and validation of the modelled behaviour.
RQ2.1	Murphy tools provide a web UI with a graphical presentation of the extracted model that can be used to manually select specific paths through the model and generate test cases for the selected path.
RQ2.2	Murphy tools was used in continuous integration process to extract model of the latest GUI application version three times a day, and automatically detect the changes by comparing the models of consequent versions. The changes were illustrated with a web UI showing the screenshots of the new and old versions of the changed states. Based on the experiences at F-Secure Ltd, the amount of manually written test code and the related maintenance effort was reduced, and creating new test scripts was easier and faster. Murphy tools were also used to support manual GUI testing by automatically launching a virtual machine with the GUI application in the selected state, reducing test setup time for manual GUI testing.

Table 7. Article VI contributions to the research questions.

Research Question	Contribution
RQ1 & RQ2	A comprehensive state-of-the-art study on automated extraction of GUI models for testing.

Table 8. Article VII contributions to the research questions.

Research Question	Contribution
RQ1.1	Providing a more general view on the challenge and possible solutions for state space explosion in extracting state-based GUI models.
RQ1.2	Providing a more general view on the challenge and possible solutions for reaching all parts of the GUI during automated exploration and model extraction.
RQ1.4	Providing a more general view on the challenge and possible solutions for extracting GUI state information in a generally applicable way.
RQ1.5	Providing a more general view on the challenge and possible solutions for validating the correctness and coverage of the extracted GUI models.
RQ2	Providing a summary of how to use extracted models to automate GUI testing and for automated GUI change detection.

Table 9. Article VIII contributions to the research questions.

Research Question	Contribution
RQ1 & RQ2	Providing a synthesis and proposing a classification for methods and tools for automated regression testing through GUI.

Table 10. Article IX contributions to the research questions.

Research Question	Contribution
RQ1 & RQ2	Summarizing the remaining challenges and state-of-the-art in GUI model extraction and automated GUI testing.

1.4 Structure of the thesis

This thesis is structured as follows: Chapter 1 provides the introduction, including background information and motivation for the research, describes the scope of the research, lists the research questions and objectives, describes the research approach, and maps the contributions of the articles into the research questions.

Chapter 2 is structured by the research questions. For each research question, including the sub-questions, first a generic description of the challenge is given, then the contributions of this thesis are described, and a comparison to the related work is given. Chapter 3 summarizes the contributions of the thesis, presents the conclusions based on the results, and discusses the threats to validity and future work. The articles of this thesis are included after references (but not in the electronic version).

2 Contributions and related work on the research questions

This chapter presents the contributions of the articles and the state-of-the-art on each research question of this thesis. A comprehensive state-of-the-art study on automated GUI testing, reverse engineering and specification mining, and using extracted model to automate GUI testing is presented in Article VI and updated in shorter but more recent Articles VIII and IX.

2.1 RQ1: Automated extraction of state-based models of GUI applications

Model extraction, also known as model inference or reverse engineering of models, is an approach aiming to generate models from an existing implementation or other artefacts of a system. In this thesis, we limit the scope to extracting state-based models of applications with a GUI.

Contributions of this thesis

Article I proposes an approach and introduces a proof-of-concept tool GUI Driver for automated extraction of state-based models of Java GUI applications. The approach uses dynamic analysis on the behaviour of the application observed through its GUI during automated exploration. The GUI Driver tool was implemented using Jemmy Java library[12] for creating automated tests for Java GUI applications. Jemmy provides an application programming interface (API) for the GUI Driver tool to observe a Java GUI and get the information required to create a GUI state model and to interact with the observed widgets of the GUI. The generated models include structural tree models generated for each state of the GUI application and a single state graph capturing the behaviour as states and state transitions. The structural tree models, saved into XML files, present a snapshot of the GUI state in a specific moment of time, comprised of all the widgets and their properties and values. The state graph presents the behaviour of the GUI and maps the structural models into the abstract states.

Article II extends the research on GUI Driver, and proposes an iterative process of 1) automated GUI exploration and 2) manually providing valid input directly

[12] http://jemmy.java.net

into the input fields of the GUI being modelled. The goal is to extract state-based GUI models with better GUI coverage. Article III presents a comparison of GUI model extraction techniques and an empirical comparison between GUITAR and GUI Driver tools.

Article IV introduces Murphy tools for automatically extracting models of GUI applications from the user interface (UI) flow, and using the created models for GUI testing. Originally, F-Secure Ltd developed Murphy tools for their internal use, but currently it is available as open source. Murphy dynamically analyses the GUI while automatically interacting with the application, as if it were an end user trying out all the possible user interactions, such as entering text in a text field, pressing a button or a link, selecting items or ticking checkboxes. The main idea is to traverse through all the possible states of the GUI application and automatically construct a state-based model of the observed behaviour during the automated exploration or "crawling" the GUI. Internally Murphy creates a directed graph to model the behaviour of the GUI application. The nodes of the model are states of the GUI screen or window, and the edges are actions that the end user could perform in that specific state of the GUI. Murphy tools are implemented in Python and using Windows accessibility API and image recognition to extract GUI state information, so any Windows application with a GUI is supported. Article V evaluates the Murphy tools in industrial development environment using continuous integration process.

Related work

TESTAR is an open source tool for automated testing through GUI. It is based on automated exploration of the GUI and maintains an internal state model that can be used for action selection strategies and visualization of the test results (Rueda, Esparcia-Alcazar & Vos, 2016). TESTAR is implemented mostly in Java, and it uses Windows accessibility API for extracting the GUI state information, in a similar way as Murphy tools. Although TESTAR development has been started during 2010-2011, at the same time as the GUI Driver tool, the model extraction functionality is a more recent extension of TESTAR. Currently, TESTAR is being extended to use a graph database for saving the extracted state models.

Miao and Yang (2010) propose an FSM based GUI Test Automation Model (GuiTam) and a dynamic reverse engineering approach to automatically construct the models and use them for GUI testing. Run-time GUI information is

programmatically readable, which provides an opportunity to automatically generate a GuiTam for an application by traversing its GUI states (Yang, 2011).

Memon et al (2003a) propose GUI ripping, a dynamic process for automatically analysing the structure of GUI and using the captured GUI tree to create event-based models that can be used for test case generation. Although Memon's work can be considered a pioneer of model extraction through GUI, the extracted models are event-based and therefore less related than state-based approaches.

Paiva, Faria and Mendes (2007) propose a semi-automated dynamic technique, mixing manual and automated GUI exploration. The goal of the approach and presented REGUI2FM tool is to reduce the modelling effort and provide mapping information for executing abstract test cases on a concrete GUI during MBGT process. The extracted partial "as is" Spec# model is validated and completed manually into "should-be" model, including also expected outputs to be used as test oracles. Morgado, Paiva and Faria (2012) developed the approach further into ReGUI tool 2.0, using automated GUI exploration and dynamic analysis to create the models. ReGUI 2.0 uses Windows accessibility API to extract the GUI information and interact with the GUI.

Tramontana, Amalfitano, Amatucci and Fasolino (2018) have published a comprehensive systematic mapping study on automated functional testing of mobile applications. Although a long term goal would be a generic, platform-independent solution, this thesis focuses on desktop applications. Also, using extracted models to automate testing does not fit into traditional definition of functional testing, because usually a direct link to functional requirements is missing.

Amalfitano, Fasolino, Polcaro and Tramontana have published extensively on dynamic reverse engineering of rich internet applications (RIAs) and mobile applications, for example DynaRIA tool (2013) for RIAs, and AndroidRipper tool (Amalfitano, Fasolino, Carmine, Memon & Tramontana, 2012) for Android applications.

Mesbah, van Deursen and Lenselink (2012) presented a technique and open-source tool called Crawljax for crawling asynchronous JavaScript and XML (AJAX) based applications to dynamically infer a state-based model, state-flow graph, modelling the various navigation paths and states within the application. The reconstructed models can be used to generate linked static pages for exposing AJAX sites to general search engines, accessibility improvements, or in

automatically exercising all UI elements and conducting state-based testing of AJAX applications.

AutoBlackTest[13] (Mariani, Pezzè, Riganelli & Santoro, 2012) is a test case generation technique that builds the model and produces the test cases incrementally, while interacting with the application under test. AutoBlackTest extends IBM Functional Tester[14], a commercial capture and replay tool, to extract GUI information and interact with the GUI.

