Topic Driven Testing

A dissertation submitted towards the degree
Doctor of Engineering
of the Faculty of Mathematics and Computer Science of
Saarland University

by

Andreas Michael Rau

Saarbrücken, 2020

Abstract

Modern interactive applications offer so many interaction opportunities that automated exploration and testing becomes practically impossible without some domain specific guidance towards relevant functionality. In this dissertation, we present a novel fundamental graphical user interface testing method called *topic-driven testing*. We mine the semantic meaning of interactive elements, guide testing, and identify core functionality of applications. The semantic interpretation is close to human understanding and allows us to learn specifications and transfer knowledge across multiple applications independent of the underlying device, platform, programming language, or technology stack—to the best of our knowledge a unique feature of our technique.

Our tool ATTABOY is able to take an existing Web application test suite say from *Amazon*, execute it on *ebay*, and thus guide testing to relevant core functionality. Tested on different application domains such as eCommerce, news pages, mail clients, it can transfer on average *sixty percent* of the tested application behavior to new apps—without any human intervention. On top of that, topic-driven testing can go with even more vague instructions of how-to descriptions or use-case descriptions. Given an instruction, say “add item to shopping cart”, it tests the specified behavior in an application–both in a browser as well as in mobile apps. It thus improves state-of-the-art UI testing frameworks, creates change resilient UI tests, and lays the foundation for learning, transferring, and enforcing common application behavior. The prototype is up to *five times faster* than existing random testing frameworks and tests functions that are hard to cover by non-trained approaches.

Zusammenfassung

Moderne interaktive Anwendungen bieten so viele Interaktionsmöglichkeiten, dass eine vollständige automatische Exploration und das Testen aller Szenarien praktisch unmöglich ist. Stattdessen muss die Testprozedur auf *relevante* Kernfunktionalität ausgerichtet werden. Diese Arbeit stellt ein neues fundamentales Testprinzip genannt *thematisches Testen* vor, das beliebige Anwendungen über die graphische Oberfläche testet. Wir untersuchen die semantische Bedeutung von interagierbaren Elementen um die Kernfunktionenen von Anwendungen zu identifizieren und entsprechende Tests zu erzeugen. Statt typischen starren Testinstruktionen orientiert sich diese Art von Tests an menschlichen Anwendungsfällen in natürlicher Sprache. Dies erlaubt es, Software Spezifikationen zu erlernen und Wissen von einer Anwendung auf andere zu übertragen unabhängig von der Anwendungsart, der Programmiersprache, dem Testgerät oder der -Plattform. Nach unserem Kenntnisstand ist unser Ansatz der Erste dieser Art.
Wir präsentieren ATTABOY, ein Programm, das eine existierende Testsammlung für eine Webanwendung (z.B. für *Amazon*) nimmt und in einer beliebigen anderen Anwendung (sagen wir *ebay*) ausführt. Dadurch werden Tests für Kernfunktionen generiert. Bei der ersten Ausführung auf Anwendungen aus den Domänen online Shopping, Nachrichtenseiten und eMail, erzeugt der Prototyp *sechzig Prozent* der Tests automatisch. Ohne zusätzlichen manuellen Aufwand. Darüber hinaus interpretiert themengetriebenes Testen auch vage Anweisungen beispielsweise von How-to Anleitungen oder Anwendungsbeschreibungen. Eine Anweisung wie ”Fügen Sie das Produkt in den Warenkorb hinzu” testet das entsprechende Verhalten in der Anwendung. Sowohl im Browser, als auch in einer mobilen Anwendung. Die erzeugten Tests sind robuster und effektiver als vergleichbar erzeugte Tests. Der Prototyp testet die Zielfunktionalität *fünf mal schneller* und testet dabei Funktionen die durch nicht spezialisierte Ansätze kaum zu erreichen sind.

Acknowledgments

For some people a dissertation is just another step in their (academic) career. For others it is the culmination of long lasting research. For me, it is a little of both. It takes years to finish, dedication to follow through with ideas that lead to nothing, and working long nights when a deadline is due. But this is not something you do on your own. Here, I have the opportunity to thank those people who helped me on the way. Even those, who will never read these words. Hopefully, the ones I mention and do not read these words make up for those who I do not praise. Please bear with me. I appreciate all the help I had.
To Andreas Zeller, my supervisor, collaborator and advisor. There are standard ways to thank your professor, but this is not my way of doing things. I want to thank you for giving me the opportunity to follow through on my own research ideas. While for some people academic success is measured in the number of publications and citations, you always encourage people do to novel things. Think out of the box. Do not just do incremental work. It is a risky way of doing things, but brave ideas are the ones that have impact. So thank you for pointing out the overall picture, when I was stuck in the experimental details.
I would like to thank my colleagues and friends from Testfabrik. Valentin Dallmeier, who gave advice and feedback on the initial idea. He taught me a lot in the early years of my PhD. Together with Michael Mirold and Bernd Pohl, I had the chance to get in touch with industry and put a perspective on the differences between academic research and industrial needs. I'd like to thank the students I worked together with in their Bachelor and Master theses. I learned a lot trying to teach you.
A big thanks goes to Sascha Just, my office mate and friend for most of my time as a student. Having lots of experience in academic research, knowledge in almost any area of data processing and empirical evaluation, he gives great feedback and ideas. Having an excellent taste in whisky is also not a disadvantage.
My final thanks go to my family. My parents, my brother, but especially my wife Jenny for her patience and love. A journey, almost a decade long, comes to an end now. A new one will follow.

Table of Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Testing applications has always been an integral part of the software development process. Testing is not only a matter of quality assurance just before the software is finally deployed or shipped to the customer. Instead, it is a repeated process in all levels of development from requirement engineering, design, prototyping, implementation, verification, and maintenance. Testing is further complicated by team structure and product structure. While individuals can test their implementation independent of dependent system parts (i.e. unit testing), integrating different project parts into a final product requires adapted testing strategies (e.g. integration testing).

In the last two decades, *information technology* has gained a major development and disposability boost. In 2018 alone, 4.93 billion mobile devices, more than 2.1 billion personal computers and about 9 billion Internet of Things (IoT) devices are connected to the web worldwide. While these numbers are impressive, the potential growth also impacts the software development process. Software systems are continuously evolving, while more and more competitors publish alternative software solutions and develop markets on other platforms. Essentially, this means that software is available on multiple platforms (e.g. desktop, mobile, etc.), connected to other services (e.g. integrated services), and new features have to be shipped with tremendous speed at low cost. Even sophisticated development techniques like *agile development* combined with *Continuous Integration (CI)* can barely counteract this troublesome pressure.

For developers and product owners, *software quality* encompasses many aspects. While *functionality* is essential, software should also be *maintainable*, *secure*, and *extendable* such that future use case scenarios can be realized. However, even these keywords only define coarse concepts of what is expected of a system. Laymen often naïvely associate software quality with the terms "software without bugs" or even "works as expected". These assessments hardly translate into a formal development goals, but is rather intuitively based on experience in related subjects or domains. Especially these *expectations* explain how users, in contrast to developers, experience a system.

For most users, an application is experienced through the Graphical User Interface

(GUI), which abstracts complex technical events into human readable output comprised of text in a natural language, images, and icons. Typically, the GUI also reflects the *state* of the application, which can be changed by the user by interacting with the user interface (UI) elements like input fields, drop down menus, or the like. Again, technical events (e.g. network events, event handler communication, or data access) are not explicitly observable. Nevertheless, there exist established procedures to notify users about the success/failure of operations with appropriate success/error message and styling/coloring properties. To further ease the use of applications, there are established workflows for using common functions such as logins, buying a product, or writing emails. All these characteristics are required to follow established usability standards. This includes which UI elements should be present, which interactions are required, and what information should be provided. In contrast to developers, who develop a system and its testing procedures from the technical point of view, the user perspective is based on the subjective *experience of certain use cases*. In other words, users interpret the UI by checking for *semantic* concepts and can thus transfer their previously learned experience even if the application has *syntactic* differences.

Accordingly, developers face a serious technical debt while developing an application. Not only do they have to implement each and every use case (i.e. every *feature*) for every application anew, integrate them into the environment, but also have to provide extensive tests to ensure software quality. Imagine you want to *test the shopping procedure* of an application. Figure 1.1 shows how complex this workflow is. Repeatedly testing this workflow manually (e.g. for regression) or developing reproducible test cases requires significant effort for developers. They cannot automatically leverage established common knowledge from other application to speed-up and improve testing, because existing test frameworks are interpreting all instructions syntactically. If the application changes the UI structure, the test case cannot be translated. In addition to that, they cannot simply fall back to random testing. The likelihood that a state-of-the-art random testing technique effectively tests a complex procedure, such as shopping, is negligible. Certain actions have to be executed in a specific order (search for the item, select the correct one from a list, add to cart) and the sheer number of possible interactive UI elements in each page leads to a combinatorial state explosion.

This is where *topic-driven testing* (or short TDT) comes into play. By mining a large set of existing applications, topic-driven testing gains use case specific knowledge, interprets the *semantic concepts* behind every Graphical User Interface (GUI) element, and is able transfer this knowledge to other applications. This dissertation introduces the novel concept of *topic-driven testing* and it makes two major contributions in the field of software testing.

(i) First, it presents a way to automatically detect features in existing applications, by mining its existing system test cases (UI-tests) and randomly crawling their user interfaces. With this knowledge, topic-driven testing learns *which* features should be present in an application of a certain domain, *how* features are accessed by the users, and also *observe* how a software behaves both visibly and technically while these features are executed. On top of that, TDT can also interpret vague step-by-step instructions written in natural language to test explicit application behavior.

(ii) The second major and most noteworthy contribution of this thesis targets the ability to *transfer this knowledge across applications* to guide testing. This development might allow a *shift of paradigms* in the future of software testing and development. Instead of independently developing and testing features always anew from scratch, developers can check if their implementation is at least as good as a competitor's from a user perspective—or at least follows established (by majority vote) standards that are familiar to a user. Given tests can also be executed on multiple applications. By interpreting semantic entities, topic-driven testing (TDT) is robust against syntactic changes, be it reordering of UI elements, different labels, or the like. Finally, TDT allows to relate the application models of two different applications, bringing the capabilities of model checking into the testing scenario. We can learn and transfer state-of-the-art policies from a set of applications on how applications *should* behave and detect outliers—even telling a user if something is *different*.

Reconsidering the motivating example in Figure 1.1 of a shopping workflow, TDT mines a natural language description of the workflow (which serves as an input) and semantically executes the instruction on an arbitrary system under test (SUT). The topic-driven procedure can learn, how a use case is executed and execute this knowledge on any application with a GUI. The input to the presented procedure can vary. During the conducted empirical studies, the input to topic-driven testing has been changed from pre-defined test cases (e.g. SELENIUM test scripts) for web applications as well as more general use case descriptions for Android apps. In other words, the procedure deals with a wide variety of inputs and can be executed on multiple browsers, platforms, or devices. More precisely, it provides cross-platform and cross-browser compatibility.

Of course there exists a multitude of tools—i.e. automated testing procedures—to help developers in their testing effort, which we further discuss in terms of related work in Section 2.3. Basically, we can differentiate between *random testing* approaches and

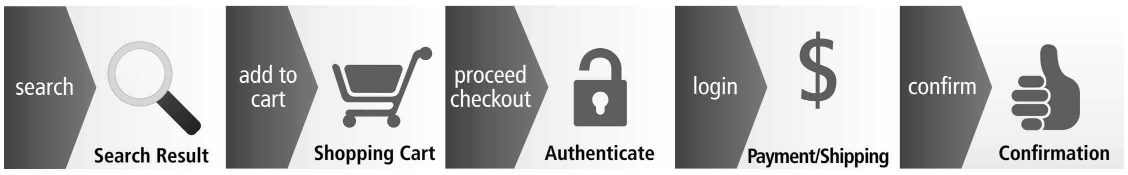


Figure 1.1: Typical workflow for buying a product in an eCommerce application

guided testing approaches. Random testing techniques require virtually no prior knowledge about the SUT and have a low initial setup cost for a developer. In order to test the application, the testing procedure explores all possible input combinations until the exploration space is exhausted. Due to the cheap setup costs, random testing is widely used to get basic application models, or apply penetration testing (especially in the area of fuzz testing, i.e. fuzzing). For real world applications, random testing approaches rarely scale. The state model and the input space for potential GUI actions is too large to explore the application exhaustively. Furthermore, random testing rarely offers more than the most simplistic oracles (i.e. a ground truth for the correct application behavior). Complex reactions on inputs (e.g. error messages or incorrect behavior) require a basic understanding of the SUT which results in higher setup costs. Instead, the testing procedure can check if the application breaks.

The second area of testing approaches are *guided testing* procedures or *model-based testing* procedures. This group is trained with prior knowledge, such as how to interact with the SUT (e.g. with grammars or input languages), and the expected outcome for the inputs (the oracle). The better the provided ground truth and interaction methods are specified, the more realistic the achieved test coverage and error detection are. In this scenario, '*better*' translates to *more effort* for setting up the testing procedure.

Without trivializing the contribution of these works, there also is an indisputable huge *gap between the academic advances and their acceptance in industry*. Many techniques have been adopted into industry to ease methods of testing. Both areas, random and guided tests, discover thousands of software faults and save millions of dollars. Still, there are test areas, say for instance acceptance testing, which are mainly done manually. The reason is quite simple: *expressing testable, i.e. machine executable, specifications from vague descriptions is complex and in all generality does not yield the expected results*. The human language as well as the human perception is often not precise enough to be expressed in hard facts, which developers can reliably use for testing. Instead, human developers close this gap by either manually executing the SUT and test it for bugs or write test cases to automatize this process in the future. This effort is repeated over and over again; for every application, platform, or even software release.

Instead of discussing literally hundreds of contributions and research projects, such as cross-browser, cross-platform, or staging system (e.g. for regression testing) that can partially tackle these challenges, let us take a step back and check what crucial edge human developers have on automatic approaches. Humans can infer and transfer knowledge across application. Based on the prerequisite that humans have an understanding on what a system should do, they can develop tests including an *oracle*. This oracle is a definition of *the correct behavior*. Humans can use their experience from other applications or problems and transfer this experience to the new system. This includes which functions a system should offer, how a system should be used, and how the system should behave. Furthermore, humans can go with vague, sometimes incomplete instructions, such as “buy a knife on Amazon“, because they can semantically interpret them. Instead, machines need precise instructions which are then syntactically executed on a system, which includes an understanding about the involved entities (i.e. knife and amazon) as well as sequence of actions for the action “buy”.

Topic-driven testing serves as the interface to *translate observable application behavior across applications and is further able to interpret imprecise natural language commands*. A semantic interpretation of the displayed texts closes the gap between vague instructions, and testing procedures and machine executable instructions.

1.1 Topic Driven Testing in a Nutshell

In essence, *topic-driven testing* derives for each element presented in the GUI of an application a semantic meaning. It allows to compare application features and functions independent of their underlying programming language, architecture, or platform. It allows us to *learn* the semantics of a SUT by mining the GUI and observing the UI changes which follow interactions, to *transfer knowledge* across application, and enable *new testing principles* that are closer to a human understanding of a software.

Figure 1.2 gives a simplified process overview to summarize topic-driven testing in a nutshell. The idea is pretty simple. Topic-driven testing takes arbitrary use case specifications as an input. We tested our technique on existing web test suites (SELENIUM test scripts) as well as more general *process snippets*, i.e. short natural language descriptions of use cases. In the first case, the given input test suite is first executed on the application they were designed for. During this execution, we observe the UI changes

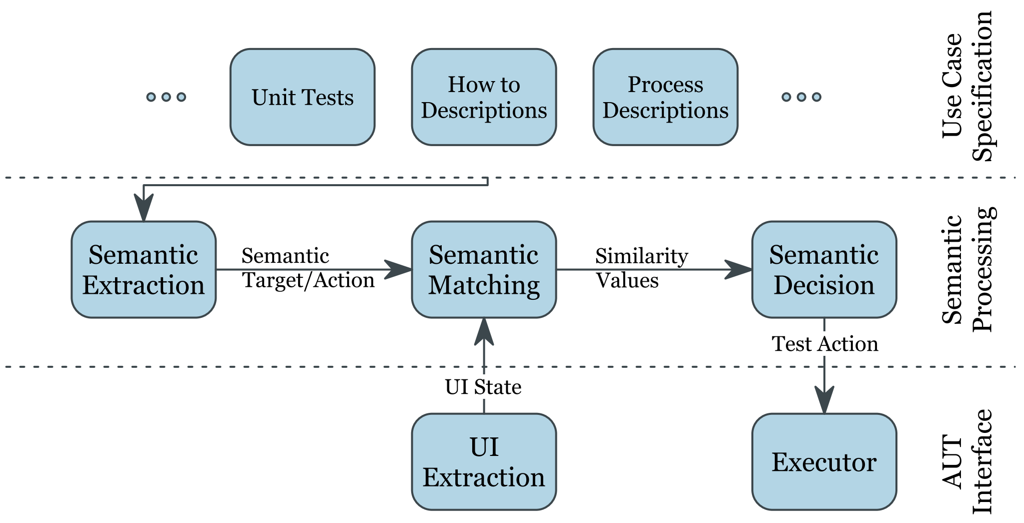


Figure 1.2: Process overview about the new testing principle of *topic-driven testing*. The central part is the application and platform agnostic semantic analysis and interpretation unit, called *Semantic Processing*. The input to the testing procedure (on the top) is interchangeable (test instructions, test cases or application models). The bottom is the execution part to the underlying application under test (AUT) that interprets and executes the generated test instructions and allows to extract the state model.

and targets for test actions to extract the natural language content of the GUI targets. Now, we start the exploration on a new target application and observe the state that is displayed in the GUI ('UI Extraction').

The technical complexity of interacting with the specific device or platform is hidden in the *AUT interface* layer. Topic-driven testing builds on an existing technology stack by integrating the testing procedure into existing approaches to interact with the SUT. As a consequence, the test principles can be applied (and have been tested) on both traditional web applications as well as Android apps. This provides credible evidence that topic-driven testing can be applied on arbitrary applications with a GUI.

The 'UI extraction' interface allows the technique to analyze the UI screen of the test application to bind interactive elements with their describing texts. Now, the central part 'semantic processing' puts those two parts together. The test guidance is given by the provided use case specification and topic-driven testing interprets and matches these to the target application by calculating the semantic similarity. The semantic decision communicates with the executor to send appropriate testing commands to the SUT.

1.1.1 Contributions

In the scope of this dissertation, we developed a prototype implementation for web applications as well as Android apps, and conducted a series of empirical evaluation to show the effectiveness of topic-driven testing. All results and prototype implementation are made publicly available to foster external inspection, replication, and further development of the techniques.

The core contribution of this dissertation is a *novel testing technique/concept which tests a system through the GUI based on semantic properties*. So far, TDT has been tested and executed on dozens of real world industry-sized applications both in the domain of web application as well as Android apps. In both cases, the test applications origin from different application domains to show the generality of the technique. The empirical studies provide credible evidence that we can use topic-driven testing can be used to transfer existing test suites to new applications with only minor effort. The analysis shows that on average *more than sixty percent of the test cases can be automatically transferred* and that we can guide a web crawler towards relevant functions *seven times faster* than a random approach. The presented technique is effective in identifying application specific functions. Derived from the test cases, topic-driven testing *identifies and matches 75% of the tested functions with a high precision of 83%*.

Furthermore, topic-driven testing allows to mine simple natural language instructions (called *process snippets*). Defining process snippets is a one time effort and can even be automatized by mining existing how to instructions [41]. The semantic interpretation of these instructions (and their translation into machine executable statements) is precise. Compared to random testing, the integration of test guidance based on the semantic interpretation of process snippets *speeds up the exploration process by a factor of five*, while *discovering twice the amount of core functions* in this time. The main test advantage is gained in the first few minutes of the exploration. Thus, the technique can be further integrated with other testing techniques.

Summarized, this dissertation presents the following contributions to the field of software testing and analysis:

Novel Testing Technique

Topic-driven testing offers a novel testing angle for testing the GUI that is complementary to existing testing. By mining semantic structures we can learn how humans interact with the SUT and *automatically generate tests* that are centered around use cases rather then syntactic properties. This property makes the generated tests robust against syntactic changes that are typical for evolving software systems (i.e. robustness for regression testing [42]).

The central contribution is *another step into full test automation* that is superior to random testing. Testing can be steered towards relevant core functions and areas of interest by converging the exploration towards semantically similar areas. The definition of semantic instructions is a matter of mere minutes, but the presented results indicate that they are executable on dozens of applications. The implications are obvious: with low effort, a tester can test a wide variety of applications. The conducted empirical studies give credible evidence that the tests can be executed on hundreds of applications without significant additional (manual) effort and allow to observe their behavior. This is especially valuable for further academic research since they generate reproducible test traces targeted to specific use cases.

The testing technique is independent on the underlying platform and can be executed on arbitrary GUI applications. While a cross-browser and cross-platform compatibility is also achieved by other state-of-the-art frameworks, the ability to additionally run it on multiple different applications at once is a unique feature.

Feature Identification

To the best of my knowledge, no-one before matched features across applications independent of the underlying application platform. Topic-driven testing reliably identifies 75% of the features in our test set with a precision of 85%.

Detecting specific functionality, i.e. features and concepts in source code, is an essential part of program comprehension [63]. We already discussed that the user view on the definition of functionality is different than the definition from a developers point of view. In practice, users reduce functionality down to interactive UI elements and a certain procedure. A basic authentication function is thus build on inserting a username/password combination into the appropriate input fields followed by pressing the proceed button. One of the contributions of this work is to mine the GUI for such features by executing a combination of system tests together with random exploration. This lays the foundation to extract semantic properties of the natural language content describing the interactive elements. Furthermore, it allows to learn, which features are realized in an application.

The presence, absence, and location of these features in the SUT can be automatically verified and tested by learning features and transferring this knowledge to new applications. This procedure also allows to test general application specifications such as the system behavior and the reactions of the SUT on certain actions.

Automated Model Translation

Even though web applications are accessible through a more or less stable browser interface, they naturally have differences in layout, styling, structure, and workflow. These *syntactic differences* have so far rendered automated feature matching approaches useless. Existing techniques, e.g. for cross-platform feature matching [72] or methods based on string similarity [90], are insufficient to translate application models. In order to learn standard behavior, this thesis presents a newly developed matching and translation strategy based on semantic text similarity [30, 55]. The same holds for mobile apps as well.