Augusto[15] (Mariani, Pezzè & Zuddas, 2018) is a GUI test case generation technique that can automatically produce system test cases for application independent functionalities (AIF), systematically covering semantically relevant scenarios and including precise functional oracles that can reveal non-crashing faults. Augusto encodes the high-level commonly expected semantics of AIFs into models that are automatically adapted to the specific characteristics of the application under test (AUT). Augusto dynamically extracts the GUI model and recognizes patterns that are manually modelled into AIFs using Alloy [16] specification language.

SAPIENZ[17] (Mao, Harman & Jia, 2016) is an approach for multi-objective search-based testing to automatically explore Android applications. SAPIENZ aims to optimise test sequences, minimising length, while simultaneously maximising coverage and fault revelation. SAPIENZ has been further developed and taken into industrial use at Facebook (Alshahwan et al., 2018). As a tool for Android testing only, we consider SAPIENZ out of our scope.

Active automata learning uses a test-based approach for inferring models of black-box systems. Observations are made by actively interacting with a system under learning (SUL). Active automata learning has been used to learn through GUI of web application (Bainczyk et al.2016) and Java GUI applications (van der Laan, 2017).

2.1.1 RQ1.1: State-space explosion

State-space explosion remains a challenge in GUI modelling, especially when automatically extracting state-based models (Yang, 2011). Any nontrivial program

[13] http://www.lta.disco.unimib.it/tools/
[14] https://www.ibm.com/us-en/marketplace/rational-functional-tester
[15] https://github.com/danydunk/Augusto
[16] http://alloy.lcs.mit.edu/alloy/
[17] https://github.com/Rhapsod/sapienz

has a large number of possible states, depending on the definition of the state and how to distinguish them. Without applying any abstraction for GUI state definition, using all widget properties and their values, even such a simple GUI application as a calculator will have impractically large number of states. For example, pressing any of the number buttons will add that number into the calculator display, creating a new state, at least until reaching the maximum digits of the display. Over the years, optimizations to the original learning algorithms have yielded significant improvements in terms of the speed of model inference and the size of extracted models, making it possible to infer state space sizes of 100 000 states or more, which is sufficient to test many kinds of industrial applications (Meinke & Walkinshaw, 2012). The challenge is to find the balance between increasing expressiveness to extract more accurate models and keeping the computational complexity on feasible level for model inference and model checking. Abstracting away too much information from the SUT increases the risk of losing opportunities to discover faults (Meinke & Walkinshaw, 2012). In most proposed GUI model extraction approaches, the modelled applications have been rather small, not showing that the model extraction methods scale up to non-trivial GUI applications.

Contributions of this thesis

In Article I, GUI Driver tool uses two criteria for state abstraction. The first criteria for two GUI states to be considered the same is that both states must have the same enabled GUI actions. The secondary check is to compare the structural models of the states. The number of the structural changes that are tolerated for similar GUI states can be specified in the settings. An important benefit of using available actions as the first criteria of state abstraction is that the resulting state model is traversable and can be used as a map to plan the action sequences. Article II further increases the abstraction by reducing the widget properties that are considered relevant in state comparison. In practice, the data values of the GUI do not affect the abstract GUI state. Instead, the data values are coupled with the state transitions of the state graph, reducing the number of states but increasing the number of possible state transitions.

In Article IV, Murphy tools use various ways to capture the state of the GUI. When using Windows Accessibility API to get a widget tree, the state abstraction used is similar to GUI Driver, abstracting the data values. When using image recognition to detect changes in the appearance of the GUI, data fields are masked to abstract the GUI states. In other words, a dialog that has an OK button enabled

represents a different node than otherwise similar dialog with the OK button disabled, but a dialog would represent the same node regardless of the value in a text field of the dialog.

Article V presents empirical experiences on using the state abstraction of Murphy tools in industrial case study. During the evaluation, Murphy was used to extract models of 3 commercial GUI applications. Several versions of each application were modelled, because at any specific time there were at least 3 release versions on different phases of the quality assurance process, and the software developers added new features and released new versions during the development. The number of nodes (GUI states) in the extracted models was between 81 and 178.

Article VII proposes filtering out the external changes that might affect the detailed behaviour of the system, but are not relevant for the modelling purposes. A practical solution is to use virtual machines to stabilize the model extraction environment. A new, clean virtual machine can be launched automatically each time the GUI application has to be started or restarted.

Related work

In a GUI Test Automation Model (GUITAM) (Yang, 2011), only bi-valued properties are considered when judging whether two states are the same. That means all text fields and other widgets with more than 2 options are abstracted away when creating the state model. The principle is similar to our approach of abstracting data values, but simplified and less accurate.

TESTAR tool generates concrete and abstract state identifiers by calculating a hash of selected widget properties (Rueda et al., 2016). Abstract state identifier is computed based only on Role property of each widget of the state, abstracting all the other widget properties. The abstraction is the same for all kinds of widgets, making it simple, allowing quick comparison, but inflexible.

Salva and Zafimiharisoa (2014) propose to separate text field values from other widget properties. This approach is very similar to our approach but simpler and more recently published. In many cases it will produce more states than our approach, as other data values still affect the number of states.

He and Bai (2015) propose to use an average similarity of GUI state based on comparing the values of each widget property. This approach is simple but inflexible and inaccurate.

Paiva, Tillmann, Faria and Vidal (2005) propose to use hierarchical finite state machines (HFSMs) to reduce the number of states in the ”flat” FSM resulting from

the exploration of the model. The approach seems to divide the information into separate connected models instead of abstracting the state information. However, in practice it could work very well in visualization of complex models.

2.1.2 RQ1.2: Reaching all parts of the GUI

Using automated GUI exploration for dynamic analysis and model extraction has a challenge of automatically reaching all parts of the GUI. A common obstacle for automated exploration is a GUI state transition that requires correct, meaningful input, such as a login screen. Usually, a way to manually provide the valid input before or during the GUI exploration is required (Article II), meaning that the modelling is assisted manually by a person before or during the reverse engineering process (Kull, 2012). Another option could be an iterative approach consisting of automated GUI exploration and manual elaboration of extracted models, so that the automatically generated initial model is reviewed, corrected, and extended manually by a person after the model extraction (Kull, 2012). The efficiency of these semi-automatic modelling techniques depends on the degree of required human intervention (Kull, 2012). For model extraction purposes, the goal is to reach all parts of the GUI with minimum number of interactions, getting maximal GUI coverage as fast as possible. The main factor affecting the exploration is the action selection strategy used for selecting which of the available actions to execute in each state of the GUI.

Contributions of this thesis

To address the challenge of state transitions requiring specific input, Article I proposes using a graph editor tool to manually elaborate the generated GraphML model. The editor is used for adding valid input values, and the refined model is used for test case generation. As indicated before, Article II proposes an iterative process of automated exploration and waiting for the user to provide valid test data into the input fields of the GUI application. When automated exploration does not find any new actions or GUI states, it drives the GUI into a state with input fields and asks for manual input. The manually provided test data is saved into the generated state graph. This approach solves the challenge, is easy to use as it is similar to manual testing, and requires the user to monitor the model extraction process only when using the tool on a new application, or if the application has changed significantly.

Murphy tools (Articles IV and V) are using SUT-specific instructions defined in Python programming language, allowing the user to programmatically define specific input to be used or specific action to be taken in a specific GUI state. Usually, a couple of iterations are required, tweaking and fixing the instructions, to extract models with a good coverage of the GUI. The instruction file is used for invoking the generic UI crawling library, and can be modified with application specific rules, but often the generic UI crawler library is sufficient for generating the models. For simple GUI applications, the instructions will merely setup the initial state, such as start up the application to be modelled, and then invoke the generic UI crawler library. For more complex GUI applications, application specific modifications can be used for adjusting or correcting the behaviour of the generic UI crawler library, for example adding extra edges to a node when all interactions were not properly recognized, instructing the crawler of the values to be used in certain text fields, removing edges from a node, or instructing the crawler not to visit specific edges of the specified nodes.

To define the instructions for Murphy tools (Articles V and VII) that would reach all parts of the GUI and cover all the possible states and transitions of a complex GUI application would require several iterations and a lot of time and effort, even though the model extraction after defining the instructions would fully automated. Some of the GUI flows are difficult to explore, for example requiring external events, such as dialogs shown only when the network connection is lost, and would provide only a little added value for the automated testing. Therefore, it is up to the test engineers to decide when a sufficient coverage of the GUI has been reached and the iterative process of improving the instructions of Murphy tools is finished. Of course, the duration of model extraction process also depends on the size and complexity of the GUI being modelled.

In Article I, GUI Driver implements an action selection strategy that exploits the abstract state graph generated during the automated exploration of the GUI. By default, GUI Driver randomly selects one of the GUI actions that have not been executed in that GUI state before, and if all of the actions of the state have already been executed, it uses Dijkstra's algorithm on the state graph to find an action that follows the shortest path to a state with new actions. The enabled actions of consecutive states are compared and new GUI actions are prioritized over the ones that were available in the previous state. That way after opening a drop-down menu it is probable that one of the actions of that menu is selected.

Article II categorized widgets into three groups: GUI controls, GUI options and GUI inputs. The categorization is used to improve the action selection strategy,

considering only GUI controls as executable actions, and options and inputs only for providing test data values. The values of the options and inputs are always saved as properties of the selected GUI control. However, pseudo-random data selections and insertions are made to see whether they change the resulting GUI state after executing the controls of that state, or dynamically changes the GUI state by enabling or making visible one or more GUI actions. For example a button for editing the selected item might be enabled only when an item is selected from a list.

Article III extends the widget classification, dividing widgets into four groups: 1) GUI controls, 2) GUI options, 3) GUI inputs, and 4) GUI infos. An example of a more intelligent strategy is choosing GUI controls first without changing the data values, then again after changing one data value at a time. For example, in some GUI applications pressing buttons with empty input fields results in error messages informing the user to provide input first, and pressing the same buttons after providing the input triggers the intended functionality. Exploiting the extracted state model allows prioritizing actions that have not been selected yet or have been selected only once. If all actions of the current GUI state have been executed, based on the results of executed actions it is possible to select an action that leads to another GUI state with actions that have not been executed yet.

GUI applications tend to have a very large number of state transitions between the GUI states, making it impractical to try to reach a specific state multiple times through different paths. Instead, with a suitable level of abstraction, covering all the states of the GUI is more practical approach. Therefore, the goal of the Murphy tools (Articles IV and V) is to discover as many GUI states as possible. The default action selection of Murphy tools is similar to GUI Driver. It selects an unvisited action from the current GUI state. If there are no unvisited actions in the current state, it uses the model to look for a path into a state with unvisited actions. If a path is not found from the current state, it restarts the GUI to see if there is a path from the starting state. However, it is possible to customize the instructions of Murphy tools to overwrite the default action selection to contemplate special cases, for example if the models are meant to be used for testing all the possible transitions between specific states.

Related work

Paiva et al. (2007) proposed a semi-automated dynamic technique and REGUI2FM tool, mixing manual and automatic exploration to access parts of the GUI that are

protected by a key or are in some other way difficult to access automatically. This is similar to the iterative process of GUI Driver proposed in Article II.

For action selection, ReGUI tool 2.0 (Morgado et al., 2012) follows a depth-first algorithm on the widget layout order of GUI. The menu options are navigated first to extract the initial state of the GUI, i.e., which GUI elements are enabled/disabled at the beginning of the execution. Then, ReGUI navigates through all the enabled menu items and verifies whether they open any new windows. If so, ReGUI extracts its structure and closes it. After all menu items, ReGUI goes through all the menus again to verify whether a previously enabled element became disabled or vice-versa.

TESTAR tool (Vos et al., 2015) provides protocol classes as a programmable Java interface that can be used for instructing TESTAR how to automatically explore the GUI of a specific application. Protocol provides functions with default implementation that can be overwritten in application specific protocol implementation, such as DeriveActions(), SelectAction(), BeginSequence() and FinishSequence(). The protocols can be used for defining more advanced action selection algorithms than the default random selection, such as using genetic programming to evolve action selection rules (Esparcia-Alcázar, Almenar, Vos & Rueda, 2018), Q-learning strategies for action selection (Esparcia-Alcazar, Almenar, Martinez, Rueda & Vos, 2016), or for example providing correct login input or graceful shutdown of the SUT. This is very similar to the instruction files of Murphy tools presented in Articles IV-V.

AutoBlackTest (Mariani et al., 2012) integrates two strategies, learning and heuristics. It uses Q-Learning to learn how to effectively act in an unknown environment and multiple heuristics to effectively address a number of common but complex cases, such as compiling complex forms, executing the application with normal and abnormal values, and handling specific situations (for instance, saving files).

2.1.3 RQ1.3: Restricting the modelling to the interesting parts of the GUI

Successful automated GUI exploration brings another challenge: how to restrict the exploration and model extraction to stay in the interesting parts of the GUI, e.g., staying inside the application being modelled or in a specific part of it. Especially with web applications, there has to be a way to limit the exploration and model extraction, otherwise you might end up extracting the model of the whole Internet.

With desktop applications the same applies for browsing the file system, and many desktop applications have a “Help” menu that opens a web browser into web content.

Contributions of this thesis

In Article IV, Murphy uses boundary nodes to limit the scope of the UI crawler into the areas of interest. Boundary nodes are defined in the SUT-specific instruction file for marking the nodes that Murphy should not go beyond during the UI crawling, for example, when the GUI application launches a web browser and opens a web page, or when it is not feasible to crawl through the whole help system of the GUI application. Another way to use the instructions file to restrict the modelling is using triggers to select a specific action in a specific GUI state. For example, in a dialog for selecting the language for the installation of an application, the instructions could be modified to instruct the UI crawler library to crawl only the English version of the UI flow. The model extraction would have defined the edge for selecting the installation language into the model, but the UI crawler would ignore it and select English.

In our experience, it was also useful to add edges representing special actions with environment considerations, for example performing certain UI action when access to the network is not available, or when running low on memory. These special edges have to be manually inserted into the instructions, but in some cases they enriched the resulting model in useful ways.

Article V gives a practical example of using the boundary nodes for restricting the GUI exploration and model extraction. In the example, the file browser of the operating system is defined as a boundary node and recognized by the screen title “Browse For Folder”. In addition, the boundary nodes were used for partitioning the extracted models from one large model into a set of smaller and simpler models. This reduced the complexity of the models and made it easier to use the models in testing. During the experiment, separate models were created for the installation flow, the flow of actually using the application and the uninstallation flow. Also, each different supported language was modelled as a separate model. The partitioning is done with the boundaries in the SUT-specific instructions, and each model requires a separate instruction file with different boundaries.

Related work

TESTAR tool (Vos et al., 2015) distinguishes the SUT from other applications by processes of the operating system. If the SUT process is put into background, for example, as a result of an action that opens a browser, TESTAR brings it back to foreground. At the end of a test sequence, TESTAR kills all the processes started during the test run. The protocols of TESTAR provide means to program application specific rules that can be used for restricting the GUI exploration in a similar way as boundary nodes or triggers in Murphy tools.

2.1.4 RQ1.4: Extracting GUI state information in a generic way

There are existing frameworks that provide instrumentation for observing the GUI, such as Jemmy and Microsoft UI Automation[18], and the technical domain of GUI applications is wide but similar enough to be cost-effective through re-using the same expertise and tools on various systems. Nevertheless, most model extraction approaches for GUIs limit the applications that can be modelled to specific programming languages or platforms, usually based on the instrumentation framework used for observing the GUI (Article VII). It is challenging to provide platform independent GUI reverse engineering techniques and usually implementing support for each programming language and platform requires too much effort.

Contributions of this thesis

In Article IV, Murphy uses various approaches called drivers for recognizing elements and windows of the GUI application on various platforms to accomplish platform independency. Among the already implemented drivers,

- one uses Windows accessibility APIs for UI element detection and enumeration,
- another uses a proprietary API developed by F-Secure for enumerating and querying windows and elements of the UI, and
- a third driver simulates the end user by cycling through the UI elements by pressing the 'tab' key and using image recognition to analyse the changes in the screenshots to determine the elements and behaviour of the GUI.