Topic-driven testing allows to learn common behavior from a wide variety of applications and lays a foundation for further research both in an academic as well as in an industrial setting. The provided mapping between two applications allows to transfer knowledge and compare common application behavior based on their workflows. Potential further research could analyze for every use case under test the acquired resources, the access to sensitive data, or perform a security analysis. Automatic outlier detection and behavior analysis was part of the conducted research [7] and shows initial evidence that the correct access to sensitive resources (guarded by the API access) is bound to the description of interactive elements or rather their meaning.

Baseline for Further Testing Methods

The core contribution of this dissertation is the novel testing principle topic-driven testing. The conducted empirical studies in the later sections show its versatility and value to automatize GUI testing. Time and scope of the dissertation limit the possible extent of empirical evaluation. Despite that, I would like to discuss further possibilities that lay in the scope of the presented testing technique.

It is reasonable to extend the prototype to further automatize *acceptance testing*. The conducted studies show the capabilities to work with natural language descriptions (use case descriptions) to test a SUT. This is the first half of typical acceptance testing that is conducted by human testers and so far cannot be automatized. The other half of acceptance testing is the specification of the expected behavior of the application when certain steps are executed. These specification include the presence, styling, and position of the UI and its elements. On top of that, the workflow of the application is part of the use case specifications as well. Our data *indicates* that topic-driven testing can be useful or be extended to perform this task. In order to *prove* the applicability of acceptance testing it is mandatory to perform additional user studies with real testers.

The same can be said about *usability testing*. The IEEE standard computer dictionary [25] literally describes usability as '*the ease with which a user can operate, prepare inputs for, and interpret outputs of a system or component*'. Nielsen [58] further break usability down to five main attributes: *learnability*, *efficiency*, *memorability*, *error proneness*, and *user satisfaction*. *Usability* is a fundamental concern for every end-user application. As a consequence, learnability and memorability are taken into consideration to ensure users do know how to complete a task. The textual content or replacing icons presented to the user must allow for a natural understanding of the underlying task. Existing testing frameworks [6, 32] tackle some of these main attributes, but also require extensive specifications in the first place. The ability to transfer and learn transfer these specifications allows a huge benefit for testing. An integration with machine learning frameworks to observe common usability patterns in a majority of applications might be helpful. TDT lays the groundwork for such a method by introducing the capability to compare application behavior.

1.1.2 Assumptions and Limitations

Every testing technique and empirical evaluation follows certain assumptions and limitations that originate from time and resource constraints for conducting the empirical evaluation as well as technical limitations. We already discussed the potential benefits of topic-driven testing in general. Before delving into the the technical details and discuss the conducted empirical studies, I would like to sketch the limitation and assumptions made in this dissertation. This readies us to judge the benefits openly.

Topic-driven testing is a model-based dynamic testing technique that analyzes the client-side GUI to guide testing. The application itself, especially the server-side backend code, is treated as a black-box. Therefore, it suffers from technical limitations that are also bound to the underlying technology stacks (i.e. the AUT interface).

First of all, changes to the SUT are only observed at discrete points in time, say for example every second. Second, the UI extraction is not instantaneous. Remembering the process overview in Figure 1.2, the UI analysis part that retrieves the application state in the GUI is integrated into an existing technology stack. Between the individual discreet points of extraction the testing method can miss certain application behavior. This lack of precision cannot be prevented beyond a certain point. Network latencies, device utilization, and device load prevent completely deterministic results. To prevent or at least limit bias in the individual empirical studies, all experiments were repeated ten times (cross validation) and the results shown in the dissertation are averaged.

Second, our testing only detects changes that are observable through the GUI, but does not keep track of internal changes of the program state outside the UI—again a side-effect of treating the application as a black-box. Server-side errors that occur in the back-end, but are not propagated to the GUI or do not manifest in visible UI changes, are not transparent to our technique. At this moment, this cannot be changed. I think the proposed method is orthogonal to existing testing approaches and further research might tackle this limitation. Giving the testing method insight into the application itself in order to satisfy additional requirements is still possible. This dissertation limits itself by considering every application as a *black-box* and can thus focus on showing the generality of the proposed technique.

Due to the dynamic nature of the proposed testing technique, the analysis may be *sound*, but not *complete*. Imagine we start a testing procedure for a web application at a certain web page and even manage to explore the complete input space. There could still exist a certain part of the application that is not reachable from this start page. Its main limitation is that all studies never completely cover all application behavior or the complete input space. This is also not the intended contribution of this dissertation, since a complete testing procedure would not scale on real world applications. Nevertheless, it puts a certain perspective on all discussed results. The drawn conclusions might not be generalizable on the complete application behavior. Since the *coverage* of the testing procedure cannot be measured in a black-box application, the evaluation cannot simply measure the impact as well. At this point, we assume that the application behavior does not simply change for certain system parts. Even the unexplored parts of the GUI are intended to be used by human users. As a consequence, they should follow the same design principles as the explored application parts. Still, this assumption is discussed as a potential threat to validity of the technique.

A final, maybe negligible limitation, is the language of the applications under test. All experiments presented in this dissertation are executed on applications with a GUI in *English* using a pre-calculated word vector model—the word2vec-GoogleNews-vector (googleNewsVector) [29]. While it is possible to train such models for other languages (or use other off-the-shelf models), the used model was trained on three million words and phrases. The effort to transfer the technique to other language is manageable, though. The size and diversity of this corpus might positively influence our results. Moreover, the technique cannot be executed on multiple language inputs, i.e. multilingual apps, at once without further steps. Since the semantic meaning might also suffer from cultural influence, additional analysis might be necessary. At this point, the empirical evaluation set is limited to appropriate applications.

1.2 Thesis Structure

Chapter 2 provides a short introduction into automated UI testing techniques for both Android apps and web applications, related definitions, and related work. After presenting these concepts and covering the most common information retrieval frameworks, we will see a brief overview on state of the art natural language analysis methods.

Chapter 3 presents how interactive elements are bound to their describing labels, i.e. how topic-driven testing infers the semantic properties of UI elements. It includes how page segmentation is used to group elements with their description and details about the noise reduction, which is employed on the textual description for the semantic analysis. This lays the basic for topic-driven testing in general.

Chapter 4 follows up with a discussion on how we identify *features* in the GUI. Our investigation on five hundred top web applications shows that features can be grouped by their semantic similarity. This knowledge allows to re-identify features (i.e. map features) across applications. The empirical studies indicate precision rates of about 85% for identifying features across web applications.

Chapter 5 focuses on how topic-driven testing can be used to improve web testing. The executed empirical studies show that we can learn from already completed web test suites how to interact and test functionality. This knowledge can be transferred to new previously untested target applications to guide testing towards relevant functionality and it is possible to at least partially translate test suites across applications.

Chapter 6 describes how topic-driven testing can be used to automatize Android app testing through the GUI. By providing a general set of use case descriptions written in human understandable language, testing can be quickly guided to relevant functionality on a large set of applications and better penetrate a SUT.

Chapter 7 concludes this dissertation with a summary of the results, the lessons learned on the way, and key ideas for future research.

1.3 Publications

The dissertation builds on the following published research papers (chronically ordered):

Andreas Rau. Topic-driven testing. In *2017 IEEE/ACM 39th International Conference Software Engineering Companion (ICSE-C)*, pages 409–412. IEEE, 2017

Vitalii Avdiienko, Konstantin Kuznetsov, Isabelle Rommelfanger, Andreas Rau, Alessandra Gorla, and Andreas Zeller. Detecting behavior anomalies in graphical user interfaces. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 201–203. IEEE, 2017

Andreas Rau, Jenny Hotzkow, and Andreas Zeller. Efficient GUI test generation by learning from tests of other apps. In *ACM/IEEE 40th International Conference on Software Engineering: Companion Proceedings*, pages 370–371, New York, New York, USA, 2018. ACM Press

Andreas Rau, Jenny Hotzkow, and Andreas Zeller. Transferring tests across web applications. In *International Conference on Web Engineering (ICWE)*, pages 50–64. Springer, 2018

Furthermore, the following reports are publicly available or under submission:

Andreas Rau, Maximilian Reinert, and Andreas Zeller. Automatic test transfer across applications. Technical report: `https://www.st.cs.uni-saarland.de/projects/attaboy/`, Chair of Software Engineering, Saarbrücken, Germany, 2016

Andreas Rau, Jenny Hotzkow, and Andreas Zeller. "Add to Cart, Proceed to Checkout": Composing Tests from Natural Language Snippets. *ACM SIGSOFT 28th International Symposium on Software Testing and Analysis (ISSTA 2019)*, 2019 Under Submission

The experiments, empirical studies, and experimental setups which are presented and discussed in this dissertation are designed and executed by the author.

2 State of the Art

This chapter shortly introduces the technical background and terminology on which this dissertation builds on. It presents the current state-of-the-art in the area of information retrieval techniques (Section 2.1), provides introduction into natural language processing , and semantic analysis (Section 2.2). The final Section 2.3 concludes with a presentation of state-of-the-art test automation frameworks in the domains of Android and web.

2.1 Information Retrieval (IR)

In general, *information retrieval (IR)* describes the process of retrieving information (relevant to a specific need) from a set of resources, such as web pages or other documents. Primary examples are search engines that let us search for web pages by short search queries and return a sorted list of more or less relevant results.

IR has a long history starting in the early 1900s when Joseph Marie Jacquard invented the Jacquard loom, the first machine to control a sequence of operations using punch cards, but we limit ourselves to present the areas and research directions relevant for this dissertation in the domains of Android and web. Even though topic-driven testing is per se independent of the underlying platform, it is necessary to extract the natural language content first. The following sections describe how this is currently done for a SUT running on Android and web. Chapter 3 will later describe the changes to the underlying technology required for topic-driven testing.

The GUI structures content in a human understandable way to convey information and abstracts the underlying technical complexity, such as network communication or event handlers that take care of the actual user actions. This technical complexity is not relevant for human understanding. Nevertheless, due to the abundance of information available in the web and Android apps it is essential to filter for *relevant* information that can be found in text content, images or icons. Both, web applications and Android apps, structure the GUI into a hierarchical tree structure, where the leaves represents the UI elements.

Retrieving the relevant information typically encapsulates two tasks, which are presented in the following sections: (i) structuring the raw input and (ii) filtering irrelevant information, so called *noise*.

2.1.1 Information Extraction

Android In Android, the UI is structured into a so called tree-like *widget hierarchy*. The nodes of this tree, called *widgets*, are the technical interface that encapsulates all UI elements, e.g. structural elements, container elements, ImageViews, text labels, icons, or buttons. For static elements, their layout can be predefined by an external property file (i.e. `layout.xml`). In additions, apps can load dynamic content over the network into so called WebViews. By nature, the layout of WebViews and their content can be dynamically changed as well.

Each widget has certain properties which can be used to determine if an element is visible, can be clicked, or the like. Furthermore, properties contain information about the displayed text (text and description property) and determine the element position and element dimension in the UI. Android offers to access this information for testing using the android debug bridge (ADB). This interface is used by most Android testing frameworks to remotely executes commands on the device and exchange data between computer and device. The underlying technology stack is of minor interest for this thesis, but extracting this widget structure is implemented by the UI testing tools presented later in Section 2.3.2. Topic-driven testing analyzes the UI hierarchy and analyzes the presented properties. The initial UI extraction is built upon existing Android testing frameworks.

Web Applications The GUI of web applications is typically rendered in browsers. As with Android apps, the UI is structured into a tree-like structure, called the Document Object Model (DOM). All UI elements are nodes in this XML-like structure that have a unique identifier, the `xpath`. It also stores UI properties such as HTML object type (e.g. `div, img, label`), text content, placeholder texts, rendering options, dimensions and display position.

To extract information from the UI, there exist web testing frameworks, such as SELENIUM [15]. SELENIUM offers an HTTP interface to remotely control a test browser and exchange information, such as the DOM-content, between the controlling computer and a web application.

2.1.2 Page Segmentation

An important step for information retrieval is to structure the retrieved raw text content from the SUT for further classification, such as ranking the content according to its relevance or filtering out irrelevant information such as advertisement (i.e. noise). Especially web applications have a lot of irrelevant content present on the page. Examples are external content (advertisement), tracking utilities (activity monitors), or template structures used to unify the interface on all pages of an application. These structures do typically not include useful properties for classifying a web page. It is therefore necessary to *partition* each page to identify relevant and coherent information. Even though both of the discussed platforms structure their UI in a tree hierarchy, developers can freely place UI elements independently with additional layout properties. In other words, a proximity of elements in the DOM or the widget hierarchy does not imply a visual proximity in the rendered UI.

Álvarez et al. [3] proposed to use *clustering techniques* as well as *edit distance techniques* to extract data from the web automatically. By analyzing the UI hierarchy one can extract and group table structures according to their position in the DOM. With the development of highly reactive web applications and apps, this strict hierarchy analysis yields unsatisfactory results. Win et al. [85] showed why structural DOM properties are insufficient or downright harmful for correct information retrieval and classification. Purely structural page segmentation do not follow a human understanding of the presented content and are susceptible to dynamic advertisement our noise. Instead, Vision-based Page Segmentation (VIPS) techniques [16] are superior in filtering and structuring the displayed content. If the data set is large enough (e.g. a few hundred web pages), it is possible to train heuristic models for instance based on the distribution and density of UI elements [38].

One of the main goal of topic driven testing is to guide *quickly* testing towards relevant functionality. With a heuristic page segmentation that requires an input of a few hundred pages first, let us say just for testing the Amazon application, every testing advantage is lost due to the extensive setup/training phase of page segmentation. The latest version of a more lightweight approach was presented by Akpınar and Yeşilada [1] that is able of segmenting HTML documents. They apply a set of predefined rules (see Table 2.1) to replace the heuristically learned distribution and can be applied without an initial learning phase, accordingly.

Algorithm 1 shows a pseudo-code implementation of their technique. By iterating the DOM recursively, their advanced VIPS algorithm groups together visually close elements. First, all invalid nodes, i.e. nodes with no visual representation like `STYLE`,

Table 2.1: Vision-based Page Segmentation (VIPS) rules for element clustering. If the node property is fulfilled for a certain node, the listed action is executed. *'NC'* creates a new visual cluster. *'T'* - traverse recursively and analyze child nodes

Action	Node Property
NC	tag is `HEADER`, `FOOTER` `NAV`
T	dimensions equal to parent-node dimension
NC	all children are *virtual text* nodes[1]
NC	tag is `UL` or `OL` and has only one child
T	tag is `UL` or `OL` and has multiple children
NC	tag is `LI`, i.e. list item
NC & T	tag is `TD` or `TR` (table row or cell), $width > 100$ pixel
NC & T	is *line-break* node
T	is **not** an inline node

`SCRIPT` or `META`, are removed out of the DOM. After that—starting with `BODY` as the root node—the DOM tree is further traversed recursively. For each node, the algorithm checks, if the element is in between predefined boundaries $isValid()$, i.e. that an element is within the viewport of the page and has no empty dimensions. $isValid()$ thus checks, if a UI element can actually be seen by the user. However, if a node is invalid, possible child nodes are still traversed. Otherwise, container nodes with zero height or width would be excluded and we could not achieve the desired granularity. Afterwards, the node properties are checked against a set of extended VIPS rules (see Table 2.1). The idea is to generate a new visual cluster (NC), whenever a new visually aligned group is presented in the DOM.

After processing a DOM tree, this algorithms provides us with a list of visually aligned elements, which allows for a more efficient noise reduction.

2.1.3 Noise Reduction

Noise reduction is a methodology to differentiate valuable information from irrelevant information. Bar-Yossef and Rajagopalan [9] define *noise* as a combination of template structures and pagelets that hinder information retrieval and data mining, e.g. crawling. Especially web applications use predefined layout templates and boilerplate code to structure their pages. This allows for an easy maintenance and styling, but also includes

[1]An inline node that contains text and/or (inline) child nodes with text content

```
Input: $n$, root node of a DOM tree
Result: $B$, a list of block $b$
// LBN = LineBreakNodes
// IN = InlineNodes
begin
    Let $C \leftarrow n.getChildren()$
    if $C \neq \emptyset$ then
        for c in C do
            if isValid(c) then
                applyRules(c)
            else if $(c \in LBN \wedge c \notin IN \wedge c.getChildren \neq \emptyset)$ then
                blockExtraction(c)
        end
    end
end
```

Algorithm 1: Visual Element Clustering - Recursive algorithm extracting visual clusters of a given DOM tree. These clusters can be used to extract the describing context of a UI element.

irrelevant information. Using state-of-the-art noise detection techniques allows us to exclude irrelevant information from further analysis.

Li et al. [89] present a noise detection that is based on the styling properties of elements. They follow two assumptions: (i) boilerplate code follows simplistic styling properties. In order to ease consistency when this code is reused, the styling properties of boilerplate code is summarized in specialized CSS classes. This means that within the actual DOM, elements with less styling properties are less likely to present relevant information. (ii) The second assumption is that relevant elements have diverse actual content. By analyzing the DOM tree each element gets annotated with a certainty factor called *node importance*. After mining about 500 pages, the algorithm scales and determines an appropriate cut-off value to classify noisy elements.

Vieira et al. [82] build on this work and extend it by a tree mapping algorithm to calculate the similarity of the styling properties of partial DOM-trees. If elements within a subtree are repeatedly styled the same way this implies that they are created by boilerplate code. The technique is based on the tree edit distance [71] requires less samples, but does not differentiate between global noises (e.g. caused by templates) and local (intra-page noise) such as local advertisements. Lingwal [45] presented a combined noise reduction and content retrieval framework. The presented *Content Extractor* classifies the content by building a word vector that extracts information by analyzing the text content using term frequency-inverse document frequency (TF-IDF).

2.2 Natural Language Processing

Topic-driven testing is essence mining and interpreting the natural language of the GUI to learn testing procedures and guide testing through an application. Natural Language Processing (NLP) is a subfield of computer science that investigates how computers process and analyze large amounts of natural language data and includes tasks such as speech recognition, natural language understanding, and natural language generation. It may also be seen as an extension to information retrieval, i.e. as a way to interpret retrieved information.

While the field of Natural Language Processing (NLP) itself offers many techniques, we will focus on a small sub-class which is beneficial for the presented method of topic driven testing: inferring the semantic meaning of a system under test. According to Baroni et al. [10], there exist two categories of techniques for such a semantic analysis: context-count-based methods which we discuss in Section 2.2.1, and context-predictive methods, which we discuss in Section 2.2.3.

Explained in a nutshell, count based methods statistically count how often some words occur or co-occur with other words in a large text corpus. The produced statistic is then mapped to a small dense vector for each word. Predictive approaches rather try to predict a word from its neighbors. They are trained on smaller, dense embedding vectors that represent the parameters of the model.

2.2.1 Topic Analysis

Standard NLP techniques allow to process the textual content, create short summaries, and extract the main topics. In natural language, every combination of words—from sentences to paragraphs or complete documents—are supposed to convey a meaning. At the document level, text understanding can be done by analyzing its topics. Technically, a *topic* describes a distribution within a vocabulary. In other words, the combination of certain words describe a topic. Learning, recognizing and extracting the topics of a document is called *topic modeling*. In general, all topic modeling techniques are following two basic assumptions: (i) each document consists of a *number of topics*, and (ii) each topic consists of a certain *word distribution*.

Topic-driven testing already encapsulates the term *topic* and is relying on topic modeling techniques. Topic analysis is used to tie functionality of a system to a describing label and is used to identify what functionality is present in a given UI. This section gives a brief introduction into state-of-the-art topic modeling techniques. Later, we will see how they are incorporated into our framework.

One of the first topic modeling techniques was presented by Luhn [46] in 1958. He presented a heuristic method to summarize technical documentations called *Luhn's methods*. Sentences in a document are ranked after the number of occurrences of *significant words*. Sentences with a high ranking are then selected for each paragraph as representatives and can be used to summarize a document. In practice, all topic modeling methods are enhancement to this fundamental idea. A representative term describes the underlying topic. One of the extension to this method is the *term frequency-inverse document frequency (TF-IDF)* algorithm already briefly mentioned in the previous section of information retrieval. Erkan and Radev [21] describe the two relevant parts of this method:

$$tf_t = \frac{count(t)}{\sum_{t'} count(t')} \tag{2.1}$$

$$idf_t = \log\left(\frac{N}{n_t}\right) \tag{2.2}$$

$$tf.idf_t = tf_t \cdot idf_t \tag{2.3}$$

First, the *term frequency* (tf_t) describes the probability in which a term t, i.e. a word, occurs in a document. Second, the inverted document frequency (idf_t) describes how often t occurs in a set of documents, where N is the total number of documents and n_t is the number of all documents that contain t. A large term frequency value tf indicates a high importance of t, but only if the self-information of t with respect to all the document (idf_t) is high. Accordingly, TF-IDF is the product of these values.

Mihalcea and Tarau [54] propose a ranking method based on an undirected unweighted graphs. The nodes of this graph model co-occurring terms in a document. Like the presented Luhn's method, terms are associates with scores depending on their importance of the document. Elements with a high chance to occur together are chosen as representatives for a part of the document and express a topic.

Yihong and Liu [26] presented a topic modeling method that is based on the latent semantic meaning and derive sentences that describe a document. Their method describes a document as a sentence matrix.

In order to find out which sentences describe a document, they apply a principal component analysis on the matrix, called *singular value decomposition (SVD)*. By doing so, they find those sentences that cover most of the variance in the documents [75].

Finally, Blei et al. [13] introduced *Latent Dirichlet Allocation (LDA)*, a probabilistic topic model to annotate document archives with thematic information. LDA assumes that the order of words in a document is irrelevant for the underlying topic model (bag-of-words assumption). Intuitively, LDA takes a list of documents, an estimation about the number of topics n that are present in these documents, as well as a value m, which is the number of terms that describe this topic. n and m can for instance be defined manually. Now, LDA analyzes the documents and identifies for each of the n topics, which m keywords describe a topic best. The result is an $n \times m$ probability matrix that defines the significant keywords in the given documents. Technically, LDA is based on a three-level Bayesian network. Each term of a document is modeled as a finite mixture over an underlying set of topics, while each topic is modeled as an infinite mixture over an underlying set of probability values. Using an empirical Bayes parameter estimation algorithm on the network, LDA can derive those words in a document that express a list of topics best.

The presented methods need to be trained on a large set of documents (i.e. corpora), may require supervised learning, or previous document labeling. For the sake of topic driven testing, the derived topic models are also often not precise enough. The functionality we try to encode, is typically not found in large text documents. In the end, we depend on a hybrid approach for topic analysis, if the textual content in the UI is significant enough, otherwise we interpret and match the semantic meaning of elements by analyzing word vector models (see Section 2.2.3).