[18] http://msdn.microsoft.com/enus/library/ms747327(v=vs.110).aspx

The idea is to automatically compare screenshots taken during the "focus shuffling" to find the changed areas, such as the bounding rectangle of the selected element on the screen and the shape of the mouse cursor. Moreover, we can reason about the structure and behaviour of the GUI based on the clues that the GUI application offers for the end user. Murphy provides generic UI crawling and window scrapping services as a library.

Although our long term goal is to develop a generic, platform-independent solution, this thesis focused on desktop applications. We have not evaluated the tools on mobile or web applications nor implemented drivers specifically for them, although the driver based on Windows accessibility API should work for mobile software running on an emulator and web applications on Internet Explorer or other browser on Windows.

Related work

Many tools use just one platform specific means to extract the information about GUI and interact with it. Using Windows accessibility API is quite common, e.g., ReGUI 2.0 (Morgado et al., 2012). AutoBlackTest (Mariani et al., 2012) extends IBM Functional Tester to extract GUI information and interact with it.

TESTAR tool (Vos et al., 2015) uses Windows UI Automation API as a default means to get the GUI state information from the operating system. However, originally TESTAR was developed for macOS (previously known as OS X), and there are implementations for Linux accessibility API and a SUT-specific implementation for REST API of an IoT system (Martínez, Esparcia-Alcázar, Vos, Aho, & Fons i Cors, 2018). Currently, an extension for using WebDriver for web applications is being implemented.

2.1.5 RQ1.5: Validating the correctness of the extracted GUI state models

When extracting models automatically by observing an existing system, the generated models are based on the observed implementation. As such, the generated models include also the undesirable behaviour of the system, instead of capturing the requirements or expectations of the system (Article VI). This limits the possibilities in utilization of the models. Most model extraction approaches have not addressed the challenge of validating the correctness and sufficient coverage of the modelled behaviour (Article VII).

Contributions of this thesis

Article V defines the process of validating the instructions of Murphy tools by visually inspecting the model and validating the correctness of the observed behaviour and extracted model. Because the model is based on the observed implementation, instead of requirements of the system, visual inspection and manual approval of the model is required to make sure that the modelled application behaves as expected. To help the user to understand the behaviour of the modelled application during the manual inspection, Murphy abstracts a lot of details and visualizes the GUI model as a directed graph with screenshots as GUI states and images of executed widgets as transitions between the states. As stated in Article V, the user inspecting the model should compare the modelled behaviour with the correct behaviour captured in the requirements or other specifications. If inconsistencies are found between the extracted model and the specifications, one of them had an error or details were missing from the specifications. Murphy was also used to extract models of all 29 different languages of the modelled products, and the visual presentations of the models were used to validate that the internationalization was working as expected.

As a more general rule, Article VII states that if the extracted model does not include all the parts or behaviour of the GUI, the model extraction has to be improved by instructing the extraction tool to include the missing parts. If the model includes parts that are not in the specifications, and the model extraction worked correctly, the problem is either in the modelled GUI application or the specifications. Either the incorrect behaviour of the application has to be fixed or the specifications have to be updated by discussing with the stakeholders.

Related work

Grilo, Paiva and Faria (2010) describe a dynamic approach for reverse engineering GUI applications using a combination of manual and automated steps in the modelling process. The tool uses Microsoft UI automation library for the automated steps and the created model has to be manually validated and completed with the expected behaviour. The final models are in Spec# format and can be used for model-based GUI testing (MBGT).

The goal of the approach by Paiva et al. (2007) and presented REGUI2FM tool is to reduce the modelling effort and provide mapping information for executing abstract test cases on a concrete GUI during MBGT process. The extracted partial

"as is" Spec# model is validated and completed manually into "should-be" model, including also expected outputs to be used as test oracles.

ReGUI tool 2.0 (Morgado et al., 2012) is able to generate also visual models that enable a quick visual inspection of some properties, such as the number of windows of the GUI.

2.2 RQ2: Using extracted GUI models to automate testing

Extracted models are based on observed behaviour of an implemented system, instead of requirement specifications or expected behaviour. Therefore, without elaboration, the extracted models are not well suited for generating test cases and test oracles, as in traditional model-based testing (MBT) (Article VII). Conformance testing, i.e., testing the implemented system against its specifications, requires a link to the requirements before using the extracted models for test case or test oracle generation (Article VI).

With conventional software, a test case usually consists of a single set of inputs, the expected result is the output that results from completely processing that input, and the oracle is invoked when the actual observed output is compared with the oracle's expected output after executing the test case (Memon, 2002). In GUI testing, the input may consist of a long sequence of actions, and there is no specific output as each executed action may affect the state of the GUI. The oracle information consists of a set of observed properties of all the windows and widgets of the GUI (Strecker & Memon, 2012). The execution outcome may depend on the internal state of the GUI application, the state of other entities (objects, event handlers) and the external environment, and may lead to a change in the state of the GUI or other entities. Moreover, the outcome of an event's execution may vary based on the sequence of preceding events or interactions seen thus far (Yuan & Memon, 2010). When defining test sequences prior to the execution into scripts (offline testing), an incorrect GUI state during a test sequence can lead to an unexpected screen, making further test case execution useless or impossible (Memon, 2002). As noted in Article VII:

> Therefore, the correct state of the GUI has to be verified after each execution step during a test case, interleaving the oracle invocation with the GUI test case execution. Otherwise detecting the actual cause of an error can become difficult, especially when the final output is correct but the intermediate outputs have been incorrect. (Memon, 2002) The oracle information for automated GUI

testing may be selected or created either automatically or manually based on requirements or other formal specifications of the GUI or observed behaviour of an earlier, presumably correct version of the software (Memon, Banerjee & Nagarajan, 2003b). By varying the level of detail of oracle information and changing the oracle procedure, a test designer can create different types of test oracles, depending on the goals of the specific testing process used. The different types of test oracles have a significant effect on test effectiveness and cost of testing. (Memon et al., 2003b)

2.2.1 RQ2.1: Generating test cases from extracted GUI models

As stated in Article V:

> In most approaches that use extracted GUI models for testing, the test oracles are based on the observed behaviour of an earlier version of the GUI application. Using this kind of test oracles, in literature often called reference testing, changes and inconsistent behaviour of the GUI can be detected and the models can be used for automated regression testing, but conformance testing is problematic (Kull, 2012). However, some defects, such as crashes and unhandled exceptions, can be detected without the use of application specific test oracles (Yang, 2011), making it possible to begin the testing of the GUI application already during the dynamic reverse engineering process, as in (Article II).

In Chapter 1.1.2 of this thesis, we describe the shortcomings and challenges of generating test cases from extracted GUI state models, illustrated in Fig. 1.

Contributions of this thesis

In Article I, GUI Driver tool is used for extracting GUI state models saved in GraphML[19] format that is applicable for the generation of test sequences with an open source GraphWalker[20] tool. The GUI Driver is able to:

- parse and execute the generated test sequences,
- verify whether the expected states were reached after executing each specified GUI action, and

[19] http://graphml.graphdrawing.org/
[20] https://graphwalker.github.io/

- create a test report including information about
  - executed GUI actions,
  - reached GUI states, and
  - abnormal behaviour, like unhandled exceptions.

As explained before, using generated models for testing does not provide practical means to validate if the system behaves as specified in the requirements. However, a lot of issues and problems can be found during the automated modelling and execution of the generated test sequences, but conformance to the requirements should be validated separately. In Articles I-III, GUI Driver was used to test open source Java GUI applications, detecting errors and usability issues, such as a cancel button that required valid input in a text field, dialogs without a way to cancel or to go back to the previous screen, and exceptions that were printed to standard output but not otherwise handled.

When introducing new test automation tools into use, the first step is often to replace the existing, possibly manually constructed but automatically executable test cases. In Article V, the Murphy tools provide a web UI for the user to specify test cases as paths in the extracted model, and generate executable test scripts that cover the selected path. Murphy visualizes the model with screenshots of the GUI, allows user to select states of the model and provide specific input for the test case, and randomly generates the missing parts of the path, if any. The generated test scripts can be used for example in smoke testing, automatically executed after each code commit. The amount of manually written test related code, and therefore the maintenance effort, is reduced and creating new test scripts gets faster and easier. It is seldom feasible to automate all GUI testing, and therefore Murphy provides support also for manual GUI testing. The user can use Murphy web UI to select a GUI state from the visualized model, and Murphy automatically creates a virtual machine, starts and executes the GUI application to the selected state, and directs the user to connect to the virtual machine with the GUI application in the desired state. The user can select a path of multiple states that should be visited and possibly the input values to be entered into the input fields of the GUI application. Murphy executes the application to visit all the selected states, automatically deducing the route if the whole path between the states was not defined. The user can continue to use the application manually, for exploratory testing or other manual verification purposes, and the time required for test initialization is reduced.