2.2.2 Neural Networks

Neural networks originally refer to a network of circuits or neurons such as the human brain. In computer science, they are typically used to express an artificial intelligence or machine learning models. In this section, we will do a short digression into this topic since most state-of-the-art predictive models rely on neural networks to learn semantic correlations between words, sentences and documents. A neural network can be used for information processing and models information as part of a network. Based on the connectivity within the network, information is linked to each other.

The amount of relevant information in a document is limited. Count based models have to iterate over the whole text corpora and have to keep track of all counts. Prediction based algorithms based on neural networks do not have this particular disadvantage, since they only keep information on a local level called *layer*. Irrelevant information is thus not accumulated and training on neural networks can be done on a larger set of documents.

2.2.3 Word Vector Models

While topic modeling is able to derive the semantic concepts behind documents, the produced output is typically not precise. Topic modeling is more focused on creating summaries or is designed for efficient indexing of information. In contrast to modeling information with neural networks, training a count based model is expensive and does not scale when the training corpora gets large [55].

To counteract this shortcoming, one possible alternative are so called word embeddings. Traditionally, natural language processing systems consider words as unique symbols. A term such as 'shopping' is assigned a fixed identifier (say id1234) while the term 'ordering' is modeled as id4321. The encodings can be arbitrary, for instance created by their occurrence in a certain text. The identifiers themselves do not convey any useful information such as relationships that may exist between the terms. *Learning* something about how 'shopping' and 'ordering'—e.g. that the terms are semantically linked—is unlikely and means that training statistical models requires much more data.

Vector models encode words in a continuous vector space. Semantically similar words are mapped to nearby points. Intuitively, vector models depend on the *Distributional Hypothesis*, which expresses that a semantic meaning is often expressed by words that occur in the same context. Formally, a word vector model (word2vec) is a "map of words" representation, i.e.:

$$R : \text{Words} = \{w_1, ..., w_N\} \rightarrow \text{Vectors} = \{R(w_1), ..., R(w_N)\} \subset \mathbb{R}^d \tag{2.4}$$

such that the *meaning of words* is equivalent to the *distance of vectors*

$$w_i \approx w_j \equiv R(w_i) \approx R(w_j) \tag{2.5}$$

2.2.4 Semantic Similarity

The graphical user interface abstracts complex technical events and is designed for human understanding. The natural language content helps users to achieve their goal by understandably exposing the functionality, i.e. the *features* of the application, e.g. by presenting a describing label next to input fields or by filling fields with expressive default data. Human understanding allows to grasp the underlying semantic meaning of such descriptions. Even with a vague idea of a semantic concept, a human can transfer domain specific knowledge from one application to another by matching semantically similar concepts (like selecting a proper payment method and providing valid input).

Semantic similarity itself has been intensively researched in the field of human machine interface (HMI) and document classification methods. It has been shown that one can find semantic similarities between words [70] and strings [30] by training a word vector model (short `word2vec`) on a large set of documents, i.e. large text corpora. The key idea of these models is to express the words as vectors in a vector space. Based on the training data, words expressing similar concepts are mapped to nearby points. These *word vectors* capture meaningful semantic regularities, i.e. within the given documents one can observe constant vector offsets between pairs of words sharing a particular relationship.

In order to measure the semantic similarity between two words one calculates the *distance between their vectors*. As per the definition above, vectors close to each other represent words with a similar semantic meaning. The *cosine-similarity* ($\cos(\theta)$) is a measure of similarity between two vectors that are non-zero. It measures the *angle* between two vectors. The *cosine* ranges between 1 for an angle of 0° and is less than 1 for angles in the interval $(0, \pi]$. Orthogonal vectors (90°) have a cosine of 0.

Accordingly the cosine-similarity ($\cos(\theta)$) of two given word vectors A and B expresses the similarity or dissimilarity of the represented words:

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|_2 \|\mathbf{B}\|_2} = \frac{\sum\limits_{i=1}^{n} A_i B_i}{\sqrt{\sum\limits_{i=1}^{n} A_i^2} \sqrt{\sum\limits_{i=1}^{n} B_i^2}}$$

The values of $\cos(\theta)$ typically range in the interval $[-1, 1]$, where negative values converging to -1 express a high dissimilarity and values close to 1 express semantic equivalence.

In contrast to many other vector similarity measures (e.g. the *Euclidean distance* that measures the straight-line distance between two points), the cosine is a *judgment of orientation* of two vectors rather than the *magnitude*. The magnitude does not express the semantic similarity of two words. On top of that, cosine similarity has a low complexity and scales very well even on higher dimensional vector spaces with sparse vectors: it only considers the non-zero dimensions of each vector.

2.3 Automated UI Testing

So far, we have seen the technical background on how information can be retrieved from a SUT and how the natural language texts can be processed. On the other hand, we did not discuss how applications are tested. Automated testing procedures are not a recent development. The advantages are obvious. Manual testing is expensive, not reproducible and error prone in itself. To find bugs efficiently, software is typically tested on multiple levels. On *Unit* level, small entities (e.g. individual functions and classes) are tested in isolation. Since the setup cost for unit tests is small, a wide variety of potential inputs can be executed. In order to test the interplay of different system components, the application is also tested on a set of *Integration* tests. Especially components communicating with other entities (e.g. interfaces) are tested together.

In the end though, both unit and integration tests may test behavior, which is not reachable in a productive environment. The business logic of the application could for instance prohibit access to restricted resources by encoding the authentication in the state model itself or integrate special 'guards' in the front-end code which disallow potential invalid inputs before they harm a system. From an end-user perspective, *system tests* are closest to reality as all system parts are actually executed together. Most software, which is developed for non-expert end-users, typically features a GUI which hides the technical background and instead shows a human understandable interface. Testing a system through the GUI is another important step for developing software.

With the recent requirements to be functional on multiple platforms manual testing has become further unfeasible. Instead, the focus is put on random testing (e.g. random crawlers) and platform independent frameworks to communicate with the underlying system resource such as a browser. Random crawlers are effective for load and penetration testing. Without an application specific configuration, random crawlers tend to randomly click through the application and also fill input fields with random input. Thus, they explore all possible input combinations to exhaustively detect every reachable UI state. Given enough (e.g. infinite) time, all application behavior can be explored.

2.3.1 Web Testing Frameworks

Modern web applications rarely present static HTML content anymore. Instead, they often are highly dynamic and depend on Asynchronous JavaScript and XML (AJAX), a web development that allows to create asynchronous web applications. The idea is to split the web application code into two parts. First, there is the *server side*, which is run on a remote web server and offers an asynchronous data exchange layer. Second, there is

```
// Request a browser connection
val driver = new RemoteWebdriver();
// Open web page
driver.get("http://www.ebay.com")

// find + select element with DOM property name equals 'email'
val element = driver.findElement(By.name("email"))

// send appropriate text inputs to field and confirm
element.clear() // remove current text content
element.sendKeys("test@test.com")
element.sendKeys(Keys.RETURN)
```

Figure 2.1: A short SELENIUM test script that simply opens a web page, searches for an input element with name ‘email‘ and sends a random string as input.

the *client side*, which is displaying the GUI in a web browser, such as Internet Explorer, Chrome, or Firefox. The content, displayed in the browser, can be dynamically changed without the need to reload the entire page, which is a lot more efficient.

Testing the application should be independent of the underlying browser. To automatize system testing on these applications, it is necessary to control them remotely without the need for a specialized infrastructure for each browser. SELENIUM [59] is a test automation platform that abstracts the underlying browser for testing purposes and allows to remote control a browser using simple HTTP commands. SELENIUM offers a web API, that allows to send commands, such as 'click', 'hover', or the like to the browser and thus execute user commands on the interactive web elements. SELENIUM takes care of the translation of commands and executes the appropriate JavaScript code to interact with the elements in the browser.

Developers can define test scripts that control the browser, retrieve the content and state of individual DOM elements, and thus effectively test an application. Figure 2.1 shows an example for a short test snippet. The test script is simplified and does not contain any robustness measures (e.g. waiting for the DOM element 'email' to be loaded) or an oracle to verify that the actions were successful. Opening a web page and sending inputs to a target element is straightforward. The test scripts can be executed as part of an automated CI process and be used for regression testing. Nevertheless, the sample

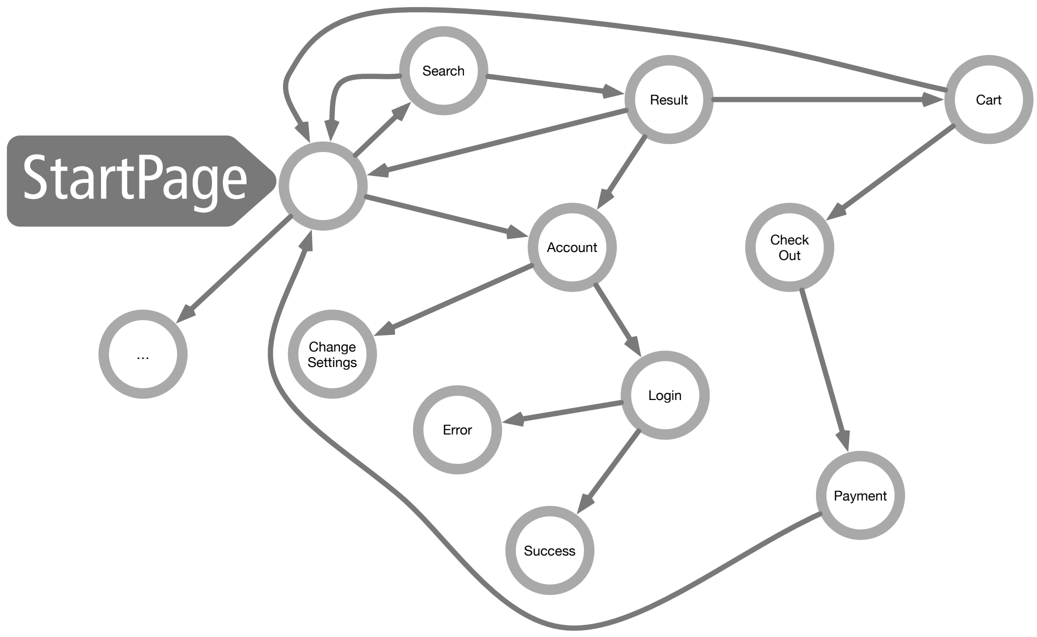


Figure 2.2: An incomplete sample exploration graph of a web application. The states of this finite state machine (FSM) are the reachable GUI states as displayed in the browser. The edges include the required actions to transition from one UI screen to another.

test is highly susceptible to changes of the underlying application. If the selector is a concrete `xpath`, the simple addition of an element above the target element may change the DOM structure and break the test execution. While it is possible to generate tests that are robust against application changes, it still reflects a core problem of transferring tests or application behavior. Interpreting instructions syntactically, makes them breakable.

Modern crawlers [20,51,52,74] communicate with the browser to control the SUT by using scripting commands via SELENIUM or directly execute JavaScript command in the test browser. Mesbah et al. [51,52] presented CRAWLJAX, an open source web crawling framework with a default random strategy. It builds a client-side state model of the SUT and explores all click-able elements and sends random strings to input elements. An example is given in Figure 2.2. The state model is a representation of the observable GUI changes in the browser. The state model is a finite state machine (FSM) that models each UI screen as a *node* and each controlling action of the crawler as an *edge*. Instead of purely randomly clicking on UI elements, CRAWLJAX tries to maximize the model coverage.

Dallmeier et al. [20] presented WEBMATE , another web testing framework that combines crawling with cross-browser testing. WEBMATE is not only crawling an application in one browser, but also analyzes the DOM for visual and behavioral differences when the same application is displayed in different browsers. WEBMATE features advanced DOM analysis capabilities and UI element analysis components. WEBMATE can compare DOM structures to identify differences and further analyzes UI elements for their visibility, styling properties, and position. This dissertation builds on WEBMATE and its advanced analysis capabilities to evaluate the effectiveness of topic-driven testing in the area of web testing (Chapter 5).

2.3.2 Android App Testing Framework

The development of smartphones, tablets and other mobile devices started about a decade ago. Not only did it open up a new market for application development, for instance mobile games, but it also started a trend to develop applications which are both accessible over the web as well as (native) apps installed on the mobile device itself. It is also an advancement of the already presented AJAX principle. The front-end code on the client side is accessible through the web browser, and the app running on the mobile device. The back-end code is still executed on a remote server, but can be queried by both the mobile app and the browser. As of today, the leading operating systems for running mobile applications are Android and iOS. This dissertation focuses on Android apps, which has a wider tool support. Still, the principles presented in this work should be applicable on all mobile platforms as well as other software accessible via the GUI.

From a testing perspective, the availability of a software on multiple end-user devices requires additional testing effort. This so called *cross-platform testing* ensures the correctness of a system independent of the test device.

In contrast to web applications, mobile apps have to adhere to strict hardware dependent limitations, such as the screen size, energy consumption restrictions, or a limited network bandwidth. App development and app testing have to take these limitations into account. Android offers a software-interface called android debug bridge (ADB), which can be used to connect an Android device to a computer, remotely execute commands or exchange data between the device and the computer. Furthermore, Android offers the testing framework *UI Automator*. Like SELENIUM, it offers an API that allows to inspect the layout hierarchy (i.e. the widgets), retrieve state information and perform operations on the SUT. Both frameworks are widely established and build the foundation of many automated Android testing tools that test an app through the GUI.

Testing mobile applications through the graphical user interface is nowadays heavily automatized. *monkeyrunner* [5], for instance, is a random testing framework that simply emits *random UI events* and is a default testing tool of Android offered alongside the operating system. *monkeyrunner* does not analyze the SUT at all and does also not give feedback about what happens during the exploration. Text inputs are only generated by chance for instance if the keyboard is opened and the generated events trigger an input. While easy to use, most of the generated events do not have an effect on the SUT.

Choudhary et al. [19] assembled a study about state-of-the-art testing frameworks for Android. For one, there are purely random input fuzzers, such as *Dynodroid* [47], *DroidFuzzer* [88], and *IntentFuzzer* [73]. Dynodroid is a more efficient version of monkeyrunner as it observes UI system events to generate more targeted UI events. It can select guide exploration to regions which were not covered yet by analyzing the distribution of the already executed UI actions. Furthermore, it can analyze the UI context to select more relevant UI events for testing and lets a user manually define inputs (e.g. for authentication). Intent Fuzzer is a tool that is analyzing how an app interacts with another app installed on the same device. It incorporates a static analysis component that identifies the structure of intents—the Android mechanism to allow inter-app and inter-process communications, such as resource accesses. Intent Fuzzer crafts these intents to analyze apps for security vulnerabilities. DroidFuzzer is a specialized input generator that generates inputs for apps that accept specific formats such as AVI, MP3, or HTML files. It identifies vulnerabilities in these apps, i.e. by dealing with errors handling these inputs.

The second category are tools that provide *model-based explorations*. Like the presented web crawlers CRAWLJAX and WEBMATE they systematically explore the behavior of the app and craft specific events targeted on the SUT. Using an application model, e.g., a FSM, should allow to create more effective crawling strategies that exceed the capabilities of the random approaches in terms of code coverage or function coverage by creating less redundant inputs. On the other hand, model-based approaches typically extract the current state of an application only through the UI at discreet points in time, say every second for instance. In other words, internal changes not observable through the GUI or changes that occur in between the discreet points in time are not observable to the testing approach. As already mentioned, this limitation is also present topic-driven testing, since it is built upon a dynamic model-based technique.

GUIRipper [4] dynamically builds an exploration graph while testing the SUT. Again, the states of the graph are the UI screens and the edges the applied action to go from one state to another. Upon reaching a new state, GUIRipper analyzes the current state for all possible *activities* (that is all possible things a user can do) and extracts a list of events that can be generated. By doing so, it systematically explores the state model using a Depth First Search (DFS) algorithm until no new states are discovered. It also accepts a set of inputs that are then inserted into the app during the exploration. ORBIT [87] is principle using the same exploration method like GUIRipper, but analyzes the source code of the AUT to further filter the list of events. It optimizes the exploration by filtering events that are unlikely to succeed in the current state. Both test frameworks restart an application if no state change can be observed for a certain period in order to explore novel behavior. SwiftHand [18] aims to reduce the necessity of restarting the application as it is quite costly. In our experiments it takes about five to ten seconds to re-start an application, but it is also costly to re-navigate to a previously reached state, since one has to re-execute all performed actions. The exploration strategy of SwiftHand is purely executing click and scroll UI events, which can be fast and efficient, but also limits the usefulness. A^{3}E-Depth-first [8] introduces another state model for their testing procedure. Instead of being based on the UI screen, it is purely based on the activities. This model is rather abstract and not as precise as the state model and misses even more application behavior, but can have a higher performance. PUMA [27] is a test framework that allows a test developer to programmatically define testing procedures, and include arbitrary dynamic testing features.

Besides Intent-Fuzzer and ORBIT, all presented frameworks are purely *black-box approaches*. They do not analyze or instrument the source code of the application and can thus run arbitrary pre-compiled applications. Unfortunately, all model-based testing techniques in the presented set require at least an instrumentation of the app in order to extract the activities. *Instrumentation* is the process of changing the compiled code of a piece of software in order to change its behavior. In dynamic testing it is often used to integrate external code snippets for analysis purposes, such as calculating coverage data, runtime analysis or extracting the possible activities in a UI screen. Changing the app behavior while testing is common, but also raises the question if the *real* system behavior is actually the same. At least an additional runtime overhead can typically be expected, which may have an impact on the analysis. To counteract this, Li et al. [43] presented DROIDBOT, "a lightweight UI-Guided test input generator for Android". Without instrumenting the SUT, DROIDBOT generates test inputs and dynamically learns a state model. It allows to define custom exploration strategies and to analyze the UI hierarchy. DROIDMATE developed by Jamrozik et al. [35] and its refined version DROIDMATE-2 (DM-2) by Borges et al. [14] are comparable techniques. They offer a compromise: the exploration can be done without an instrumentation by analyzing the widget hierarchy, screenshots, or the like, but one can define an instrumentation if further analysis properties are required.

The app testing part of our evaluation of topic-driven testing (see Chapter 6) is integrated into DROIDMATE-2 (DM-2). We describe the necessary extensions to the underlying code base of DM-2 and how it can be integrated in further testing. DM-2 and its extensions are publicly available [48]. For our purpose, DM-2 offers an extendable plug-in technology for external exploration strategies and UI analysis purposes and lays the foundation for topic-driven testing to interact with the test applications. In theory though, the testing principles are independent of the underlying testing framework as long as it allows us to analyze the GUI and to remotely execute control commands.

The presented list of tools and testing techniques barely scratches the surface of the ongoing research in the area of Android testing. Vásquez et al. [81] present an even more detailed overview containing more than seventy mobile app testing frameworks. It suffices to say that none of these frameworks offers a testing technique that is close to TDT in terms of versatility (runnable on multiple apps), platform-independence (applicable to web and mobile testing), or expressiveness (knowledge transfer).

2.3.3 Semantic Crawling

So far, we have discussed non-specialized crawlers for the web and mobile apps. The following section presents so called *semantic crawlers* that are closer to topic-driven testing.

While humans by default use semantic concepts to interact with applications, the nowadays widely used automated random crawlers do not leverage this information. Generating valid input, i.e. user actions, for complex applications and differentiating the resulting output states is crucial to every crawling technique and automated testing solution [47]. Random crawlers use randomly generated inputs (or a set of user specified rules to fill for instance password fields) to test an application exhaustively. If applied to real industrial applications, having huge state spaces, they often face a state explosion problem. In other words, a GUI offers so many interaction possibilities in every state, that exploring them is already a problem. Pure random testing does not scale to large applications [17]. But even worse, the possible interactions are linked to each other. Take for instance an authentication step. Typically, the login procedure first enables further functionality in an application to protect sensitive data from external access. That means that single actions (insert user name, insert password) cannot be considered independently, but have to be executed in a specific order. This leads to a combinatorial explosion of exploration paths.

Random crawling techniques suffer from two major challenges: (i) guiding the crawler towards relevant core functionality, and (ii) generating valid input combinations. We already discussed the differences between random crawling and the possibility to integrate model based testing techniques in order to reduce the amount of repetitive actions or actions that do not cover more functions. Testing is in general bound to a certain budget, be that time constraints, hardware limitations or simply money. Accordingly, it is essential to test those application parts that bear a high risk of containing software bugs.

In the area of web site crawling, Chakrabarti et al. [17] presented the concept of *focused crawling*. In essence, their crawler is given an example-based "canonical topic taxonomy" that is used to identify relevant pages in an AUT. The user is generating a list of URLs that contains valuable or interesting information. The focused crawler now offers a list of classes that can be discovered on these pages (say topics such as business, traffic, or sports). The user can now filter this list according to her preferences. Using this semi-supervised learning, the focused crawler learns relevant "*vantage points*", i.e. words that are expressing the topic, and follows links that are expressing similar topics. The training is typically done on a few dozen starting examples per topic. Focused crawling was developed in the era of static HTML pages and in general used for information retrieval and indexing purposes. While the approach is no longer state-of-the-art, focused crawling is designed to crawl and extract content based on user specified properties, which is also a goal of topic-driven testing. In contrast to topic-driven testing the presented focused crawler cannot express complex use cases such as buying a product nor qualifies as a testing procedure because it does not create inputs.

Generating valid input combination is a severe challenge for every crawling techniques. Syntactic differences between applications make a transfer between previously learned input specifications to new applications virtually impossible. In other words, the task to create valid inputs has to be repeated if the AUT evolves (and changes the syntactic structure), or if new applications should be tested. Without valid inputs, it is hardly possible to test applications and their use cases.

Use cases [33] describe frequently used interactions scenarios between a user and a piece of software. Such an interaction scenario is a detailed description to achieve a certain goal, such as buying a product. Our initial example (Figure 1.1) showed the workflow for "buying a product". A concrete use case description (i.e. in Pressman style [60]) incorporates a list of specific How-To instructions that express the corresponding work flow. Lau et al. [41] performed a study on how written How-To instructions can be mined into machine executable commands for testing. Their intention is to guide users through complex scenarios. Most real world application somewhere include a frequently asked question section (FAQ) or a help section, where they present a step-by-step instruction list on certain use cases. They are grouped under a general problem

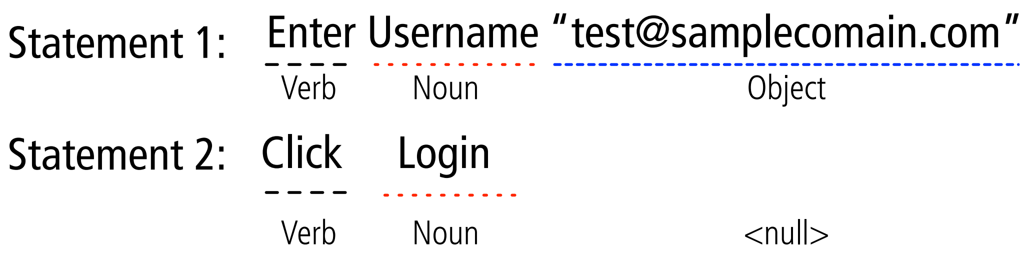


Figure 2.3: Example for deriving ATDs (a,t,d) from natural language using Part-of-Speech Tagging. For each statement, the verb specifies the action a, the noun the target t and other nouns (i.e. objects) the data d.

description (e.g. log into your account) followed by a list of single line instructions.

Lin et al. [44] present the problems of mining such natural language instructions for further test usage, such as out-of-order instructions or the presence of implicit knowledge that is not part of the instructions themselves. Thummalapenta et al. [78] presented an extension of this work and present an automated testing technique (later called ATA) for GUI-based applications. ATA is essentially translating natural language test descriptions into machine-executable instruction, i.e. use them for automated test creation. Given the description of a manual test case specification akin to a use case description, they process the natural language instructions to build an exploration graph. Figure 2.3 shows some example instructions for a login procedure. Using a part-of-speech tagger [79], the individual instructions are split into verb, noun and an (optional) object. The verb represents the *action* which should be executed on the AUT (e.g. click or enter), the noun a description of the *target* (the field name, such as the username field), and the object *optional data* that should be inserted (e.g. the email address of a user). Summarized into so called *Action Target Data Tuples (ATD)*, the instructions can then be executed by an automated testing procedure. In a proof-of-concept evaluation on three web applications, they show that ATA 68% of the given instructions could be executed on average in a semi-supervised environment. This lowers the complexity of transferring manual test instructions into an automated testing procedure and further produces test snippets that are more resilient to syntactic changes to the AUT [76].

In a later work, Thummalapenta et al. [77] replace the test instructions with a more general business logic that allows to generate oracle-like information. The presented *Web Application TEst generator (WATEG)* is a framework crawls an application, while simultaneously building a state model. The business logic of an application contains additional properties such as “one the user has logged in, there is a logout button”. These properties are verified by WATEG and can thus serve as assertions for testing.

ATA and WATEG tackle both the problem of crawler guidance and generating valid inputs for an semi-automated testing procedure. Both tools are proprietary black-box testing procedures. They both parse natural language process snippets specifically tailored for each AUT. The effort for generating the manual test instructions has to be repeated for each application, though. The model does not allow to transfer knowledge to another testing platform (say mobile applications) or to other applications (say from Amazon to ebay). The mined instructions are still syntactically processed. Syntactic techniques are limited to be executed on the original application and cannot be easily transferred across applications. The tools show the potential of natural language assisted testing approaches, but are on the other hand restricted to interpret instructions *syntactically*, while this dissertation focuses on *semantic concepts* instead.

In recent years, research has made some advances in mining valid input information and integrating them into testing and crawling. Lin et. al [44] recently presented a *semantic crawling* technique that trains a neural network (i.e. a word vector model) on a large set (i.e. hundreds) of manually labeled states (DOM-trees), which allows them to re-identify complex form fields and even similar states within an application. They leverage the semantic similarity to identify input topics. Software functionality, use cases, and input forms are not identical, though. Think of a “reset password” link. It is not an input form, but still triggers underlying code. Still, it is reasonable to assume that one can extend this work to identify general features as well.

Integrating valid input values into the learning phase would also allow to leverage the procedure for improved testing. While not being evaluated for cross-platform testing, it stands to reason that with an even larger training set it is possible to extend the approach to also handle this case. We have seen how difficult it can be to train a neural network and how many data points it needs to scale. The required human expertise and effort to train such a model is significant. The sheer size of the training data is another issue.

Hu et al. [28] also tackle the problem of generating robust and reusable UI tests by training a machine learning model with manually labeled UI screen samples and manually created tests to train a crawler how it can interact with different UI screens.

Compared to the work presented in this dissertation, the effort for training the initial model is significant. Lin and Wang trained their model on manually labeled document corpora meaning that identification of new features would require to relearn the models, especially as the model is domain-specific due to the relatively small training setup. Mikolov et al. [55] have shown that the quality of the word vectors increases significantly with the amount of training data at hand and that an unspecialized model trained on a large data set is superior to one trained on a small, but specialized data set. The generated models are difficult to understand and retrain. If new features have to be added to a previously trained model (in case the software evolves—say implementiing a new use case), the impact on other parts of the model are hard to estimate or control.

Semantic crawling is different to topic-driven testing in two points. (i) It has to be trained on a large set of input forms in order to learn how inputs are structured and how testing can provide valid inputs. (ii) Filling input forms is a sub problem for a guided testing procedure. Topic-driven testing cannot only fill input forms, but effectively steer the whole exploration to fulfill pre-defined use cases. In this respect it is closer to tools such as ATA and WATEG while simultaneously also leveraging semantic similarity *across* applications to facilitate testing.

Recently, Fazzini et al. [22] presented YAKUSU, a technique, which parses existing bug reports for a mobile application to extract an actionable list of instructions. These bug reports are tailored to allow humans to reproduce faulty behavior—i.e. the *steps to reproduce*. YAKUSU mines these instructions, identifies target descriptors and matches them to UI labels in the applications. While both our approach and YAKUSU interpret natural language instruction to guide testing, there are considerable differences in how these instructions can improve testing. First of all, YAKUSU is again limited to execute the instructions on the app they were written for. The main contribution of this dissertation is to improve exploration in all AUTs. Secondly, the steps to reproduce typically start in the first app screen and give the shortest path. In Chapter 6 we will show, how complete use cases that are integrated in different parts of the application can be tested. These may not be executable in the initial step. Moreover, these locations may be inconsistent in different application. Each app can freely place their functions in the UI. In other words, *one has to find the use cases in the FSM that expresses the AUT first*. Nevertheless, both techniques have complementary advantages which could be exploited in further research.

Another branch of research addresses the area of feature identification and program comprehension. Topic-driven testing also offers insight into how different features in a system can be controlled via the GUI, where they are located and connected. Xin et al. [86] presented FEATUREFINDER, a tool that analyzes the method call hierarchy (i.e. the stacktrace) of Android apps while testing it. The method location (i.e. the package), the method name and the parameters give an indication towards their purpose. A *feature* can be identified by a group of methods (i.e. a cluster) that is executed together.

2.3.4 Leveraging Other Applications for Testing

Testing (indivdual) applications is already a significant challenge. If the effort for setting up the test code and test infrastructure has to be repeated for every new application, platform or software version the testing costs explode. We have seen that there exist suitable random testing approaches for various platforms. Black-box testing approaches are arguably for the most part impartial to the underlying AUT. We have discussed the advantages and problems of random testing techniques. Treating and testing applications independently wastes valuable and important information, though.

Nowadays, software is not only developed for one end-system. To be accessible by a large majority of people, it is available on traditional desktop machines (independent of the underlying operating system or browser), through mobile browsers on smartphones and tablets, native mobile applications, embedded systems, and the like. You immediately see the effort, if the testing effort has to be repeated over and over again.

WEBMATE by Dallmeier et al. [20] was already shortly introduced as a web testing framework (Section 2.3.1). While it includes a crawler, its primary goal is to identify cross-browser issues. Intuitively, web applications are developed with the assumption that if a web server delivers the same code to a client browser, it is displayed the same way on the clients side. This assumption is far from reality. Clients have different network and hardware constraints, which influences how network resources are loaded. Network latency, network throughput, or available screen resolution are possible reasons for why an application might show differences on the client side even if the same code is send over the network. But even then, browsers and operating system offer different rendering engines which causes further inconsistencies and possible bugs. WEBMATE crawls an AUT on different test browsers and builds an application model. It then compares the generated state models and the DOM structure for differences. WEBMATE searches for missing DOM elements, which might indicate missing information or functionality, and differences on where elements are displayed or styled.

Choudhary et al. [72] follow a different approach and propose an automated technique for matching features across different versions of a multi-platform web application (e.g. mobile and desktop version) called *Feature Matching Across Platforms (FMAP)*. They compare features and functionality across mobile and desktop versions of a web application, by analyzing the underlying network communication. The abstraction of a network trace and the involved components are the basis for their feature matching algorithm. FMAP tracks the network requests the application is executing on the client side both in the mobile app as well as in the desktop application to identify and match common features. Network requests are typically captured by a man-in-the-middle proxy interacting as a gateway between client browser and server back end. The proxy records HTTP requests and their respective response in form of a serialized string, i.e. *HTTP archive record (HAR) file*. An HAR file contains information about all network requests that is executed from a client (e.g. a browser) to the back-end services (e.g. run on the remote server), e.g. the type of the request (GET, POST, etc.), the target URL, the execution time and execution duration, the MIME type, the status code and payload of the response and much more. And complete specification is available by the W3C [83]

Essentially, FMAP transforms the HTTP-requests into a set of actions and clusters them using the Jaccardian distance. Given two traces (one for the mobile app and one for the desktop app), FMAP models the clusters in a *maximum weighted bipartite matching (MWBM) problem* and finds matches using the Hungarian algorithm [40]. As a consequence, FMAP is able to identify features in applications if the given test suite is interacting with them and the interaction causes a network request to the service back-end under the assumption that the applications are communicating with the same back-end interface. Their preliminary evaluation shows that their proposed approach could identify several missing features in real-world, multi-platform web applications.

FMAP is very related to our work. Not only do they automatically identify features, but FMAP is also intended to identify features across applications. By design, it is restricted to identify features in applications communicating with the same back-end server (cross-platform applications).

By transferring domain specific knowledge *across* applications, a crawler can identify testable features and their desired output. Cross-browser testing [20] and cross-platform testing approaches [72] use existing application models (also in the form of test cases) to gain knowledge of the application under test (AUT) on one platform under the assumption that the same observations apply on another test platform. Both techniques are specialized to executions *within the same application on different platforms or browsers*. Hence, they cannot be used to transfer knowledge across applications without specifying additional executions for each application under test. Both approaches are complementary to topic-driven testing. By generating better applications models with topic-driven testing, it is possible to identify more features as well as inconsistencies between different platforms. Furthermore, topic-driven testing allows to identify and gain even more knowledge on an inter-application level.

Behrang and Orso [12] also researched the area of test migration on mobile apps with similar functionality, which can be seen as a *very* valuable extension to the early work of this dissertation [67]. Their tool APPTESTMIGRATOR shows that the principles of GUI test transfer are universal among web application tests and mobile app tests. While our work in 2018 focused on transferring test cases from one web application to others with similar features, Behrang and Orso applied the principle of semantic analysis on Android apps one year later. The foundation is pretty similar. Both works leverage the semantic similarity of labels, content descriptions and other texts to create an ontology of their test subjects. By learning from a set of initial tests, both works learn which actions are applied to the software under test and reproduce the same behavior on the target application. In the area of APPTESTMIGRATOR and android apps, this is done via observing and replicating GUI events, while the web test transfer is done by interacting with UI elements with similar semantic meaning. Behrang and Orso also transferred test assertions during their test transfer, which allows them to test the correctness of their target applications.

3 Mining the Semantics of Interactive Elements

Parts of this chapter have been published in Rau et al. [67, 69].

Generating tests for GUI testing is hard. First, it can be difficult to decide which actions are allowed for a specific interactive element. For some element types (e.g. checkbox, dropdown menu), it is obvious which interaction is required—e.g. a click or the selection of an item. For others, say input elements, it may be not as simple: the context implies, which input is required—a digit combination, a ZIP code, the name of a city.

In both cases, one has to be able to (a) generate valid as well as invalid inputs, and (b) be able to differentiate the observable output of the AUT for being correct (i.e. *expected*) or incorrect (i.e. *unexpected*). The latter problem is known as the *oracle problem*. Finally, test actions may change the application state (both on the client side and the server side). A simple example would be a shopping cart: in order to test the removal of an item, another (prior) test must have added it first. The sheer number of possible interactions together with a combinatorial explosion when trying to explore them all requires us to semantically abstract the GUI for better testing—the way humans do.

In this chapter, we discuss how we extract the semantic meaning of a GUI and its elements. Remembering the process overview of topic-driven testing in the first chapter (Figure 1.2), we discuss the right-hand side, i.e. the platform specific *Test Executor* part. This is the technical baseline for testing an application. For guiding the test procedure, the most important part is the identification of interactive elements (see Section 3.1) and linking them to their descriptive texts for a further semantic analysis (Section 3.2). We present the necessary extension made to the underlying test platforms WEBMATE (the test executor for web applications) and DM-2 (the test executor for Android apps). These extensions serve as interfaces for the later test execution, and the potential knowledge transfer.

Essential for our approach is identifying valid targets for a given test instruction and guiding the exploration process towards desired functionality. The key idea is to calculate the semantic text similarity between the describing labels of the GUI elements in the AUT and each target descriptor. We process each UI element in each state of the AUT and repeatedly identify target UI elements based on their semantic similarity to a given text description, for instance a machine executable statement or an ATD.

Summarized, the overall process is as follows:

1. *Feature Detection:* The displayed UI screen is analyzed to determine potential targets (interactive widgets or UI elements).
2. *Natural Language Extraction:* For each interactive target, we search for a *descriptive text* in the visual vicinity. This text can be found in the widget properties of the potential target itself or other surrounding labels, icons, texts.
3. *Text Sanitization:* We sanitize the descriptions to remove noise, tokenize the raw strings into words, and trim redundant spaces or illegal characters.
4. *Semantic Analysis:* We compute the semantic similarity between the extracted natural language content and the potential target, i.e. the corresponding descriptor in another target application (matching, knowledge transfer) or a use case description.

3.1 Identifying Features in the GUI

While the concepts of topic-driven testing are in general independent of the underlying test platform, observing and controlling an AUT requires platform specific implementations. Given a UI screen, such as the a login screen in Figure 3.1, we have to extract the interactive elements as well as their description, called *labels*. These labels can then be used for a semantic analysis.

3.1.1 Web Applications

Extracting Targets

The content displayed in the browser can be analyzed by inspecting the DOM. Each state in an application contains a DOM and each UI element is represented in the DOM. The DOM shows the effective properties of each web site element and for instance integrates external configuration such as style files ('.css' properties). It contains the

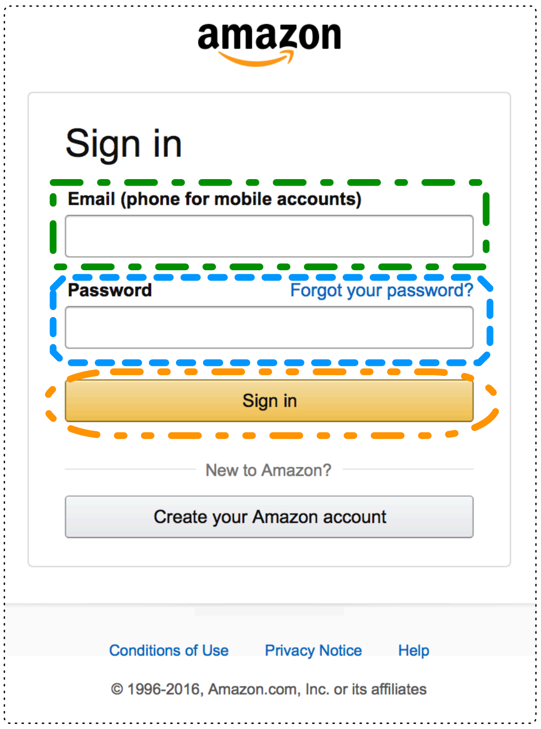

amazon
Sign in
Email (phone for mobile accounts)
Password
Forgot your password?
Sign in
New to Amazon?
Create your Amazon account
Conditions of Use
Privacy Notice
Help
© 1996-2016, Amazon.com, Inc. or its affiliates

Figure 3.1: Sample login screen. The boxes indicate the interactive elements together with their describing label.

relevant information such as the type of the element, styling attributes, display position, event handlers, and more.

Event handlers are a programmable group of properties that manage how that element reacts on events, e.g. user clicks, hover actions, or the like. To interact with an element on the page, the controlling instance tells the SELENIUM server to search for an element in the displayed DOM by giving an identifier such as the `xpath` or a certain attribute identifier (e.g. *id*, CSS-attributes). Afterwards, commands can be executed on the identified web element (e.g. *click*, *sendKeys*, *hover*).

In order to guide a test execution, we retrieve and analyze the DOM after every test action. We make use of the analysis capabilities of WEBMATE to decide if an element is visible, interactive, and where it is displayed. First of all, we extract a list of interactive elements in the DOM. That is each element with an attached event handler, e.g. anchors (`<a>`), buttons (`<button>`), menus (`<menuitem>`), options (`<option>`) and input fields (`<input>`). A full specification of interactive elements can be found in the HTML5 specification of the World Wide Web Consortium (W3C) [84]. Elements such as `head`, `body`, `meta`, `script` are excluded from the analysis, since they have no visual representation in the UI. In this dissertation the analysis is further limited to HTML5 element. External frameworks such as video players or flash are not supported by our testing framework WEBMATE and thus not further analyzed.

This list is now filtered to remove elements invisible to a human user. Imagine an invisible label somewhere on the web page. It cannot really describe the semantics of a button in the vicinity, because a human would not be able to see it. The same holds for buttons that only become visible after a preliminary actions, such as a drop down menu or a settings structure. After clicking it, a menu structure with further options becomes visible. There are valid technical reasons, why elements are invisibly loaded into the UI. A pop-up menu structure would be an example. In order to optimize the responsiveness of the system, a web application loads the required HTML content, but may hide them with additional styling options. Our analysis does not ‘miss’ hidden elements, but models them into a different state that is discovered by clicking the appropriate button first. On top of that, certain websites integrate so called *honeypot items* into the page to counteract non-human testing agents.

A machine-executable visibility analysis for web elements can be challenging. In order to check if an element is visible, WEBMATE analyzes the cascading style sheet (css) styling properties of elements (e.g. `visibility`), checks if an element is within the viewport (i.e. part of the UI that is shown in the browser), has a color that is differentiable from the background color, and checks if the element is overlain by another. The latter is done both by calculating the position of other elements and checking the stacking context (z-index) of the elements. On a side note, WEBMATE has been extended to scroll to the bottom of the page after executing an action. This is necessary, because certain web pages only load certain content after they are in the viewport of the browser. Furthermore, we consider elements to be visible, if a user can scroll to them.

Mining the Element Description

Depending on the nature of the extracted UI elements, the extracted list may not contain any analyzable natural language text content. Although certain DOM elements (e.g. of type `INPUT` store the displayed text in their attributes (e.g. the *value* or *placeholder* field), even more information is stored in the surrounding elements, i.e. the element’s context. As a consequence, we need to extract the text content of the describing labels as well, without including too much context as this may introduce noise into the later classification.

Our analysis is based on the assumption that a *short* relevant description is found

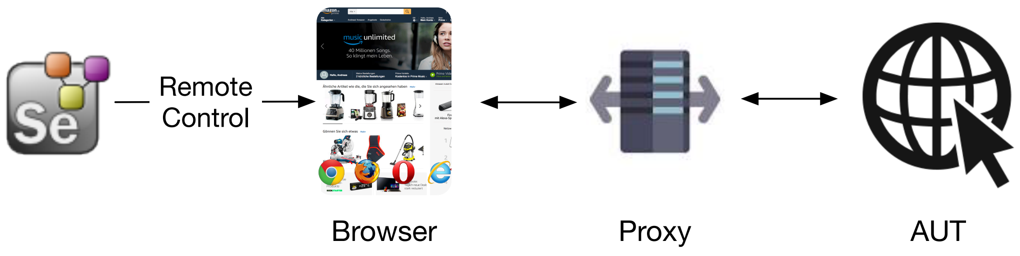


Figure 3.2: Prototype setup for capturing the application model. The SELENIUM instance controls a test browser. Network traffic between the browser and the AUT is captured by a man-in-the-middle proxy.

in the *near visual vicinity* of an element. Either an element has its own description (e.g. 'value' entry, 'placeholder' entry, 'alt' text, or the displayed text of anchors) or the surrounding elements describe the functionality. The larger the visual distance between description and interactive element, the less likely that a description is related to the element. This assumption relies on the observations of Mariani et al. [11], who analyzed the description of widgets in a GUI. They conclude that a description is in the proximity as well as typically aligned to an element. That means that they are either on the same height (horizontal aligned), or the descriptive text is directly above or below an element (vertical aligned).

We group elements of the DOM using the presented VIPS algorithm [1] (see Section 2.1.2) and calculate the Euclidean distance. If the element contains its own descriptive text, the distance is set to zero. Lengthy texts (i.e. texts with more than ten words) are excluded from the analysis, due to the assumption that there should be a *short* text describing the functionality. Texts longer than ten words are probably rather conveying other information. The list of descriptions is now sorted in ascending order (with respect to the Euclidean distance). The first element has the highest chance to be the descriptive element.

Extracting the Application Model

At this point, we have seen the technical extensions made to extract interactive elements together with their describing labels out of the browser interface. While WEBMATE offers an application model in form of a FSM, it cannot freely interpret semantic instructions or generate an application model that is required to transfer knowledge, such as tests, across applications. WEBMATE interprets individual test commands and executes them on the AUT. After executing a test command, say a click instruction for instance, WEBMATE extracts the DOM.

The key behind transferring semantic concepts across applications lays in identifying features and the elements describing their functionality using natural language (i.e. the label next to it or at least in the near vicinity of it). This information can be used to calculate the semantic similarity of features across applications. Now consider the payment screens of two sample applications (see Figure 3.3). They both have similar interaction possibilities. Both screens offer the same functions, e.g. adding a credit card for payment. A close inspection reveals that the structure is significantly different, the position of elements is different, and finally the displayed texts are different. How can we—as humans—decide that the function of this screen is to enter payment information? We first have to define what a function, i.e. a *feature* is, before we can train a machine to semantically interpret it. Intuitively, a *feature* is a sequence of actions that achieve a certain (common) goal. Now this informal definition has to be translated to a machine-executable specification.