Article V reported experiences on using Murphy tools on commercial GUI applications. A large part of the behaviour of the GUI is already tested and directly

detectable defects, such as crashes and unhandled exceptions, are found during the model extraction phase. 220 existing manually written GUI testing scripts were replaced by deriving the corresponding test cases and scripts from the extracted models. Unfortunately, precise information about the effort used for manually creating the test scripts was not available, but defining the test cases and generating the scripts from the models was obviously faster and required less effort. As a result, the amount of hand written code specific for GUI testing was significantly reduced, reducing the effort for maintaining the test cases and creating new ones in the future. Murphy was used to support the execution of the manual GUI test cases for the modelled applications, and the time required for performing the manual GUI testing was significantly reduced. Although the reduction varied, depending on the application being tested and the particular test cases, generally the results were very promising. For example, when manual testing required over 30 minutes, it required less than 10 minutes to test the same test cases with the help of the generated models and the exploratory testing tool of Murphy. The main advantage was that Murphy automatically executed the tedious and repetitive steps and the steps that required waiting time. This way, the user had to execute only the steps that required manual analysis and verification of the results.

Related work

The first approach for generating test cases from automatically extracted GUI models was GUITAR tool by Memon et al (Memon et al., 2003a). The testing process with GUITAR consists of four main steps:

1. GUI ripping: Using a crawler-like tool called GUI Ripper to automatically launch and execute the application under testing (AUT). The GUI Ripper tries to expand hidden GUI elements and all the possible GUI windows, and capture the structure of the GUI into an XML-based model called GUI Tree (or GUI Forest). Each node of the GUI Tree represents a window and encapsulates all the widgets, properties, and values in that window (Memon et al., 2003a).
2. Model construction: Using gui2efg or another model converter to construct an EFG or another event-based graph model from the GUI Tree. In EFG, each node represents an event, and all events that can be executed immediately after this event are connected with directed edges from it (Memon et al., 2003a).

3. Test case generation: Using graph traversal algorithms to generate test cases or event sequences by walking on the event-based graph model with a given coverage criteria, such as covering all events or all edges.
4. Replaying: Using Replayer tool to execute test cases and verify the results.

The approach uses very limited GUI exploration for extracting event-based GUI models, resulting in inaccurate models and some test cases that cannot be executed in practice.

Miao and Yang propose (Miao & Yang, 2010) an FSM based GUI Test Automation Model (GuiTam) and a dynamic reverse engineering approach to automatically construct the models. Test cases can be automatically generated by traversing the states of the GuiTam. The test oracle information will be captured by performing the same test cases on an earlier version of the same GUI application. After the oracle information is captured, the test cases can be executed on the modified version of the AUT. By comparing the observed state behaviour with the corresponding oracle information, differences can be detected and possible defects can be discovered (Miao & Yang, 2010). Test cases can be generated by randomly selecting a given number of paths from the GuiTam, or by traversing the states and transitions of GuiTam according to given criteria, e.g., an algorithm can select the paths from the GUITAM until all states are covered to meet the state coverage criterion (Yang, 2011). The paths can be as long as possible so that the minimum number of test cases can be generated which satisfy the criterion (Yang, 2011). This approach is very similar to ours in Article I.

A recent research on TESTAR (de Gier, Kager, de Gouw & Vos, 2019) presented an approach to extract GUI state information into a graph database and analyse the extracted behaviour after the execution by making database queries. With suitable queries, it is possible to analyse complex historical properties involving multiple SUT states. De Gier et al. propose automated oracles to check whether an application follows the rules for the accessibility to allow visually impaired to use it. Some accessibility oracles are checked “online” during the TESTAR execution and others “offline” with database queries after the execution.

2.2.2 RQ2.2: Automated change detection by comparing extracted models of consequent GUI versions

Defining automated test oracles is a main challenge in scriptless (random) testing (Article IX) and the same applies to test generation from extracted models. Most

random testing tools use only generic checks to find exceptions and crashes or getting the GUI into unresponsive state. Most of them do not support manually defined application specific test oracles or regression testing between SUT versions. Some tools, like TESTAR, provide an option to define application specific oracles and instructions for the GUI traversal, but defining them prior to model extraction is a manual task.

In Chapter 1.1.4 of this thesis, we describe how automated change detection addresses the shortcomings of generating test cases from extracted GUI models, illustrated in Fig. 2.

Contributions of this thesis

Article I contemplates the possibility to observe the changes happening in the GUI by comparing two consecutive structural models while automatically interacting with the GUI application. Another possibility would be visually analysing the human readable graphical representation of the extracted GUI model against the requirements of the system. Normally, MBT is about manually modelling the SUT based on the requirements and automatically generating a test suite based on that model to check if the SUT fulfils the requirements. In our approach the generated models are based on the actual implementation, so the generated GUI state model should be compared with the requirements of the system. Also, automatically generated graphical models can help developers to understand and analyse an existing implementation if proper models of the system are missing. Article II states that the ultimate goal for automated GUI modelling could be generating human-readable graphical models containing enough information on a suitable level of abstraction to allow direct comparison to the requirements of the system.

Article V presents an approach using Murphy tools for automated change detection by comparing extracted models of consequent GUI versions In a continuous integration process, the same extraction script can be used to automatically extract a model of the latest development version several times a day, and automatically compare the models and send warnings if changes in the behaviour are detected. With major releases, the model extraction scripts might require minor modifications, such as pointing the script to the correct version of the application and to use the correct and valid product keys. When changes are detected, Murphy provides a web UI for the user, showing the screenshots of both versions for each state having differences, highlighting the changes, and asking the

user if the changes were desired new behaviour or undesired deviations and faulty behaviour.

A large part of the behaviour of the GUI is already tested and directly detectable defects, such as crashes and unhandled exceptions, are found during the model extraction phase. Incorrect behaviour, usually found with conformance testing, can be detected when the extracted model is manually inspected and validated. When the extracted model is inspected and approved, it can be used for automating and supporting various testing activities. New models of the latest versions in the continuous integration process were automatically captured and compared with earlier models 3 times a day, and warnings were sent whenever changes in the behaviour were detected. Then the test engineers used the web UI of Murphy to check each change and decide if the change was intended or an error. A large part of the detected changes were intended, and could be called false positives, but instead of having to update the related test cases, the test engineer just starts using the new model for the automated comparison. The process is similar to regression testing, but actually uses model extraction and model comparison, instead of automatically generating and executing test cases with test oracles based on behaviour of the earlier version.

Related work

To our knowledge, no other research has been published on automated change detection by comparing extracted models of consequent GUI versions. TESTAR is currently being extended with model extraction and the plan is to implement similar functionality.

3 Summary, conclusions and discussion

3.1 Summary

This thesis introduces two approaches and tools for automated extraction of state-based GUI models: GUI Driver and Murphy tools. Both approaches use dynamic analysis during automated exploration of the GUI and similar methods for GUI state abstraction. With both tools, the models were used for test case generation and regression testing. In addition, Murphy tools support automated change detection based on model comparison.

GUI Driver was implemented in Java, based on Jemmy Java library for analysing the state of the GUI (Article I). The primary criterion for defining GUI states was the available actions on that state (Article I), and the state abstraction was further improved by associating the data values with state transitions, instead of GUI states (Article II). To reach all parts of the GUI, GUI Driver provides a semi-automated and iterative process of automated exploration and manually providing valid input (Article II), and a widget classification that allows implementing improved action selection algorithms (Article III).

GUI Driver is able to export the state model into GraphML file format, supported by GraphWalker test sequence generation tool. Test sequences generated by GraphWalker can be executed with GUI Driver, testing the conformance to the modelled behaviour. Also, the system is being tested for robustness during the model extraction process, and for example system crashes and unhandled exceptions can be automatically detected. In Chapter 1.1.2, we described the challenges and shortcomings of generating test cases from extracted GUI models.