The structural differences in the structure make a direct identification of the function of a UI element difficult. Instead, we follow the idea of the already presented FMAP approach, a new state can be reached after a network request has been executed against the application back-end service. In order to implement platform independent applications, the code of a system is split into the client-side UI code and the server-side back-end code. The front-end systems exchange data, and commands with the back-end. Our previous example on payment showed that *features* are a group of actions. In many cases a feature cannot be expressed by one single UI action. In order to enter payment information, one has to select the payment method (say credit card), the credit card company, enter a valid card number, the name on the card, expiration date and the credit card verification number. In isolation, each of these actions are not testable functions. But after submitting the information (by pressing 'Ok', 'Submit', 'Proceed' or something similar), an network request is send to the back-end service with all the relevant information. Additionally, the GUI changes to reflect a successfully interaction. By observing the network communication, we can learn which group of actions belong together and form a *feature*.

We extend SELENIUM to group individual interactions with web elements together to features. Figure 3.2 gives a brief overview on how the controlling instance finally communicates with the AUT. In order to extract the application model, we extend SE-

LENIUM to record all incoming HTTP requests from the controlling instance (i.e. the *actions*) and proxy all requests from the browser to the AUT through a network proxy.

By recording any executed test command, we learn which elements have been interacted with, i.e. the edges of the application model graph. The remote proxy allows us to further capture all networks requests the browser is sending to the AUT. The responses from the server can be filtered to only check for specific MIME types, especially `text/html` or `text/javascript`. By grouping together all recorded user actions in between such network requests, we derive the set of tested features, namely the list of UI elements on which SELENIUM actions have been executed upon. Taking the example of the earlier payment use case (Figure 3.3), the UI elements used to register a new debit card on eBay and the button *Done* or *Cancel* are grouped together as they represent a single form submission.

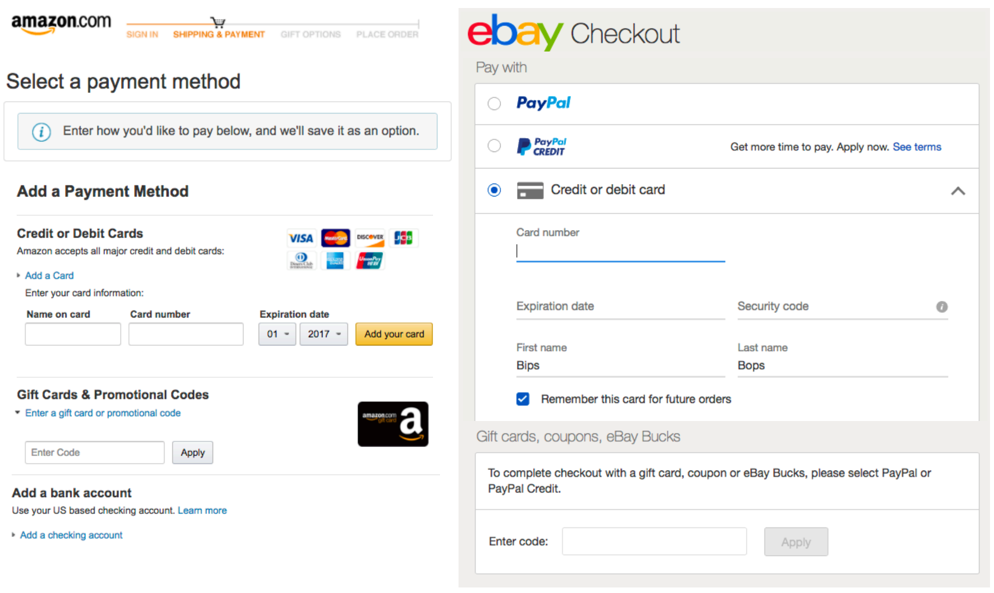
The extensions to SELENIUM do not only group UI interactions into features, but also allow us to learn and integrate arbitrary SELENIUM based test services. That is test scripts or any external crawling framework that uses SELENIUM to communicate with the AUT. This offers further integration opportunities with existing frame, but is not further investigated in this dissertation.

After grouping together the elements with their describing natural language context and learning the relevant UI interactions that form individual features. This information can be used for the semantic analysis that is the foundation of topic-driven testing. Due to the platform specific differences between Android apps and web applications, we will first look into the UI analysis of Android apps and then proceed to the semantic analysis which is platform independent (see Section 3.2).

3.1.2 Android Apps

Interactive Widget Detection

As with the presented web testing, we first have to identify which UI elements are likely to trigger new behavior in the app. For us as users, interacting with an app is pretty easy. We already discussed that Figure 3.1 shows a simplified authentication screen for an eCommerce application. We see for each input argument the respective target (email, password, and the "Sign in" button) and touch it with our finger. From a programmatic point of view, this is not as trivial—there likely is not only the sign in button which can be clicked, but rather a multitude of alternate UI elements which may be overlayed with the button. Android is using empty container elements, which are purely used to structure the UI and have no real functionality. Moreover, buttons (or other elements like input fields for the matter) are not guaranteed to be conveniently named to be easily



amazon.com

SIGN IN · SHIPPING & PAYMENT · GIFT OPTIONS · PLACE ORDER

Select a payment method

Enter how you'd like to pay below, and we'll save it as an option.

Add a Payment Method

Credit or Debit Cards

Amazon accepts all major credit and debit cards:

Add a Card

Enter your card information:

Name on card	Card number	Expiration date	
		01 2017	Add your card

Gift Cards & Promotional Codes

Enter a gift card or promotional code

Enter Code Apply

Add a bank account

Use your US based checking account. Learn more

Add a checking account

ebay Checkout

Pay with

PayPal

PayPal CREDIT Get more time to pay. Apply now. See terms

Credit or debit card

Card number

Expiration date Security code

First name: Bips

Last name: Bops

☑ Remember this card for future orders

Gift cards, coupons, eBay Bucks

To complete checkout with a gift card, coupon or eBay Bucks, please select PayPal or PayPal Credit.

Enter code: Apply

Figure 3.3: For the given payment selection screen, a user selects a suitable option based on the available ones. For testing the application automatically this allows to select a valid payment method.

identifiable. In the worst case, there may even be hidden elements behind this button which we do not want to falsely identify as our target widget. Obviously a user would not be able to click hidden nor overlayed (layout) elements by design. To simulate this, we have to ensure that we only extract one interactive widget for the buttons area. The Android operating system provides an interface to retrieve the UI element hierarchy. Thereby each element is represented as a node within a tree. Each node has certain properties which can be used to determine if an element is e.g. visible, can be clicked etc. We consider an element to be interactive if it is:

$$\begin{aligned} interactive &\equiv enabled \land visible \land accessible \\ &\land (editable \lor clickable \lor checkable \\ &\lor longClickable \lor scrollable) \\ accessible &\equiv bounds.width > min.width \land bounds.height > min.height \end{aligned}$$

We introduce our own property *accessible* to handle scenarios in which widgets can be only reached by scrolling through the screen or when they are hidden behind other UI elements like navigation bars to make them invisible or inaccessible for the user. The values *min.width* and *min.height* express the pixel value at which we assume a human user to be able to interact with the item. This value is statically set to five in all our experiments, because smaller elements are subjectively hard to hit. We identify elements which are currently not in the viewport based on their boundary property. If an action target is out of this area, DM-2 first scrolls into the respective direction such that it becomes visible, before interacting with it. The corresponding extension have been integrated into the code base of DM-2.

Furthermore, we ensure that only one interactive widget is extracted for a respective boundary. While iterating through the UI element tree, we select the deepest interactive element as distinct target for the respective subtree. This counteracts the problem of selecting overlayed widgets that are not accessible to a human user. The result of this procedure is a list of interactive widgets and the possibility to interact with them.

Mining the Element Description

Now we have successfully identified all potential target elements, which are present in the UI in Figure 3.1, but so far we are not aware of their underlying function. We would like to extract their labels, such as “Sign in”, or “Email (phone for mobile accounts)”. The labels for these elements could be defined as their own *text*, or *description text* properties, as well as through their sibling leaf node. These siblings potentially contain a text label as a property in one of the ancestor nodes or sometimes even completely unrelated to the buttons sub-tree.

If there is a text or description information on the interactive element itself, we use that text for our semantic analysis, otherwise we derive the natural language content from the nearest visually aligned UI element [11]. For each interactive widget from the earlier phase that has no descriptive text on its own we calculate the neighboring non-interactive elements with textual content. Starting from the top of the UI screen, we calculate the Euclidean distance of the neighboring elements and sort it in ascending order. Next, we consider the bounding boxes of both the interactive element as well as those of the potential label element. If their bounding boxes are intersecting, we do consider them as being correlated. Now, the text of the first element of this list is extracted as the describing label for this element. In our example, this filters out a few potential false positives for the later semantic similarity. For instance, we group together the password input field with the label “password” instead of grouping it with “Forgot your password”. The same is true for the “Sign in” button.

In essence, this methodology is similar to the one for web analysis, but takes the platform specific differences in terminology and technology into account. At this point, the platform specific differences between web applications and Android apps can be hidden

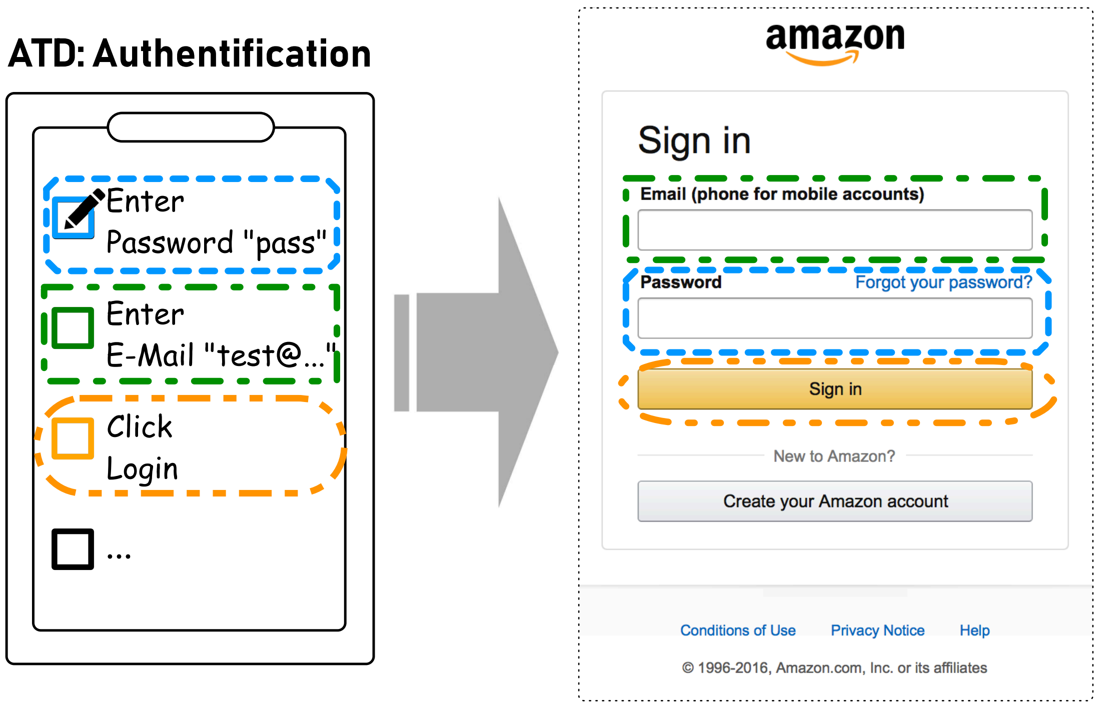


Figure 3.4: Example for interpreting ATDs for crawling. The highlighted boxes indicate the valid target matching between the ATD target and a UI element.

in an appropriate application programming interface (API). The semantic interpretation layer builds upon this interface and is thus independent of the platform.

3.2 Semantic Element Analysis

Up to this point, we have presented platform specific analysis frameworks that extract the potential targets for UI actions as well as their descriptive labels. In other words, the previous steps present the technical foundation on which we can execute topic-driven tests, semantic comparisons, and knowledge transfer. In the next step, we discuss the central part of our technique, the semantic analysis and matching procedures from Figure 1.2. After having extract suitable targets with the previous analysis, we analysis the semantic properties of features in order to guide testing.

Figure 3.4 is an extension of the earlier presented login screen. The example now includes a machine executable list of instructions (ATDs) that cannot be executed by syntactic means. The instructions were not written for the AUT, but are vague in nature

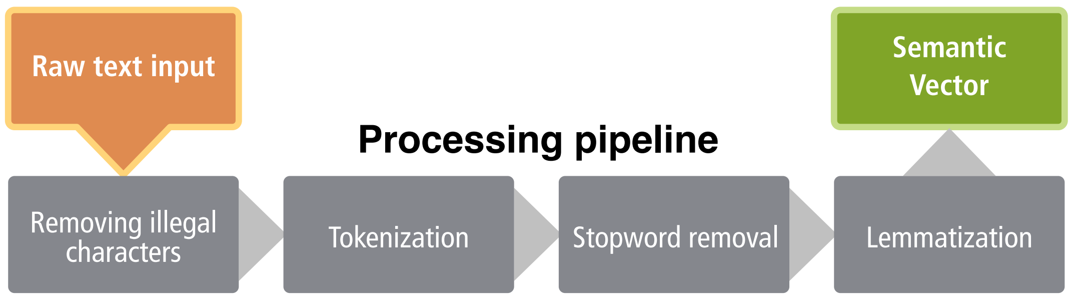


Figure 3.5: Semantic analysis pipeline implemented in MALLET. Starting with the raw text input, each text is transformed into a vector

and just describe a typical login. So far, existing frameworks cannot interpret these instructions. Topic-driven testing now links the semantic meaning to the actual GUI. Each label is first pre-processed (i.e. 'Text Sanitization') to have a comparable and stable baseline for each string in the AUT, see Section 3.2.1. Afterwards, a semantic vector representing the label is computed, which can be used to express semantic similarities, see Section 4.1.

3.2.1 Text Sanitization

Before the labels of an ATD can be matched to their counterpart in the AUT based on semantic similarity, the labels extracted from the UI have to be pre-processed in order to sanitize the given input. Of course, the same procedure can be applied on an ATD, especially the target descriptor, in case the pre-processing is not done in advance. The extracted descriptive label text is typically not purely in natural language, but may also contain illegal characters (e.g. special Unicode characters), line breaks, or the like. Because of this, each text is pre-processed according to standard NLP operations [56]. Figure 3.5 shows a general process pipeline that we integrated into the MAchine Learning for LanguagE Toolkit (Mallet) framework, a popular machine learning toolkit. Each step in Figure 3.5 is written as a Mallet pipe allowing for a fast processing of the natural language input. Algorithm 2 presents all pre-processing steps. In the first step all illegal characters are replaced by a whitespace character to get rid of invalid/incomplete HTML-tags (e.g. '<',/) or special symbols (e.g. special Unicode characters). After tokenizing the given string of the label into individual words.

In order to follow the established grammatical rules of the language, the processed

```
Data: a: set of UI elements, word2vec: WordModel
Result: preparedStrings, Set of valid words
foreach str ← UIElement.textualContent do
    // Replace non-literal chars with whitespace
    sstr ← sanitize(str)
    stem ← foreach word ← tokenize(sstr) do
        yield stemWord(word)
    end
    noSW ← removeStopWords(stem)
    yield noSW.filter(word → word2vec.contains(noSW))
end
```

Algorithm 2: Word pre-processing algorithm for given UI elements. After removing illegal characters, each word is reduced to its base form using stemming and lemmatization, the stopwords and unknown words are removed.

labels use different forms of a word, such as *proceed, proceeds*, or *proceeding*. Our semantic analysis ignores the order of the words. It is thus useful to reduce the inflectional form of each given word (and also of derivationally related forms) by morphological stemming [57] and lemmatization [36] to a common base. The methodology follows the assumption that the inflectional form introduced by the grammar of a language and even the order of words is of minor importance for a human to understand the meaning of a given label (bag-of-words assumptions). As a consequence, words like *am, are*, or *is* are reduced to their base form *be*. In the last pre-processing step, we filter out the most common words of the language (stopword removal) and remove unknown or invalid words, which are not in the corpus of our word vector model.

Stemming usually refers to a crude heuristic process that chops off the ends of words in the hope of achieving this goal correctly most of the time, and often includes the removal of derivational affixes. *Lemmatization* usually refers to doing things properly with the use of a vocabulary and morphological analysis of words, normally aiming to remove inflectional endings only and to return the base or dictionary form of a word, which is known as the lemma. If confronted with the token 'saw', stemming might return just 's', whereas lemmatization would attempt to return either 'see' or 'saw' depending on whether the use of the token was as a verb or a noun. The two may also differ in that stemming most commonly collapses derivationally related words, whereas lemmatization commonly only collapses the different inflectional forms of a lemma. Linguistic processing for stemming or lemmatization is often done by an additional plug-in component to the indexing process, and a number of such components exist, both commercial and open-source.

3.2.2 Semantic Analysis

After the individual labels have been pre-processed, we now have the technical means to extract on each UI screen in the AUT all interactive elements together with their sanitized descriptive labels. Central to topic-driven testing is the ability to infuse these items with a semantic meaning. A word vector model (word2vec), as presented in Section 2.2.3, can be used to derive the semantic meaning of words and sentences. Our goal is to infuse the interactive elements of the pages with their semantic meaning. To do so, the accompanying labels are processed and we extract the word vector of each (pre-processed) word from a given word vector model.

Typically, these `word2vec` are trained on pre-labeled documents, which requires a supervised training phase. In our case, every feature requires a training set of a couple of hundred web pages [44] that have to be (manually) labeled. On top of that, this type of model requires retraining as soon as a new feature is added. In order to facilitate topic-driven testing as a testing procedure without a high setup cost, this kind of model is not appropriate. An alternative are models trained on unlabeled pages. While the training phase may be unsupervised, the required training data (i.e. the set of documents) has to be larger by multiple orders of magnitude. This amount of data is hard to come by and time consuming to train even on specialized hardware. Word vector models have become pretty popular in recent years and it has been shown that models trained on a large set of unspecific documents are even superior to models trained on a very specific, but relatively small data set [55].

We forego this expensive process and instead choose the googleNewsVector model [55] off-the-shelf, henceforth just referred to as the *`word2vec` model*. This model has been trained on non-domain-specific documents including more than three billion words. Non-domain-specific models have the advantage to be easily applicable to any arbitrary application domain. Meanwhile, *specialized models*, e.g. as the one presented by Lin et al. [44], need to be retrained for each new domain and even features.

The labels of the interactive elements on the pages can now be associated with their corresponding word vectors in the `word2vec` model, which allows us to calculate the *semantic similarity* between the labels of different applications.

4 Topic Driven Testing

Parts of this chapter have been published in Rau [64], Rau et al. [67], and Avdiienko et al. [7].

So far, we have discussed the technical background to extract the interactive UI elements and their semantic meaning both from Android apps as well as applications accessible through a browser interface. The semantic meaning of the elements is encoded in the corresponding word vectors. In this chapter, we discuss how this technical base can be used to calculate the similarity between UI elements in different applications (Section 4.1), and apply this concept on the top five hundred pages of the Quantcast index [62] that lists the most popular pages Section 4.2. On this set of representative applications we will analyze the feasibility of an automated feature learning and knowledge transfer that is the essential contribution of topic-driven testing.

4.1 Semantic Similarity of UI Structures

In the previous chapters, we discussed how interactive elements in the graphical user interface of both android apps and web apps can be analyzed together with their semantic meaning that is expressed in the labels in their visual vicinity. The pre-processed labels can now be matched against another natural language target, for instance the UI element labels in another application (knowledge transfer) or we can calculate the similarity between a given command in natural language into a corresponding machine executable statement.

An educated look into common applications reveals that the textual descriptions vary from single words (e.g. address, street, username) to sentences, short paragraphs and sometimes grammatically invalid word combinations. Standardized sentence or sentenced based vector models (i.e. trained on sentences or paragraphs) that can analyze similarities between sentences and paragraphs are ruled out as they can hardly handle invalid structures and the high variance in the used labels (e.g. length, structure) would decrease the efficiency dramatically. Despite its ease of use, *this* is the main reason why

$$\equiv_{sem}(str_1, str_2) = \begin{array}{c} \\ Enter \\ username \\ email \\ address \end{array} \begin{array}{c} \begin{array}{ccc} Insert & username & here \end{array} \\ \begin{bmatrix} \mathbf{0.455887} & 0.254257 & 0.0790675 \\ 0.208744 & \mathbf{1.0} & 0.128865 \\ 0.152182 & 0.397393 & 0.111652 \\ 0.074623 & 0.213060 & \mathbf{0.1640389} \end{bmatrix} \end{array}$$

$$= \begin{bmatrix} 0.455887 \\ 1.0 \\ 0.1640389 \end{bmatrix} = \frac{\sum_{i=0}^{\dim(v)} v_i}{\dim(v)} = 0.5399753$$

Figure 4.1: Sample matrix for calculating the semantic similarity of two strings “Enter username email address” and “Insert username here” using *pair-wise cosine similarity* of two words.

`word2vec` are used in topic-driven testing for further analysis. The `word2vec` models work—as the name already indicates— on the smallest semantic unit in the sentence: *words*. As such, they cannot analyze arbitrary strings (e.g. whole sentences). Instead, we implement a novel similarity to calculate the similarity between labels. In order to do that, we use the `word2vec` model to compute the *cosine similarity* (**??**) between all possible word pairs.

Figure 4.1 shows the sample output for two given sample strings. The cosine similarity of each word pair is expressed in an $n \times m$ matrix, where n and m are the respective lengths of the given strings. We then select the best matching word pairs (ignoring the order of words) under the conditions that every word is only matched once. The resulting points in the vector are summed up and normalized using the dimension of the result vector. In our example, the procedure does not allow that the words “username”, “email”, and “address” are all matched to the word “username” in the second string, even if the resulting normalized sum of the vector would be higher then. Although the order of words is neglected, both strings express a different amount of concepts. By excluding multi-matches, the given method reflects this. The computed value $\equiv_{sem}$ (normalized by dividing it by the dimension of the vector) is in the interval $[-1, 1]$. Again, a value close to -1 indicates highly dissimilar concepts, while values close to 1 indicate a semantic match.

The similarity value $\equiv_{sem}$ serves as a certainty factor that the UI elements in the target application match the wanted feature. The returned list of potential matches is sorted in descending order. Elements with a high semantic similarity thus are ranked at the top, as they are most likely to match.

As a preliminary optimization, we omit label pairs in which the ratio in the number of words is larger than five; i.e., if the number of words in one label is more than five times the number of words in the second label. We just assume that they are not matching because this ratio is large enough to assume that two labels convey a different meaning. The applied normalization reduces the calculated value drastically, if the length difference increases. Sorting the results according to the similarity causes the results to be ranked at the bottom. This optimization allows for a faster calculation and also circumvents to match items to large text blocks throughout the target application.

The presented algorithm allows us to finally calculate the semantic similarity between two given UI elements and lays the foundation for any topic-driven testing method.

4.2 Semantic Analysis of Real World Applications

In contrast to traditional testing methods, topic-driven testing is focused on linking interactive elements together with their descriptive labels to a semantic meaning. Before we get started to integrate the feature matching into testing procedures and frameworks, we now take a look into the data of real world web applications and check how their functionality is encoded into semantic properties. The previous Chapter 3 introduced the technical methods on how the semantic meaning is extracted from web applications and Android apps. Now we apply the same technique on the top five hundred web pages out of the Quantcast index [62]. Quantcast is a web ranking page that lists web pages according to the number of monthly visitor counts. We inspect the top U.S. web pages, which have a monthly active user base of one million to eighty million people.

Now we apply the semantic analysis (Section 3.2) on these pages and extract the word vectors corresponding to the elements with the presented technique. We are interested in the feasibility, correctness, and usefulness of topic-driven testing when we analyze the GUI on a large scale, e.g. on a large number of web applications. In a nutshell, we try to assess the following research questions:

RQ 1 (Feasibility) *Have real world applications common language features exploitable for testing?*

RQ 2 (Correctness) *Can we identify features with semantic means by analyzing the distance between word vectors, proximity in the word vector space, or clustering techniques?*

RQ 3 (Novelty) *How do existing syntactic metrics compare against semantic metrics for mapping features?*

By answering RQ 1, we get a basic understanding on whether the procedure is feasible for testing. In order to interpret test snippets, transfer knowledge, or transfer features, these entities need to be present in a significant part of the application set. RQ 2 investigates if semantic similarity is the correct technique to identify features. Finally, we compare ourselves against syntactic techniques to show that semantic means are actually superior (RQ 3).

4.2.1 Feasibility of Topic-Driven Testing

The word vector model used in this dissertation models each word as *300-dimensional vector*. In order to visualize this high-dimensional data, it is necessary to transform the vectors into a two-dimensional representation. To do so, we use the t-Distributed Stochastic Neighbor Embedding (t-SNE) technique developed by Maaten and Hinton [80]. By applying a Barnes-Hut approximation of the data points, this method transforms the set of *300-dimensional word vectors* into a two-dimensional set of points.

Figure 4.2 shows a visualization of all the descriptive texts of the top five hundred Quantcast pages in the U.S., combined with a density analysis. A detailed list of the used evaluation subjects is available in the appendix (see Appendix A) together with an estimation of monthly visitors. About thirty applications could not be evaluated, due to an incompatibility with WEBMATE—e.g. because they use outdated JavaScript libraries that are not compatible with WEBMATE. Still the data set is large enough for an analysis. The chart is pretty dense and visualizes the word vectors of more than three hundred thousand words.

The individual dots in the chart each detail a single word from one of the web pages. We checked the dataset to ensure that the visualized data represents a proportional spatial distance to the original data set. In other words, t-SNE transformations keep local similarities and do a decent job do preserve the global structure of data (clusterization). Words close to each other in the original dataset are mapped to nearby points in the visualization.

The dataset shows that certain words in the vector space occur often. The yellow cloud structures surrounding the words represent a the results of a kernel density estimation. Their shape and size indicates the the density of the data points within. Take the three almost circular shapes at the very top of Figure 4.2 (highlighted by the red circle). The structures each contain many close-by points. The words represented by the

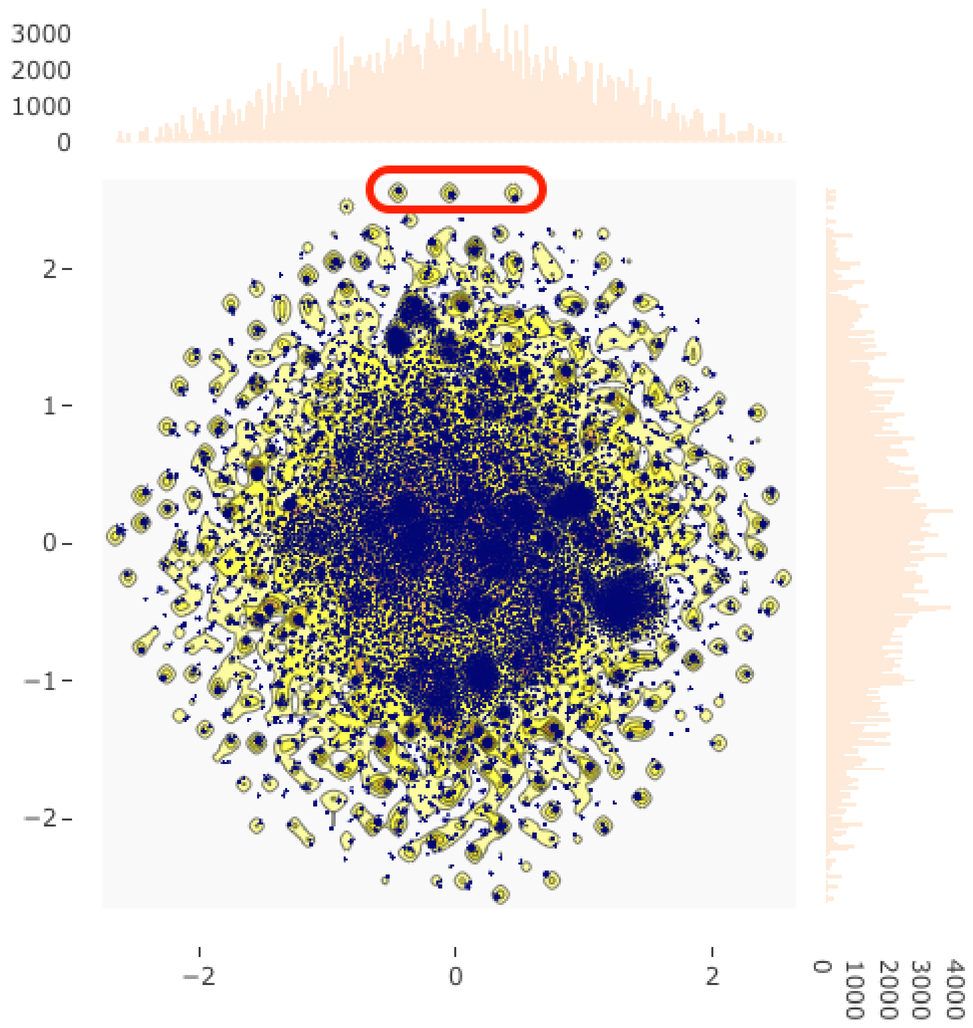


Figure 4.2: Word vector visualization of the Quantcast top 500 web pages after applying a t-SNE-transformation to reduce the high-dimensional vector to a two-dimensional vector space. The *x*-axis and *y*-axis show the respective vector values after the transformation. The bar-plots on the top and right show the absolute number count at this point.

vectors occur multiple times in the five hundred web applications. At this moment, this observation cannot be translated into any insignificance of these words.

We can also observe that their exist other larger structures in the visualized data that indicate less dense areas. Since t-SNE-transformation tries to preserve the local similarities, these less dense areas might either occur to the linear embeddings or there might exist semantic structures that are less common throughout the analyzed application set. The Quantcast index contains applications coming from a multitude of domains, ranging from search engines, coding platforms, social platforms, and more. Not all offer the same functionality. As a consequence there exist rather unique semantic entities.

On the other hand, we can see larger groups of data points that are bundled together especially in the center, lower center, and lower right part of the chart. Even in this crowded chart with lots of data points they are clearly identifiable. These are not just repeated single words, but instead groups of words that are semantically related. We can see that there exist *semantic entities* throughout the applications, i.e. *topics*. We can also see that they are often not expressed with the same word, otherwise the data structures would always look like the circular shapes we have seen earlier. Like single UI elements do often not represent an application feature, individual words are also not indicators for functions. Still, the data indicates general semantic properties that are present in more than one application. Transferring these properties and use them for testing is the incentive for topic-driven testing.

The principle of topic-driven testing is thus backed by the data and allows us to answer RQ 1 on feasibility:

RQ 1 (Feasibility) *Have real world applications common language features exploitable for testing?*

- ☞ *The data indicates that there exist universal semantic structures (clusters) that can be representatives for features*
- ☞ *We can use semantic means to identify features and use them for testing*
- ☞ *There also exist localized semantic structures that only exist in a minority of the applications. The corresponding features may not be easily testable or a require additional effort.*

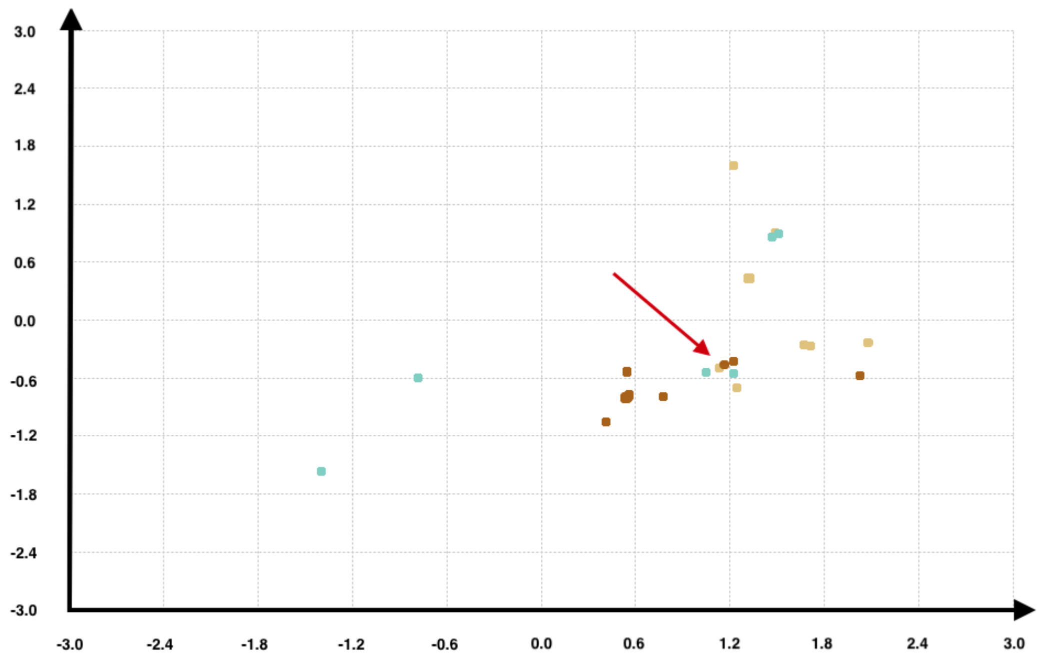


Figure 4.3: Word vector visualization using a t-SNE transformation for the features ‘*configuration*’ (brown), ‘*account creation*’ (light blue), and ‘*authentication*’ (light brown).

4.2.2 Semantic Feature Identification

In order to get a grasp of how semantic features materialize in this word vector space, we now check for specific keywords that are representatives for certain functions. We pick features that are common for a large majority of applications and check how they are semantically presented in the dataset and which words are closest. Figure 4.3 shows the word distribution for the features ‘*configuration*’ that is used to change account settings, ‘*authentication*’, i.e. functionality related to logging in, and ‘*account creation*’, i.e. signing up for an account. The displayed vector space is the same as before.

The data indicates rather dense distributions for each feature for both ‘authentication’ (beige solid circle), and ‘configuration’ (dark brown dotted circle). Considering the previously crowded vector space, this shows a pretty clear cut picture. Both functions are mapped to close rather specific areas. Also the distance, between the two areas is not significant. Right in the middle between those areas (see arrow) are words such as *username, user*, and *password*. These terms are obviously not bound to either function, but instead present in both. The *googleNewsVector* shows that the terms occur in a relevant context to both features.

The ‘account creation’ function shows erratic data entries. Keywords such as email, or password are bundled together, but typically the account creation also integrates captures, first name, and surname. In the vector space these terms are located in different areas. For the later testing, this missing cohesion is of minor importance, since the individual labels (and thus the terms) are matched one-by-one. Nevertheless, the other two features show that the relevant terms are also semantically close and we can actually learn features by grouping terms due to their position in the vector space.

As a side effect of the t-SNE transformation the distance between the vectors might be different from the initial high-dimensional vector space. The visual clues indicate that we can use arbitrary distance methods (e.g. Euclidean distance) to calculate the semantic similarity. Even if the relations are preserved, we use cosine similarity on the original data. Literature [61] shows that cosine similarity is better suited to express semantic similarities.
This essentially answers RQ 2 to a certain extent:

RQ 2 (Correctness) *Can we identify features with semantic means by analyzing the distance between word vectors, proximity in the word vector space, or clustering techniques?*

☞ *The data indicates that keywords expressing certain functions are semantically close and appear in the same area in the vector space.*

☞ *The distance between word vectors can be used to predict features across applications.*

4.2.3 Syntactic Mapping Procedures

Mapping features is an essential step for a further integration into testing. We now analyze the data set to verify why syntactic matching methods cannot be exploited for mapping procedures. That is, why using semantic similarity can improve state-of-the-art testing.

Figure 4.4 is again showing words of the Quantcast top five hundred web pages. Again, the set of words was reduced to relevant words for the functions ‘*configuration*’, ‘*account creation*’, and ‘*authentication*’. In Figure 4.3, the same words were displayed, but only with their word vector. Figure 4.4 now includes the two dimensional vector that represents the element position (in this case the position of the label, not the interactive element). In a direct comparison, one can immediately see significant differences. While the word vectors in Figure 4.3 present us with very tight groups for each function,

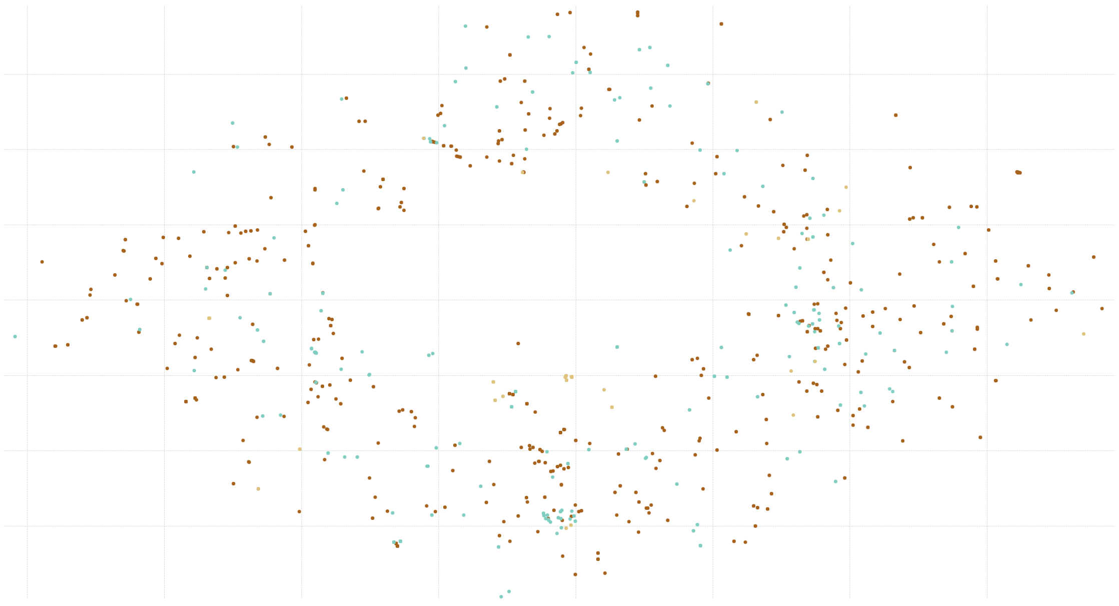
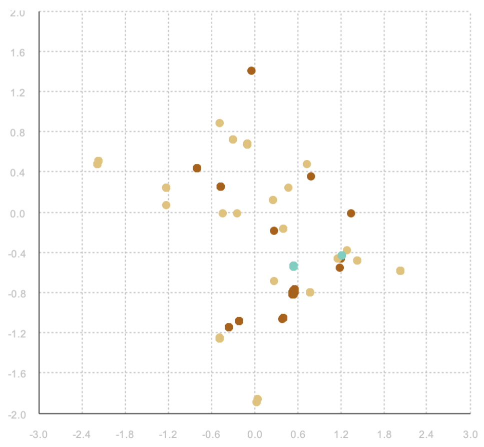


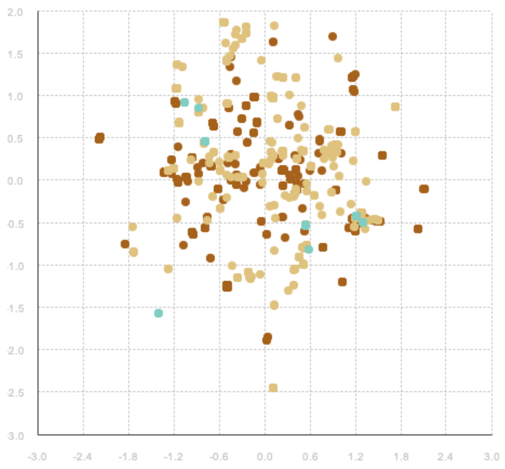
Figure 4.4: Distribution of word vectors including the display position (x, y-coordinates) of the features '*configuration*', '*account creation*', and '*authentication*'.

including the x and y coordinates into the vector leads to rather unpredictable results. The vectors are now distributed all over the vector space and also the features are intermingled. Clustering and distance calculations cannot produce useful prediction results. On a side notice: one can notice the dubious large empty space right in the middle of the vector space. This space is an artifact of the t-SNE transformation and does not necessarily indicate a large part of the UI screen where keywords from our samples do not occur. The t-SNE transformation does preserve the distance between the elements and the overall distribution. As a consequence, we cannot clearly pinpoint the empty region to an equally empty region in the GUI.

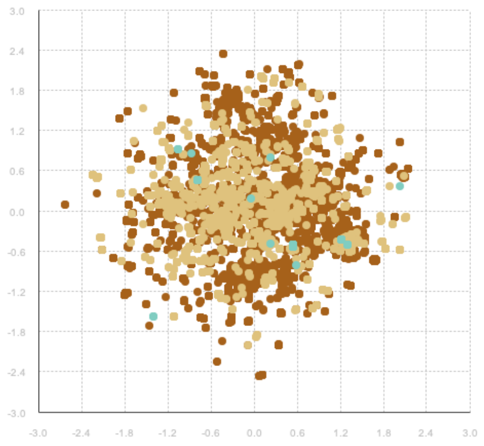
Alternatively, one can try to match the strings of the appropriate keywords directly. The problem with this technique is that even small differences in the words do have a strong impact on the effectiveness, i.e. in this case more precisely the recall. Besides, the presented basic NLP measures, such as stemming, offer already a wide variance of tools. Still, small differences, e.g. a capitalization of words ('Sign In' vs. 'sign in'), additional characters such as hyphens that link some words, or changes to prepositions ('Sign In' vs. 'Sign On') can make the mapping difficult. Tackling each syntactic difference individually might be possible, though. To measure the effect of small syntactic differences, we investigate the Levenshtein distance to the keywords referenced by the same features as above ('*configuration*', '*account creation*', and '*authentication*'). The



(a) Levensthein Distance 1



(b) Levensthein Distance 2



(c) Levensthein Distance 3

Figure 4.5: Word distribution for the features ‘*configuration*’, ‘*account creation*’, and ‘*authentication*’. The sub-figures highlight for each feature the word distribution of the words in the corresponding Levensthein distance.

Levenshtein distance measures the edit distance between two given character sequences, i.e. the number of insertions, deletions or character changes necessary to transform one string into another.

Figure 4.5 displays the word vector space of the same keywords as before, but also highlights words with a small Levenshtein distance. The sub-figures show an increasing Levenshtein distance starting from one to three. It is interesting to see how many words are in close syntactic proximity. The number of potential targets grows exponentially with an increasing Levenshtein distance. While the data does not show that semantic measures are superior to syntactic means per se, we can now estimate the number potential targets if we apply syntactic means. If we implement an algorithm that is robust against three character changes, the data in Figure 4.5c indicates a large number of potential targets—more than thirty thousand.

Another interesting observation is the comparison between semantic similarity and syntactic changes to the underlying words. As already mentioned, the distribution of word vectors increases dramatically, if the Levenshtein distance is increased. It shows that the language is diverse and that the spelling stands in no relation to the semantic meaning. The word vectors express semantic similarities and the t-SNE transformation preservers the overall relations. The erratic distribution in Figure 4.5 can thus be seen as a confirmation that the semantic correlation is not bound to the spelling. The observations from Figure 4.4 and Figure 4.5 allow us to answer RQ 3 about the capabilities of semantic analysis compared to syntactic measures.

RQ 3 (Novelty) *How do existing syntactic metrics compare against semantic metrics for mapping features?*

☞ *Syntactic measures are insufficient for mapping features. The number of potential targets increases exponentially with syntactic changes to the underlying test set.*

☞ *The position of features and their interactive elements in the graphical user interface is not consistent across applications.*

☞ *Semantic measures are superior in identifying features and mapping them across applications.*

4.3 Conclusion

As depicted in Figure 1.2, topic-driven testing consists of three central parts. The previous chapter Chapter 3 presented the technical foundation to extract features from an

AUT and infer a semantic meaning from their descriptive labels. These were the two bottom layers of the overview figure. In this chapter, we analyzed a large group of five hundred real world web applications to show the feasibility of using semantic word vectors for feature identification and for a potential transfer across applications. This is the topmost layer.

We investigated more than three hundred thousand words in five hundred popular real world web applications and identify groups of elements that are clustered in close proximity and exist in a majority of applications. These clusters of semantically close words are indicators that there are common features in these applications. The presented data backs our assumption that we can exploit semantic text similarity to identify features and their corresponding UI elements. This is essential to guide testing, transfer application specific knowledge, or learn from existing test suites. The data also indicates that the keywords that represent certain features are mapped to nearby points. This allows to learn common features from a set of applications. On the other hand, the data also indicates sparse areas, i.e. regions in the vector space that have words only present in a minority of the applications. Considering the idea that we want to learn how to test common features (be that from test scripts or use case descriptions), this means that there are rather unique features in some applications. While not untestable per se, we would require additional effort to specify how these features can be tested. And this effort only pays of for single applications, not for a majority.