GUI Driver tool was evaluated by modelling and testing open source Java GUI applications. The options for suitable SUTs were limited, and we followed the examples of existing research on GUITAR tool by Memon's research group.

Murphy tools (Articles IV and V) were implemented in Python. Murphy has multiple options for analysing the state of the GUI, for example using Windows accessibility API or image recognition, and provides a web based user interface (UI) for using the tool. Murphy tools include a remote execution component, allowing the execution of GUI testing and model extraction in virtual machines. In a similar way to GUI Driver, Murphy tools abstracts the data values from GUI states to limit the number of states in the model. To reach all parts of the GUI, Murphy allows the user to define SUT-specific instructions, triggering pre-defined behaviour when

specified conditions are fulfilled. To restrict the modelling to stay on the interesting part of the GUI, Murphy allows defining boundary nodes in the SUT-specific instructions. Although not completely generic, using image recognition for extracting GUI state information was less dependent on a specific operating system or execution platform, but also less accurate. For validating the correctness of the extracted GUI models, Murphy tools visualize the GUI state model into a graph with screenshots of the GUI as nodes, allowing the user to visually analyse the modelled behaviour.

Murphy tools provide an interactive web UI with visualisation of the GUI state model that can be used to define specific paths with specific inputs and generate test cases accordingly. This way, the extracted GUI state model is used as an interface for help the user in defining and generating regression test cases. Murphy supports automated change detection by comparing extracted models of consequent system versions. In Chapter 1.1.4, we described how the automated change detection addresses the challenges and shortcoming of generating test cases from extracted GUI state models.

Automated change detection was evaluated in industrial software development environment at F-Secure Ltd, using continuous integration to extract a new model from the latest SUT version three times a day. The detected changes were reported by email and visualized with screenshots of changed states on both versions shown on the web UI of Murphy tools. Unfortunately, not all the details could be published due to confidential information, but the results from the evaluation were very positive and promising.

3.2 Conclusions

Based on our results, we claim that dynamic analysis during automated GUI exploration can and should be used for automatically extracting state-based models through GUI of the application. The system is already being tested for robustness during the model extraction, and the extracted models can be used in various ways to automate software testing.

We have shown that the state space explosion can be prevented by abstracting the GUI states, and our results show that associating data values with state transitions instead of states is an efficient way to reduce the number of states. Using available GUI actions as the first criterion for defining GUI states is a good way to ensure that the created models can be traversed also after GUI state abstraction. If two states would be considered the same even if they do not have the same available

GUI actions, using the model for generating test sequences could produce sequences that cannot be executed. This is one of the main shortcomings of event-based models.

We have also shown that a semi-automated process, including the user in the loop during model extraction, has added value. Providing valid input values directly into the GUI that is being modelled, and saving the input into the model, is an easy way for the user to teach the tool how to reach all parts of the GUI. The user has to monitor the model extraction during the teaching, but after that the model can be used for automated testing. However, for the users with adequate programming skills, defining SUT-specific instructions for the tool gives more options and better control over the model extraction process. Also, the SUT-specific instructions support restricting the GUI exploration of the model extraction tool. This is especially important with web applications, so that the tool does not end up modelling the whole Internet.

Action selection algorithms have a major impact on how quickly the automated GUI exploration covers all parts of the GUI. Our widget classification allows defining more detailed action selection algorithms, for example, first executing all GUI controls may result error dialogs about empty input fields. Executing actions on GUI options and GUI inputs, and then re-executing GUI controls with the new data values may result different state transitions.

Although our ultimate goal would be to provide a generic way to extract models of application running on any OS, platform or device, that remains to be future work. The most generic solution appears to be using image recognition to extract the widget information from the screenshots of the GUI. However, an API-based extraction, for example using Windows accessibility API, produces more detailed and more accurate information about the widgets shown on the GUI. Probably the best solution is to use API-based model extraction if possible, and use image recognition as a backup if a suitable API is not available.

Our experiences show that using screenshots to visualize the state model is a feasible solution for visually validating the correctness of the extracted models, as long as the models remain small enough for the user to understand them. Dividing the models into multiple smaller ones, for example using SUT-specific boundaries, is a useful way for keeping the models small.

Although extracted models can be used for model-based test case generation, our results show that automated change detection might be a better option for supporting regression testing. As explained in Chapters 1.1.2 and 1.1.4, there are challenges in automatically generating test cases from extracted models, but

automated change detection by comparing extracted models of consequent versions overcomes some of those challenges.

Using a visualized graph as a UI for the user to define and generate test cases was evaluated to reduce the amount of regression test code that has to be maintained. The web UI of Murphy tools was used also for automatically execute the GUI to a specific state, to reduce the time required for executing manual GUI test cases.

In our opinion, automated GUI exploration and model extraction requires changing the attitude towards testing. Using extracted models to automate testing does not fit into traditional definition of functional testing, because a direct link to functional requirements is missing. The approach is closer to robustness testing and should be thought and used in addition to more traditional functional test automation. Currently, efficiently using the tools requires some programming skills, although in the future the tools will probably get easier to use.

3.3 Future work

The source code of GUI Driver tool has not been published open source nor been maintained in the recent years. The source code of Murphy tools is open source and available in GitHub, but has not been maintained in the recent years. Currently, our development efforts are directed to enhancing and improving open source TESTAR tool.

Currently, TESTAR is being extended with functionality to extract state-based GUI models, and in the near future, we plan to implement support for automated change detection. Automated change detection will be evaluated in software development environment of various industrial companies.

To implement more generic way to analyse the GUI, we aim to use TESTAR to produce training data through Windows accessibility API, and use machine learning (ML) with image recognition to improve the accuracy of widget detection from screenshots.

We have ongoing research on various ways of using artificial intelligence (AI) and ML for more intelligent action selection. However, the current metrics for comparing action selection strategies are quite rudimentary, and there is room for improved metrics. Also, ML requires good metrics to guide the learning process.

When using random action selection, GUI testing is easy to execute in parallel, for example using multiple virtual machines. However, the more intelligent action selection algorithms require some method of communication between them to benefit from parallel execution. We plan to define and implement a generic

platform for remote and parallel GUI testing that would also support more intelligent action selection and model extraction.

Another topic for future research is automated test oracles for TESTAR. One direction is using the extracted state models for mining potential temporal oracles that the user has to validate before they are used in testing.

3.4 Threats to validity

Evaluating the Murphy tools in the development environment of only one company could be a threat to the validity, but we believe the approach can be applied more generally in development and testing of software with GUI.

Also, evaluating GUI Driver only with open source Java applications could be a threat to the validity, but we believe the approach would work for other Java applications as well.

The approaches researched in this thesis are currently being implemented into open source TESTAR tool, and the plan is to continue evaluating the approaches in industrial software development environments.

List of references

Alégroth, E., Feldt, R., & Kolström, P. (2016). Maintenance of automated test suites in industry: An empirical study on Visual GUI Testing. *Information and Software Technology, 73*, 66-80. doi:10.1016/j.infsof.2016.01.012

Alshahwan, N., Gao, X., Harman, M., Jia, Y., Mao, K., Mols, A. … Zorin, I. (2018). Deploying Search Based Software Engineering with Sapienz at Facebook. *International Symposium on Search Based Software Engineering (SSBSE 2018)*, 3-45. 8-9 Sep 2018, Montpellier, France.

Amalfitano, D., Fasolino, A.R., Carmine, S., Memon, A.M., & Tramontana, P. (2012). Using GUI Ripping for Automated Testing of Android Applications. *Proc. 27th IEEE International Conference on Automated Software Engineering (ASE'12)*, 3-7 Sep 2012, Essen, Germany, 258-261.

Amalfitano, D., Fasolino, A. R., Polcaro, A., & Tramontana, P. (2013). The DynaRIA tool for the comprehension of Ajax web applications by dynamic analysis. *Innovations in Systems and Software Engineering*, Apr 2013, Springer-Verlag.

Amalfitano, D., Fasolino, A.R., Tramontana, P., & Amatucci, N. (2013). Considering Context Events in Event-Based Testing of Mobile Applications. *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops (ICSTW).* 18-22 Mar 2013, Luxembourg, 126-133.