The following chapter discusses how the gained observation can be applied and shows that semantic concepts can be leveraged to re-use existing test suites, or translate simple natural language test specifications into machine executable statements to improve testing or learn application behavior.

5 Topic Driven Testing in the Web

Parts of this chapter have been published in Rau et al. [66,67,69].

Now it is time to put the theory into practice. In this chapter, we apply topic-driven testing on web applications in order to improve and generalize existing testing procedures. The input to topic-driven testing can be rather generic, i.e. everything that carries a semantic meaning to guide testing. In this chapter, we show how we can use existing system web tests (written in SELENIUM) to learn features that exist in the AUT, to guide web testing in new applications, and even partially transfer existing tests to new applications. Essentially the technique can be used on arbitrary applications with a GUI, but in this chapter we concentrate our analysis on web applications. We implemented the technique into a self-contained framework called ATTABOY . ATTABOY takes an existing SELENIUM test suite for one web application *A* and executes it on another target web application *B*.

Identifying features in web applications and then apply this knowledge to test/guide testing in other applications allows for better testing with very low additional effort. Writing one test for an application, translates into the possibility to execute it on a multitude of applications. It also lowers the initial effort of testing an application. Since developing and testing applications are often done by different people in a company, the second part can be simplified by this knowledge transfer. Having the intentions of topic-driven testing in mind, we also want to see, if there exist common implicit specifications that humans expect to see in certain systems. In topic-driven testing, this assumption is somewhat integrated with a tacit understanding. We assume that the target application has the same underlying semantic structure. Otherwise, transferring tests would not be possible by construction.

The core of our presented technique consists of two steps as depicted in Figure 5.1. In the *feature identification phase* (technical background presented in Section 3.1), we identify essential features given an existing SELENIUM test suite. The UI elements, on

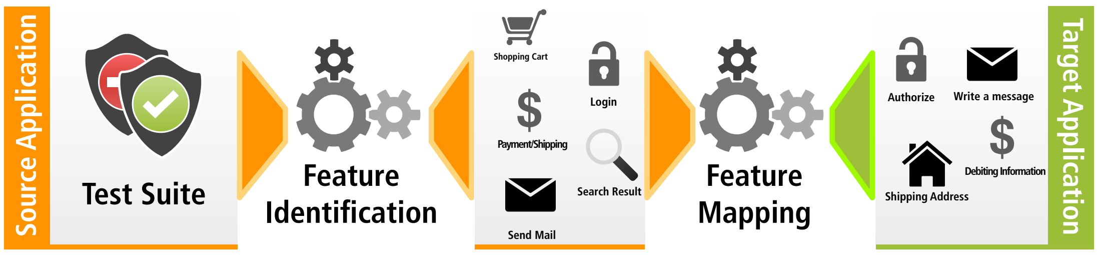


Figure 5.1: Generating tests from tests of other apps. After extracting features (as GUI elements and textual labels) from an existing application, we match these features semantically while exploring a new application, As a result, we obtain a mapping between features that helps adapting tests from the existing application to the new one.

which the test suite executed commands are relevant for a feature together with their describing labels. After extracting the textual content of these elements, we can *match these features across applications* using semantic text similarity as a metric (as presented in Section 4.1). The matched features then guide test generation on new applications in the test set.

In order to identify features in a target application *B*, we use the discovered elements together with their describing labels. The presented method does not require any other test suite to generate an application model for the target application *B*, but only requires a number of states, i.e. DOM-trees, which can be for instance generated from the traversed states while guiding a crawler through the application.

The earlier Chapter 3 described how for every interactive element the descriptive label is extracted and how the semantic meaning is expressed as a word vector. Given a state of an application *A*, we now check each state in the target for potential labels describing the same semantic concepts using semantic text similarity as a metric. The describing labels in the target application contain the same natural language content a human would see and we try to identify the closest semantic entity to the label in the target application by computing the word-wise cosine similarity.

We tested ATTABOY in an emprical study on 12 web applications each tested with a SELENIUM test suite. We transferred these test suites to the other applications in the same domain to evaluate the feasibility of the feature transfer, the possible speed-up compared to random crawlers, and for the success of the test transfer. The study offers initial evidence of the capabilities of topic-driven testing and is very successful in identifying features across applications.

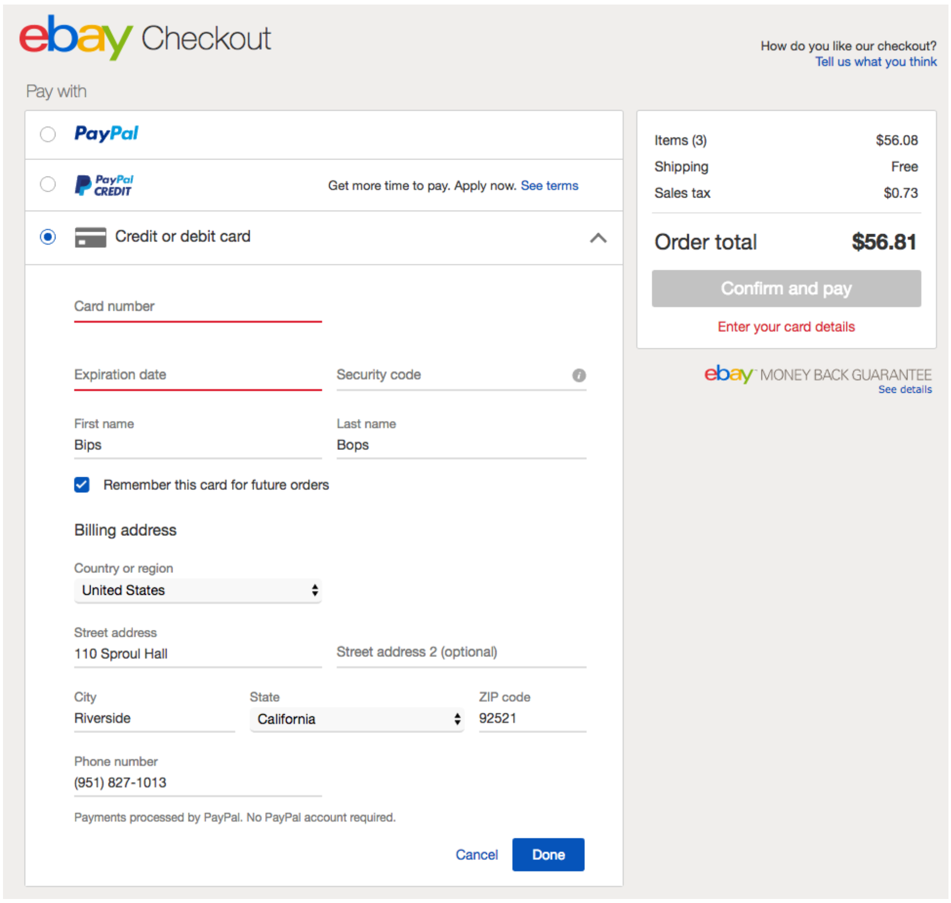
Summarized, the empirical study yields the following results:

1. Our technique maps features between source and target applications with a recall of 75% and a precision of 83%.
2. Using our most conservative estimate, this mapping speeds up exploration from a single state on average by 740%.
3. If the searched functionality is deeper in the application, this speedup multiplies across states, yielding a total speedup along the path that can be exponential.
4. All this greatly simplifies test generation for new web applications when tests in the same domain already exist.

5.1 Introduction

As an example of the difficulties computers encounter, assume you want to generate a test that purchases some item on a shopping site like eBay, starting from its home page, `ebay.com`. For the moment, let us assume that a test generator already found out how to select a product, how to place it in a shopping cart, and how to proceed to checkout. Then, the test generator is faced with the payment screen shown in Figure 5.2.

As an experienced human, you would know what to do here—namely fill out the forms, and press one of the highlighted buttons (first, “Done”, then “Confirm and Pay”). For a test generator, filling out such a form is much more difficult. Besides the problem of having appropriate payment data to fill out, it is hard to determine which interaction to perform next. Overall, the eBay checkout site sports *no less than 1,269 user interface elements* in its document object model (DOM). On the screenshot, we already see payment options, feedback forms, guarantee options, cancel options; further below follow address and shipping options, as well as a myriad of photo links to possibly related products. For a computer, with all UI elements being equal, the chance of hitting the “Confirm” button on the first try is less than one in a thousand. Of course, an exhaustive test generator can easily try triggering each of the 1,269 UI elements and spend hours wading through all the legal and promotional material—a commercial site like eBay has on its pages—and eventually determine that the “Confirm” button is the one that opens up new functionality. However, remember that we initially assumed that the test generator already has managed to select a product, placed it in the cart, and proceed to checkout. *Each of these steps is just as difficult to complete as the checkout step discussed above*—the eBay home page itself sports more than 2,268 UI elements, and the


ebay Checkout

How do you like our checkout?
Tell us what you think

Pay with

- ○ PayPal
- ○ PayPal CREDIT — Get more time to pay. Apply now. See terms
- ● Credit or debit card

Card number

Expiration date | Security code

First name: Bips | Last name: Bops

✓ Remember this card for future orders

Billing address

Country or region: United States

Street address: 110 Sproul Hall | Street address 2 (optional)

City: Riverside | State: California | ZIP code: 92521

Phone number: (951) 827-1013

Payments processed by PayPal. No PayPal account required.

Cancel | Done

Items (3)	$56.08
Shipping	Free
Sales tax	$0.73
Order total	$56.81

Confirm and pay

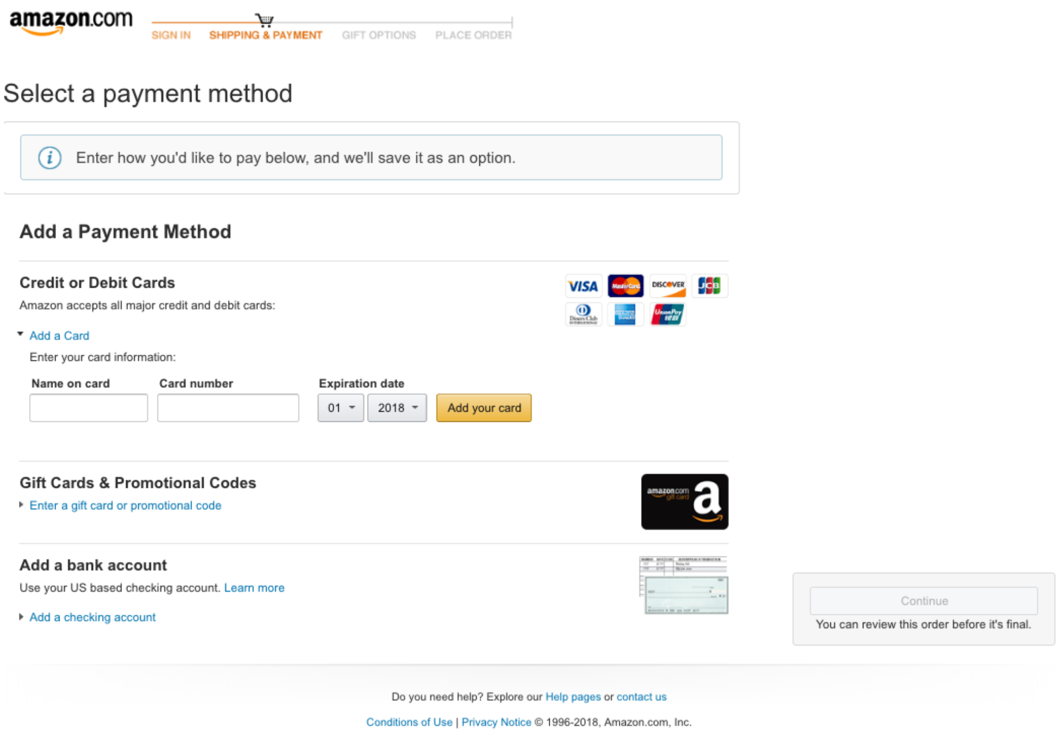
Enter your card details

ebay MONEY BACK GUARANTEE
See details

Figure 5.2: Besides providing fields to enter payment details, this page also provides additional UI elements with additional payment options, guarantees, and offers—1,269 UI elements in total. For a human, it is clear that one of the highlighted buttons ("Confirm") is supposed to be pressed next. But how would a computer know?

low probabilities of following the purchase path multiply with each other. This is what makes discovering "deep" functionality within a web site such as eBay so inherently challenging. A crawler would have to explore thousands to millions of states before reaching new and test-worthy functionality.

For us as humans, such activities are much easier, because we are able to interact with GUIs based on the *semantics* of user interfaces—we *know* what terms like "feedback", "pay", "cancel", "special offer" mean, and we choose the ones that are closest to our goals. We also have *experience* with similar sites; if we have shopped once on, say, Amazon.com, we know how to repeat the process on other shopping sites, where we can re-identify familiar concepts like "cart", "checkout", or "payment". Indeed, the payment



Select a payment method

Enter how you'd like to pay below, and we'll save it as an option.

Add a Payment Method

Credit or Debit Cards

Amazon accepts all major credit and debit cards:

Add a Card

Enter your card information:

Name on card | Card number | Expiration date 01 2018 | Add your card

Gift Cards & Promotional Codes

Enter a gift card or promotional code

Add a bank account

Use your US based checking account. Learn more

Add a checking account

Continue

You can review this order before it's final.

Do you need help? Explore our Help pages or contact us

Conditions of Use | Privacy Notice © 1996-2018, Amazon.com, Inc.

Figure 5.3: Can we use an existing test for this web page to facilitate test generation of the eBay page in Figure 5.2?

page for Amazon (Figure 5.3) is not that different from the payment page on eBay; concepts like “name on card” or “card number” are easily found on both. (Amazon, however, requests possible gift options after card details before finally allowing the user to place the order.)

Now assume that for the Amazon payment in Figure 5.3, *we already have an automated test* that selects a product, adds it to the cart, proceeds to check out, and finally pays for the product. *Would it be possible to leverage the knowledge from this Amazon test to generate a test for eBay?* By extracting the general sequence of actions (selecting, adding to a cart, checking out, paying) from the existing Amazon test case, we could *guide* test generation for the eBay site towards those actions already found on the Amazon test suite.

As sketched in Figure 5.1, we start with an automated test for a *source* application (say, Amazon). Each step in the test induces a state in the source application. From these states, we extract the *semantic features*—that is, the user interface elements the test interacts with together with their textual labels. While generating a test for a new target web application (say, eBay), we attempt to *match* these features while exploring the application, determining the *semantic similarity* between the labels captured in the source and the labels found in the target. We then *guide* exploration towards the best matches, quickly discovering new and deep functionality, which is then made available for test generation. The final result is a *mapping* of the features found in the source to the features found in the target, allowing for easy (and often even fully automatic) adaptation of the tests created for the source application onto the new target application.

To the best of our knowledge, ours is the first work that allows for generating tests in new applications by learning from existing tests of other applications, which only share the general application domain. The results open the door towards automatic adaptation of tests across applications: *Having written a "pay" test once for one web shop, developers can automatically apply it again and again for one web shop after another.*

In the previous section, we have seen how the natural language texts in a GUI are linked to the underlying functionality and features of applications. Furthermore, the descriptive texts are semantically similar across applications. In other words, we can take a user usable function from one application, extract their descriptions, and transfer this knowledge across applications using the proposed topic-driven testing approach.

Applied to web application testing, this opens interesting research angles. First of all, we can guide testing towards desired functionality instead of testing it blindly, e.g. by using random testing approaches. Second, we can observe application behavior for an application *A* and check if other applications behave the same if tested the same way.

In this chapter, we present ATTABOY, a combined mining and test guiding approach. Figure 5.4 shows a short overview on how it works. ATTABOY mines a given web testing suite (Section 5.2), say for testing `Amazon`, and generates a state model that expresses the application behavior. In a second step, it takes a start URL of another application, say `ebay`, and mimics the recorded test commands.

In order to show the effectiveness of TDT, we conducted three empirical studies that answer the following research questions:

RQ 4 (Accuracy) *Can we identify web features using semantic UI element similarity?*

This is one of the core contributions of our technique: Mapping features from the source application towards the features of the target application. We evaluate the accuracy of the mapping; the better the accuracy, the more effective is the guidance for subsequent exploration and test generation.

RQ 5 (Impact on web crawling) *Can automated feature identification be used to improve web crawling?*

This is the main application of our technique: What difference does it make to a crawler if guidance exists? We conservatively evaluate the worst case speedup of guided crawling vs. non-guided crawling.

RQ 6 (Test Transfer) *Can a given* SELENIUM *test suite be transferred to the test suite of another application within the same domain?*

We will investigate on three application domains, if we can re-use existing tests and transfer them to other applications to test the same behavior. This allows to simplify test generation, and lower the initial cost for testing an application.

RQ 7 (Test Generation) *Can we transfer automatically generated test suite across applications?*

Automatically testing an application faces the aforementioned challenges of generating valid inputs and covering relevant functions. For certain applications the test creation might still be simpler than for others, maybe due to a simpler structure, less noise, or other factors. We analyze our data set if we can transfer automatically generated test suites.

RQ 8 (Generality) *How similar are web applications regarding their features and feedback mechanisms?*

If web applications show similar functions and behavior that can be exploited for testing, this increases the generality of topic-driven testing. A high generality indicates a high pay-off, since the principle can be applied on a wide variety of applications.

The conducted study goes beyond the analysis presented in Chapter 4 and is more specialized on concrete testable use cases.

5.2 Mining Test Cases

ATTABOY is leveraging a given SELENIUM test suite to generate test traces for the application under test. Our prototype features an extended version of a standard SELENIUM server, allowing us to intercept given commands (i.e. in terms of HTTP requests given to the SELENIUM grid). ATTABOY records the commands and applies the presented information retrieval techniques [20] to extract the current state of the application as currently shown in the web driver instance. The test itself is not changed. Accordingly, test traces consist of single *states* of an application, connected by sequences of SELENIUM commands. Such a state represents a single page of the application. The structure

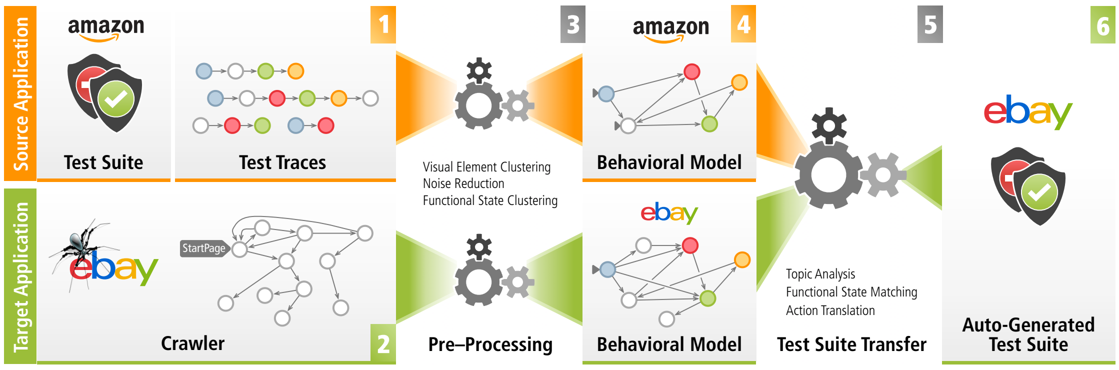


Figure 5.4: **ATTABOY Process Pipeline** for automatic test suite generation. A given reference test suite is executed (1) and processed (3). Meanwhile a random test suite for the target application is created (2) and processed as well. Afterwards the resulting behavioral models (4) are transferred by matching the topic set of each functional state (5). In the last step the action translation is performed to generate the final target test suite (6).

of a page is specified in an underlying Document Object Model (DOM) in which every element of the page is expressed as a node. As already mentioned, this tree structure contains one node for every single element on the page, together with the node content, its styling properties (e.g. CSS classes), visibility information or likewise. Figure 5.4 presents an overview on ATTABOY's process pipeline.

Considering the initial example of buying a product on Amazon, the sample test trace consists of going to the landing page, searching for a product, selecting a product, adding it to the shopping cart, clicking on 'Proceed to Checkout', logging in (entering a valid 'username' & 'password' combination) and confirming the payment and shipping options. The consistency of the application is tested by asserting the presence of certain key elements in the intermediate pages. A new state is extracted whenever a user action is executed on the browser under test (i.e. a non-native SELENIUM action like *implicitWait*, *findElement*).

After the test traces have been generated, ATTABOY pre-processes the traces into a behavioral model by applying visual element clustering (see Section 2.1.2), noise reduction (Section 2.1.3) and functional state clustering (Section 5.2.1). After this pre-processing phase, we extract the main topics describing the content and functionality of a page (Section 3.2). Mapping the topic model of the functional states against another test suite (either an automatically generated or another SELENIUM based one) allows us to automatically transfer the tests across applications.

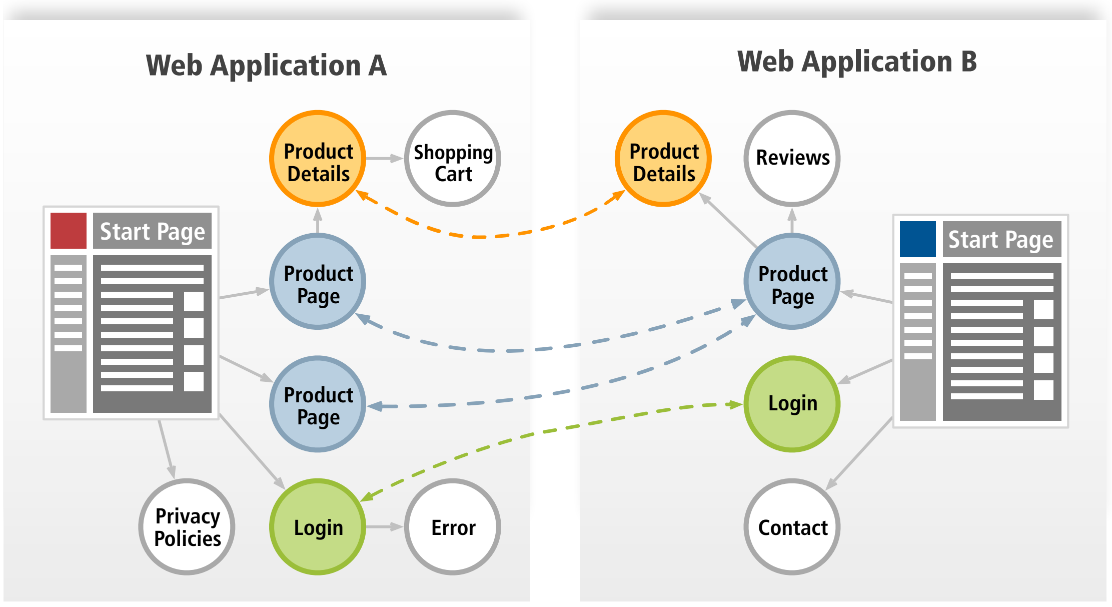


Figure 5.5: Simplified functional state overview on two eCommerce web applications. States with equal labels are clustered and represent the same functional state. Dashed arrows indicate a mapping to functional similar states. Nodes are labeled with their main topics.