Bainczyk, A., Schieweck, A., Isberner, M., Margaria, T., Neubauer, J., & Steffen, B. (2016). ALEX: Mixed-Mode Learning of Web Applications at Ease. *International Symposium on Leveraging Applications of Formal Methods (ISoLA 2016).* 10-14 Oct 2016, Imperial, Corfu, Greece.

Barr, E., Harman, M., McMinn, P., Shahbaz, M., & Yoo, S. (2015). The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering, 41:5* (May 2015), IEEE Computer Society.

Belli, F. (2001). Finite-state testing and analysis of graphical user interfaces. *12th International Symposium on Software Reliability Engineering (ISSRE'01)*. 27-30 Nov 2001, Hong Kong, China, 34-43.

Bertolino, A., Polini, A., Inverardi, P., & Muccini, H. (2004). Towards Anti-Model-based Testing. *International Conference of Dependable Systems and Networks (DNS 2004).* 28 Jun - 1 Jul 2004, Florence, Italy.

Böhme, M. & Paul, S. (2016). A Probabilistic Analysis of the Efficiency of Automated Software Testing. *IEEE Transactions on Software Engineering (TSE), 42:4*, Apr 2016 (published 5 Oct 2015), 345–360.

Cheng, L., Chang, J., Yang, Z., & Wang, C. (2016). GUICat: GUI Testing as a Service. *31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2018)*, 3-7 Sep 2016, Singapore.

Chinnapongse, V., Lee, I., Sokolsky, O., Wang, S., & Jones, P.L. (2009). Model-Based Testing of GUI-Driven Applications. *IFIP International Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS 2009).* 16-18 Nov 2009, Newport Beach, CA, USA.

Esparcia-Alcazar, A., Almenar, F., Martinez, M., Rueda, U., & Vos, T. (2016). Q-learning strategies for action selection in the TESTAR automated testing tool. *6th International Conference on Metaheuristics and Nature Inspired Computing (META'2016),* Marrakech, Morocco.

Esparcia-Alcázar, A., Almenar, F., Vos, T., & Rueda, U. (2018). Using genetic programming to evolve action selection rules in traversal-based automated software testing: results obtained with the TESTAR tool. *Memetic Computing, 10:3*, 257-265.

Fraser, G. & Arcuri, A. (2014). A Large Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24:2, 8, 2014.

Gao, Z., Chen, Z., Zou, Y., & Memon, A.M. (2016). SITAR: GUI Test Script Repair. *IEEE Transactions on Software Engineering (TSE), 42*:2, Feb 2016 (published 20 Aug 2015), 170-186. doi:10.1109/TSE.2015.2454510

de Gier, F., Kager, D., de Gouw S., & Vos, T. (2019). Offline oracles for accessibility evaluation with the TESTAR tool. *IEEE 13th International Conference on Research Challenges in Information Science (RCIS 2019)*, 29-31 May 2019, Brussels, Belgium.

Grilo, A.M.P., Paiva, A.C.R., & Faria, J.P. (2010). Reverse engineering of GUI models for testing. *2010 5th Iberian Conference on Information Systems and Technologies (CISTI)*, 16-19 Jun 2010, Santiago de Compostela, Spain, 1-6.

He, Z. & Bai, C. (2015). GUI Test Case Prioritization by State-Coverage Criterion. *IEEE/ACM 10th International Workshop on Automation of Software Test (AST 2015).* 23-24 May 2015, Florence, Italy.

IEEE (1990). IEEE Standard Glossary of Software Engineering Terminology, *lEEE Std 610.12-1990*.

Kull, A. (2012). Automatic GUI Model Generation: State of the Art. *Proc. 2012 IEEE 23rd International Symposium on Software Reliability Engineering Workshops (ISSREW).* 27-30 Nov 2012, Dallas, TX, USA, 207-212.

van der Laan, H. (2017). Automatically Generating Learning Setups for GUI-based Programs through Annotation Processing. *27th Twente Student Conference on IT.* 7 Jul 2017, Enschede, The Netherlands.

Mao, K., Harman, M., & Jia, Y. (2016). Sapienz: Multi-objective Automated Testing for Android Applications. *25th International Symposium on Software Testing and Analysis (ISSTA 2016)*, 94-105. 18-20 Jul 2016, Saarbrücken, Germany.

Mariani, L., Pezzè, M., Riganelli, O., & Santoro, M. (2012). AutoBlackTest: Automatic Black-Box Testing of Interactive Applications. *IEEE 5th International Conference on Software Testing, Verification and Validation (ICST 2012)*. 17-21 Apr 2012, Montreal, QC, Canada.

Mariani, L., Pezzè, M., & Zuddas, D. (2018). Augusto: Exploiting Popular Functionalities for the Generation of Semantic GUI Tests with Oracles. *40th International Conference on Software Engineering (ICSE '18)*, 280-290. 27 May – 3 Jun 2018, Gothenburg, Sweden.

Martínez, M., Esparcia-Alcázar, A.I., Vos, T.E.J., Aho, P., & Fons i Cors, J. (2018). Towards Automated Testing of the Internet of Things: Results Obtained with the TESTAR Tool. *International Symposium on Leveraging Applications of Formal Methods (ISoLA 2018)*. 30 Oct – 13 Nov 2018, Limassol, Cyprus, 375-385.

Matthews-King, A. (2016). GPs told to review patients at risk as IT error miscalculates CV score in thousands. *Pulse magazine, 11 May 2016.* http://www.pulsetoday.co.uk/your-practice/practice-topics/it/gps-told-to-review-patients-at-risk-as-it-error-miscalculates-cv-score-in-thousands/20031807.article (accessed online 8 May 2019)

Meinke, K. & Walkinshaw, N. (2012). Model-Based Testing and Model Inference. *5th International Symposium on Leveraging Applications of Formal Methods (ISOLA 2012)*, 15-18 Oct 2012, Heraklion, Crete, Greece, 440-443.

Memon, A.M. (2002). GUI Testing: Pitfalls and Process. *Computer, 35*:8 (Aug 2002), 87-88, IEEE Computer Society.

Memon, A.M., Banerjee, I., & Nagarajan, A. (2003a). GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. *10th Working Conference on Reverse Engineering (WCRE 2003)*. 13-16 Nov 2003, Victoria, British Columbia, Canada. doi:10.1109/WCRE.2003.1287256

Memon, A.M., Banerjee, I., & Nagarajan, A. (2003b). What Test Oracle Should I Use for Effective GUI Testing? *18th IEEE International Conference on Automated Software Engineering (ASE).* 6-10 Oct 2003, Montreal, Canada, 164-173.

Memon, A.M., Banerjee, I., Nguyen, B., & Robbins, B. (2013). The First Decade of GUI Ripping: Extensions, Applications, and Broader Impacts. *20th Working Conference on Reverse Engineering (WCRE)*, 14-17 Oct 2013, Koblenz, Germany, 11-20.

Mesbah, A., van Deursen, A., & Lenselink, S. (2012). Crawling Ajax-based Web Applications through Dynamic Analysis of User Interface State Changes. *ACM Transactions on the Web (TWEB), 6:1* (Mar 2012), 3, ACM New York, NY, USA.

Miao, Y. & Yang, X. (2010). An FSM based GUI test automation model. *2010 11th International Conference on Control, Automation, Robotics & Vision (ICARCV).* 7-10 Dec 2010, Singapore, 120-126.

Moreira, R., Paiva, A., Nabuco, M., & Memon, A. (2017). Pattern-based GUI testing: Bridging the gap between design and quality assurance. *Journal of Software: Testing, Verification and Reliability (STVR), 27:3* (May 2017).

Morgado, I., Paiva, A., & Faria, J. (2012). Dynamic Reverse Engineering of Graphical User Interfaces. *International Journal on Advances in Software, 5:3-4*, 2012, 224-246, IARIA.

National Transportation Safety Board (2017). *Railroad Accident Brief 1603, Undesired train acceleration and deceleration*, published 4 Apr 2017, https://www.ntsb.gov/investigations/AccidentReports/Reports/RAB1603.pdf (accessed online 8 May 2019)

Paiva, A.C.R., Faria, J.C.P., & Mendes, P. (2007). Reverse Engineered Formal Models for GUI Testing. *12th International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, 1-2 Jul 2007, Berlin, Germany, 218-233.

Pezzè, M., Rondena, P., & Zuddas, D. (2018). Automatic GUI Testing of Desktop Applications: an Empirical Assessment of the State of the Art. *International Workshop on User Interface Test Automation and Testing Techniques for Event Based Software (INTUITESTBEDS 2018), ISSTA/ECOOP 2018 Workshops*, 54-62. 16-21 Jul 2018, Amsterdam, Netherlands.

Rueda, U., Esparcia-Alcazar, A.I., & Vos, T.E.J. (2016). Visualization of automated test results obtained by the TESTAR tool. *XIX Ibero-American Conference on Software Engineering (CIBSE 2016).*

Salva, S. & Zafimiharisoa, S. (2014). Model Reverse-engineering of Mobile Applications with Exploration Strategies. *9th International Conference on Software Engineering Advances (ICSEA 2014).* 12-16 Oct 2014, Nice, France.

Silva, J.L., Campos, J.C., & Paiva, A.C.R. (2007). Model-based User Interface Testing with Spec Explorer and ConcurTaskTrees. *2nd International Workshop on Formal Methods for Interactive Systems (FMIS 2007)*, 4 Sep 2007, Lancaster, UK, 77-93.

Stocco, A., Yandrapally, R., & Mesbah, A. (2018). Visual Web Test Repair. *26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018).* 4-9 Nov 2018, Lake Buena Vista, FL, USA, 503-514.

Strecker, J. & Memon, A.M. (2008). Relationships between test suites, faults, and fault detection in GUI testing. *1st International Conference on Software Testing, Verification, and Validation (ICST).* 9-11 Apr 2008, Lillehammer, Norway, 12-21.

Strecker, J. & Memon, A.M. (2012). Accounting for Defect Characteristics in Evaluations of Testing Techniques. *ACM Transactions on Software Engineering and Methodology (TOSEM), 21*:3 (Jun 2012), 17, ACM New York, NY, USA.

Tramontana, P., Amalfitano, D., Amatucci, N., & Fasolino, A.R. (2018). Automated functional testing of mobile applications: a systematic mapping study. *Software Quality Journal,* Oct 2018, Springer Nature, 1-53.

Tricentis (2018). Software Fail Watch, 5th Edition, https://www.tricentis.com/resources/software-fail-watch-5th-edition/ (accessed online 8 May 2019)

Vos, T.E.J., Kruse, P.M., Condori-Fernández, N., Bauersfeld, S., & Wegener, J. (2015). TESTAR: Tool Support for Test Automation at the User Interface Level. *International Journal of Information System Modeling and Design (IJISMD), 6:3*, 2015, 46-83.

Yang, X. (2011). Graphic User Interface Modelling and Testing Automation. *PhD thesis*, School of Engineering and Science, Victoria University, Melbourne, Australia, May 2011.

Yuan, X., Cohen, M. & Memon, A.M. (2011). GUI interaction testing: Incorporating event context. *IEEE Transactions on Software Engineering*, *37:4* (Jul-Aug 2011), 559-574, IEEE Computer Society.

Yuan, X. & Memon, A.M. (2010). Generating event sequence-based test cases using GUI runtime state feedback. *IEEE Transactions on Software Engineering, 36*:1 (Jan-Feb 2010), 81-95, IEEE Computer Society.

Original publications

I Aho, P., Menz, N., Räty, T., & Schieferdecker, I. (2011). Automated Java GUI modeling for model-based testing purposes. *Proceedings of the 2011 Eighth International Conference on Information Technology: New Generations (ITNG)*, 268–273. 11-13 Apr 2011, Las Vegas, USA. doi:10.1109/ITNG.2011.54

II Aho, P., Menz, N., & Räty, T. (2011). Enhancing generated Java GUI models with valid test data. *Proceedings of the 2011 IEEE Conference on Open Systems (ICOS)*, 310–315. 25-28 Sep 2011, Langkawi, Malaysia. doi:10.1109/ICOS.2011.6079253

III Aho, P., Räty, T., & Menz, N. (2013). Dynamic reverse engineering of GUI models for testing. *Proceedings of the 2013 International Conference on Control, Decision and Information Technologies (CoDIT)*, 441–447. 6-8 May 2013, Hammamet, Tunisia. doi:10.1109/CoDIT.2013.6689585

IV Aho, P., Suarez, M., Kanstrén, T., & Memon, A.M. (2013). Industrial adoption of automatically extracted GUI models for testing. *Proceedings of the 3rd International Workshop on Experiences and Empirical Studies in Software Modeling (EESSMod)*, 1078:49–54. 1 Oct 2013, Miami, Florida, USA.

V Aho, P., Suarez, M., Kanstrén, T., & Memon A.M. (2014). Murphy tools: Utilizing extracted GUI models for industrial software testing. *Proceedings of the 2014 IEEE 7th International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 343–348. 2014. doi:10.1109/ICSTW.2014.39

VI Aho, P., Kanstrén T., Räty, T., & Röning, J. (2014). Automated extraction of GUI models for testing. *Advances in Computers, 95:2*, 49–112. Academic Press, Elsevier, 2014.

VII Aho, P., Suarez, M., Memon, A.M., & Kanstrén, T. (2015). Making GUI testing practical: Bridging the gaps. *Proceedings of the 12th International Conference on Information Technology: New Generations (ITNG 2015)*, 439–444, 13-15 Apr 2015, Las Vegas, USA. doi: 10.1109/ITNG.2015.77

VIII Aho, P., Alégroth, E., Oliveira, R., & Vos, T. (2016). Evolution of automated regression testing of software systems through the graphical user interface. *Proceedings of the First International Conference on Advances in Computation, Communications and Services (ACCSE)*, 16–21, 22-26 May 2016, Valencia, Spain.

IX Aho, P. & Vos, T. (2018). Challenges in automated testing through graphical user interface. *Proceedings of the 2018 IEEE International Conference on Software Testing Verification and Validation Workshop, (ICSTW)*, 118–121, 9 Apr 2018, Västerås, Sweden. doi: 10.1109/ICSTW.2018.00038

Original publications are not included in the electronic version of the dissertation.

705. Karvinen, Tuulikki (2019) Ultra high consistency forming

706. Nguyen, Kien-Giang (2019) Energy-efficient transmission strategies for multiantenna systems

707. Visuri, Aku (2019) Wear-IT : implications of mobile & wearable technologies to human attention and interruptibility

708. Shahabuddin, Shahriar (2019) MIMO detection and precoding architectures

709. Lappi, Teemu (2019) Digitalizing Finland : governance of government ICT projects

710. Pitkänen, Olli (2019) On-device synthesis of customized carbon nanotube structures

711. Vielma, Tuomas (2019) Thermodynamic properties of concentrated zinc bearing solutions

712. Ramasetti, Eshwar Kumar (2019) Modelling of open-eye formation and mixing phenomena in a gas-stirred ladle for different operating parameters

713. Javaheri, Vahid (2019) Design, thermomechanical processing and induction hardening of a new medium-carbon steel microalloyed with niobium

714. Hautala, Ilkka (2019) From dataflow models to energy efficient application specific processors

715. Ruokamo, Simo (2019) Single shared model approach for building information modelling

716. Isohookana, Matti (2019) Taistelunkestävä hajaspektritietovuo kansalliseen sotilasilmailuun

717. Joseph, Nina (2019) $CuMoO_4$: A microwave dielectric and thermochromic ceramic with ultra-low fabrication temperature

718. Kühnlenz, Florian (2019) Analyzing flexible demand in smart grids

719. Sun, Jia (2019) Speeding up the settling of switched-capacitor amplifier blocks in analog-to-digital converters

720. Lähetkangas, Kalle (2019) Special applications and spectrum sharing with LSA

721. Kiventerä, Jenni (2019) Stabilization of sulphidic mine tailings by different treatment methods : Heavy metals and sulphate immobilization

722. Fylakis, Angelos (2019) Data hiding algorithms for healthcare applications

Book orders:
Granum: Virtual book store
http://granum.uta.fi/granum/

UNIVERSITY OF OULU P.O. Box 8000 FI-90014 UNIVERSITY OF OULU FINLAND

ACTA UNIVERSITATIS OULUENSIS

UNIVERSITY of OULU
OULUN YLIOPISTO

ISBN 978-952-62-2405-3 (Paperback)
ISBN 978-952-62-2406-0 (PDF)
ISSN 0355-3213 (Print)
ISSN 1796-2226 (Online)