5.2.1 Functional State Clustering

Functional state clustering is something we did not discuss so far. Instead, we focused on UI elements and mapping their descriptive labels. This method allows to guide a testing procedure that is focused on following a certain sequence of commands. In order to transfer a test suite, we additionally have to check if the resulting state is the same (or rather semantically similar) to the original application state. Computing the semantic similarity between all UI elements to derive such a similarity measure would be too costly. Our ebay example already features more than one thousand UI elements. Furthermore, the real world applications are very dynamic and even those parts, which we cannot neglect as noise.

Despite the UI element analysis to guide testing, we therefore extract the overall topics in each state using LDA. LDA returns for each state a list of words that describe a predefined number of topics (see Section 2.2.1 for details). If this list of describing words is similar for multiple states in say the source application, these states are clustered. Figure 5.5 shows a simplified graph of two sample applications. States with same topics can be grouped. When we later check for state similarity across applications, we will take these word lists for matching.

In order to extract topics from a single web page with LDA, we first have to divide a page into different segments consisting of multiple elements. Take a web element representing a button that is labeled with 'OK' for instance. Although we can technically extract the topic of this label, LDA is more appropriate to extract topics of larger texts. Instead of applying topic extraction on every single element, we have to take its context into consideration. This is where page segmentation is used again: Extracting blocks and feeding their textual content into LDA generates a list of topics. Each topic consists of a ranked list of words. The MALLET[1] framework [50], a popular Java-based machine learning tool kit, features a very fast and scalable LDA implementation using Gibbs-sampling. This method allows us to extract a fixed number of topics for any web page. The raw text content of the non-noisy blocks is cleaned from illegal characters (e.g. new line characters or unknown symbols) before *tokenization* forms useful semantic units for further processing. In the *stopword removal* step, the most common words in the language (e.g. “a”, “for”, “I”) are removed, allowing the later topic extraction to focus on actual keywords. Finally, the LDA based topic extraction takes place and returns a number of tokens for each state. However, topic extraction on single states has a huge disadvantage. Consider two knowledge base articles presenting two different countries. Topic extraction will extract a number of tokens in each page, heavily influenced by the actual content of the page, but less by the underlying functionality. Although the two sample pages still share common topics, like economic status or population, an article describing a certain person will not match anymore. To circumvent this ‘overfitting’, we apply functional state clustering. We cluster all states of a web application, collect the LDA topics for each state and return the intersection of the describing words. The resulting topic set is an abstraction of all states within a cluster.

Let us again consider an eCommerce application. We often find several states representing the same functional meaning. This may be, for instance, states presenting a list of products or a product description. In other words, they provide a common functionality, which we aim to group together. Every produced cluster contains a set of states which are functional similar and henceforth referred to as *functional states*. We cluster states using a cluster algorithm with a custom distance measure. For that purpose, we integrate the agglomerative clustering algorithm provided by LingPipe[2]. The functional states can later on be transferred to other applications in the same domain.

Assuming that functional states share a common layout, they also share the same set of styling attributes. To style the layout of a website one usually uses Cascading

[1]MAchine Learning for LanguagE Toolkit

[2]`http://alias-i.com/lingpipe/`

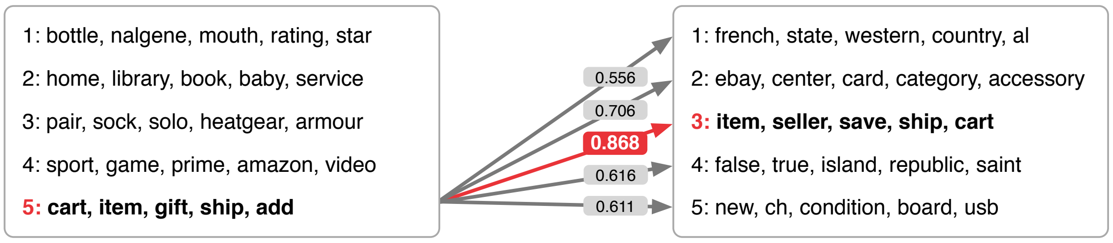

1: bottle, nalgene, mouth, rating, star
2: home, library, book, baby, service
3: pair, sock, solo, heatgear, armour
4: sport, game, prime, amazon, video
5: cart, item, gift, ship, add
0.556
0.706
0.868
0.616
0.611
1: french, state, western, country, al
2: ebay, center, card, category, accessory
3: item, seller, save, ship, cart
4: false, true, island, republic, saint
5: new, ch, condition, board, usb

Figure 5.6: Topic category matching with DISCO across the shopping carts of Amazon (left) and eBay (right). For each topic list, semantic similarity is computed to the topic lists in all cross application cluster

Style Sheets (CSS). Furthermore, it is a standard practice to define CSS classes to efficiently style HTML elements. For this reason, our distance measure for clustering states is based on the used CSS classes in a state. More specifically, we collect all CSS classes and styling properties within a state in a set and calculate the intersection to the comparative set. To compute the actual distance, we calculate the similarity of each block—called *block similarity* S_{Block}. Block similarity is typically part of noise detection algorithms and defined as

$$S_{Block} = \frac{2 \cdot M}{S_1 + S_2} \tag{5.1}$$

, where M denotes the number of matching classes (the intersection) and S_n the respective size of a set. We can configure a cut-off value designates the level of abstraction we want to use for our model. The smaller the intersection ratio of the CSS classes, the less likely it this that two states are presenting the same functionality.

Comparing the results of our state clusterer with a manually clustered reference partition shows promising results for different categories of web applications.

5.2.2 Cross App Analysis

Our key idea is that applications within the same domain share common process snippets to achieve a previously defined goal. SELENIUM test cases represent single workflow traces through the application and may fail if the absence of a certain ‘key’ element is detected, e.g. the element labeled with ‘Purchase Complete’ is present. At every point in time, a test can request a new SELENIUM driver, i.e. a new browser instance. Every browser instance is independent. That is, previously gained information (e.g. cookies, application state) are not transferred when a browser session is started. Every driver generates a new test *trace*. ATTABOY treats these traces mostly as independent, although test cases may influence each other. If for instance the application keeps some internal state which is not observable by the test case, ATTABOY will not detect an issue. In our later evaluation, this detail is irrelevant. All test cases are independent. In reality, one test case could create an account, while another one uses this information for further system interaction. Since test order dependency is not part of the conducted research, this characteristic is ignored.

Our information retrieval framework is configured to generate a new test trace whenever a new SELENIUM session is requested. A new state is extracted if a user action is executed, i.e. an action which might change the state of the application and is not a control command for the browser like *implicitWait* or *findElement*. More precisely, we do not consider time as a factor to change the state of an application, thus ignoring corner cases like timeouts (auto-logouts) or automatically sliding elements. Finally, the single test traces are combined into a single unified behavioral model as shown in Figure 5.4 by applying functional state clustering as described in Section 5.2.1.

When relating two behavioral models, two key aspects have an impact on the overall mapping: The *abstraction level* of the application under test, determining the cutoff value for the functional state clustering and the threshold ε_T to measure semantic equivalence of the topic model. The topic set of each cluster is compared against each cluster in the cross application using a performance optimized version of DISCO [39]. DISCO calculates the semantic text similarity of the topic sets. Figure 5.6 shows a sample matching between two shopping carts in Amazon and eBay. The highlighted topic set (cart, item gift, ship and add) represents the best match in this cluster pair. Three of the topics are identical, although the content of the shopping carts are divers, the matching process efficiently filters out the uncommon elements. The order of the words within the single set is ignored by the semantic text similarity of DISCO, following the intuition that the actual wording is of minor importance to understand the overall meaning of the presented text.

5.3 Evaluation

5.3.1 Evaluation Setup

To evaluate the performance of our semantic feature mapping, we selected 12 industry-sized real world applications out of 3 different domains (see Table 5.1). The candidates have been selected because they represent the most widely used applications according to the Alexa top 500 sites [2] on the web, for the selected domains. For each of these applications, we developed a custom SELENIUM test suite. These test suites are executed

Table 5.1: Evaluation subjects for matching features across applications. Features describes the number of features, which were identified when executing the test cases. Elements reflect the total number of UI elements per application with natural language content, which were analyzed for matching the given features.

	Test Subject	Features	NLC Elements	Interactive elements
Knowledge Base	`en.wikipedia.org`	29	6090	5620
	`en.wikiversity.org`	40	3703	3639
	`wikitravel.org/en`	58	1047	2461
	`en.citizendium.org`	32	775	638
eCommerce	`amazon.com`	55	5973	4564
	`ebay.com`	43	4848	3475
	`homedepot.com`	48	5389	4090
	`walmart.com`	66	3636	2249
Search-Engine & Mail Client	`google.com`	38	1521	1117
	`us.yahoo.com`	54	1335	1319
	`bing.com`	50	2140	2936
	`yandex.com`	38	795	797
	TOTAL	551	37252	32905

as depicted in Figure 3.2 to obtain the web application models and their designated features.

As already described, the matching method does not require two application models as an input, but just analyzes one (source) application model *A* to identify features and takes the DOM states of the target model *B* to identify the same features. The test setup thus runs for instance the test suite for AMAZON, extracts the tested features together with their describing labels and identifies the features in EBAY for which we provide the DOM data. To verify the correctness of the matching, we furthermore manually labeled each of the 551 features and test if the designated XPATHs match the predicted ones.

Each application is tested against all other applications within the same domain to

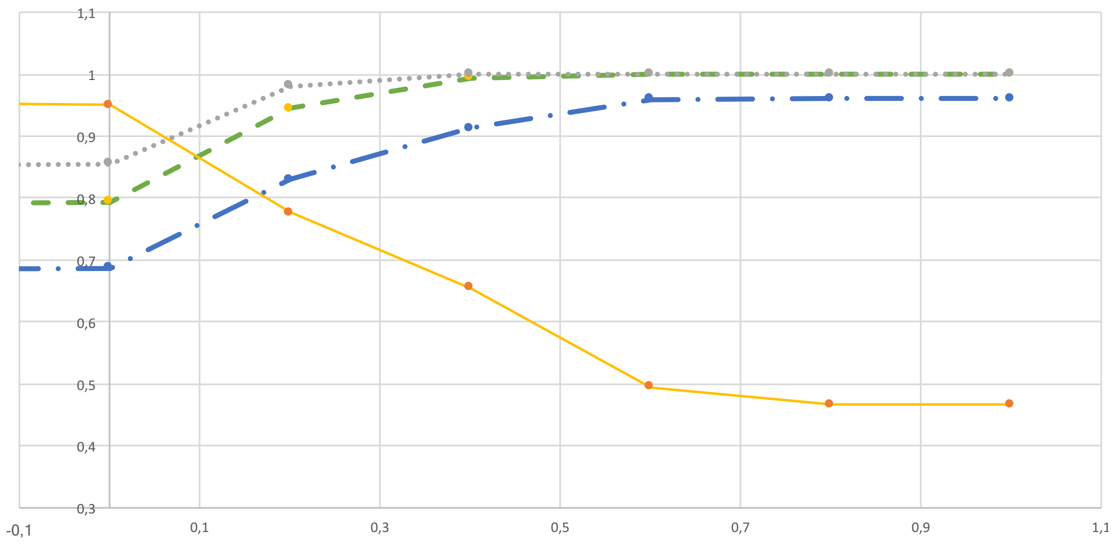


Figure 5.7: Averaged overall results for feature matching. The x-axis shows the threshold applied on $\equiv_{sem}$, the y-axis displays the values for $P@k$ and *Recall* (solid yellow line). The blue (dash-dot) line is $P@1$, the green (dashed) line is $P@3$, and the grey (dotted) line is $P@10$.

identify the features. Analyzing the produced data, we address the following two research questions:

RQ 4 Can we identify features using semantic UI element similarity?

This is the core contribution of ATTABOY: Mapping features from the source application, as exercised in the source tests, towards the features of the target application. We evaluate the accuracy of the mapping; the better the accuracy, the more effective guidance for subsequent exploration and test generation.

RQ 5 Can automated feature identification be used to improve crawling?

This is the main application of ATTABOY: What difference does it make to a crawler if guidance exists? We conservatively evaluate the worst case speedup of guided crawling vs. non-guided crawling.

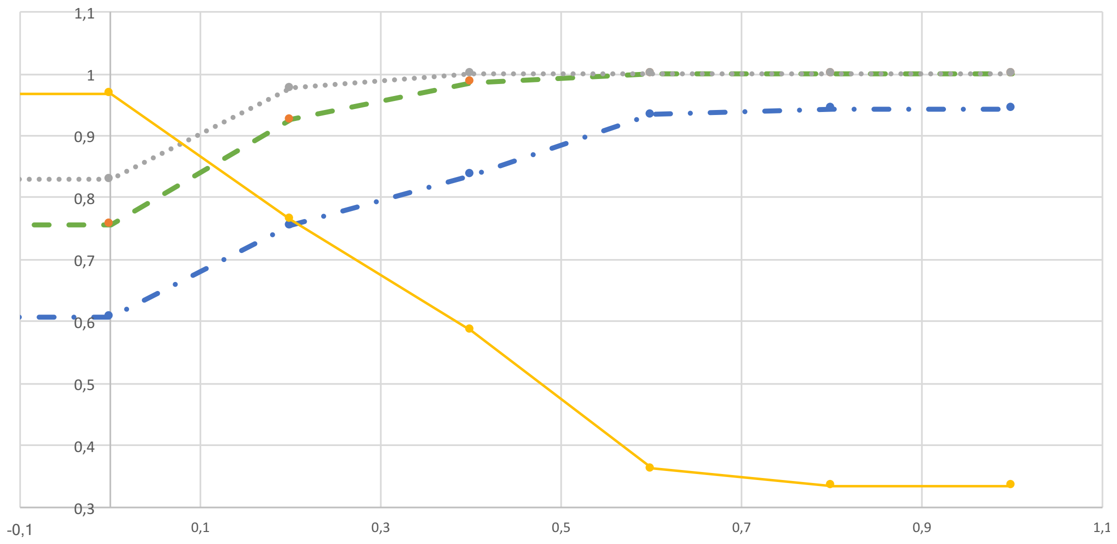


Figure 5.8: Averaged feature matching results for the domain *eCommerce*. The plot colors are consistent with Figure 5.7.

5.3.2 Identifying Features Across Web Applications

We start with RQ 4, evaluating the accuracy of the mapping of features from source application to the target application. Figure 5.7 to Figure 5.10 present the results of our evaluation averaged by the domain. The blue (dash-dot) line shows the overall precision ($P@1$) and recall values. In the area of information retrieval the *precision at k* [49] denotes how many good results are among the top k predicted ones. $P@1$ for instance indicates the precision, of all tested applications within the domain for a perfect match (the top element is the correct feature), while $P@10$ indicates that the correct result was in the top 10. The y-axis represents the threshold for the predictor to classify two features as matches. The semantic similarity calculation always returns for any given feature a list of elements, which might be possible matches together with the semantic similarity index $\equiv_{sem}$.

The threshold value 0 marks the point where only semantically similar concepts are taken into consideration. Nevertheless, it shows that even with a non-existent filtering (i.e. threshold 0), the semantic matching allows to correctly identify 69% of the features. Increasing the threshold, i.e. raising the minimum semantic similarity to identify features as matching, not surprisingly increases the precision up to a value of 1, which means that the given features have identical describing labels. On the other hand, the

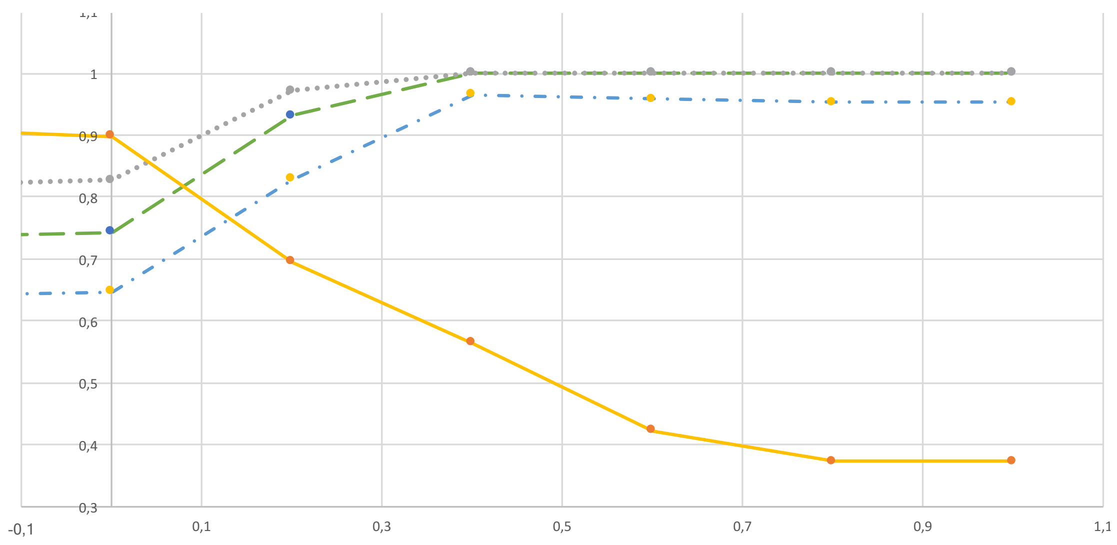


Figure 5.9: Averaged feature matching results for the domain *Search Engines & Mail Client*. The plot colors are consistent with Figure 5.7.

recall quickly decreases disproportional compared to the increase in recall, meaning that a low threshold of 0.15 is producing the best ratio.

Overall, we correctly identify 75% of the 551 features in the test data with a precision of 83%. This overall value does not reflect the average precision and recall values per domain, since the number of tested features differs within the domains. When checking the matching success of the individual domains the result of the domain *Knowledge Base* (Figure 5.10) is noteworthy compared to the other results. The features here can be matched with an average precision and recall of more than 90%. This means that they not only share pretty much all features, but that they are represented in the same semantic context. Looking at the test subjects, this is no surprise. The first three sample pages use the same underlying content management system (i.e. the mediawiki framework[3]), the tested features are often encapsulated within the same describing labels. Thus, the precision values are even higher than in other domains. But even more complicated features such as filling a shopping cart or filling payment information are identifiable by the semantic feature matching. Figure 5.8 (eCommerce) shows that direct matching has a precision of 77% at the same recall rate. The category *Search Engines & Mail Client* (Figure 5.9) offers less functions, but the overall matching success shows average results.

[3]`https://www.mediawiki.org`

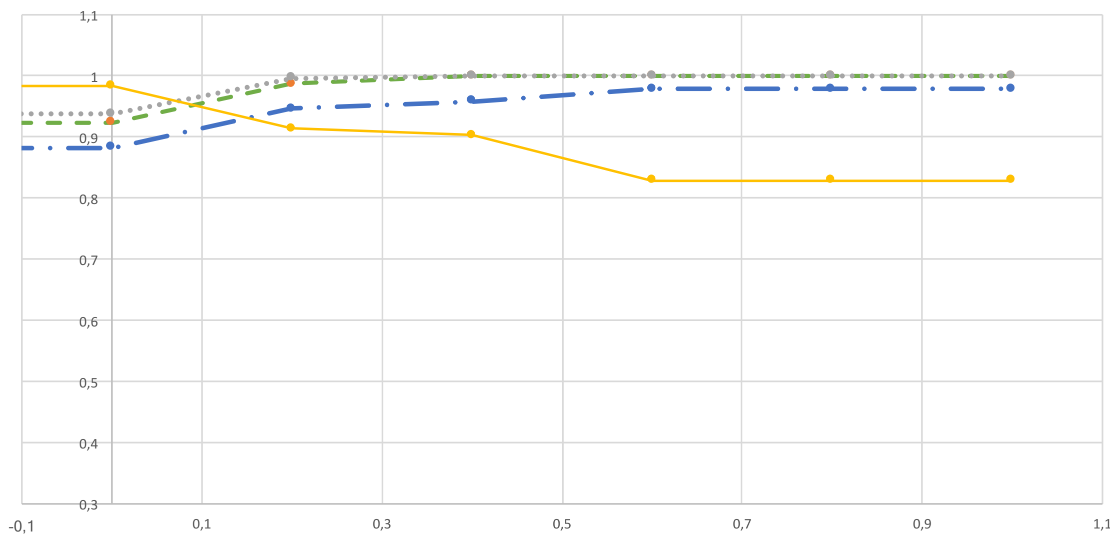


Figure 5.10: Averaged feature matching results for the domain *Knowledge Base*. The plot colors are consistent with Figure 5.7.

Summarized, this answers our research question RQ 4 about Accuracy: *Can we identify features using semantic UI element similarity?*

☞ *Overall, semantic feature identification is able to directly match* ***75% of the features with a precision of 83%*** *(P@1).*

☞ *The results provide credible evidence that we can effectively and automatically match and identify features. The test domain does have an impact on the accuracy, though.*

5.3.3 Crawling Speedup

We continue with RQ 5, assessing how much automated feature identification improves crawling and subsequent test generation. Random crawling techniques often face two problems: (i) the number of possible actions in a given state is large, and (ii) generating valid actions to interact with the elements. With the presented technique, we address both accounts. The DOM states of our web application models in our test setup feature a total of 155,858 DOM elements, when we execute the SELENIUM test cases and apply functional state clustering. Most of them may not be interactive, but the tested real world application use thousands of event handlers on their pages. The majority is used to track user behavior and thus do not express features. To evaluate the impact of *semantic* *feature identification* on crawling we imagine the most simple scenario for generating tests. Table 5.1 shows the number of naively interactive UI elements for each page — just counting standard HTML elements like anchors, INPUT, SELECT nodes. In the recorded test states we thus have more than thirty thousand elements. Even without considering interdependencies between features or how to reach a feature (leading to state explosion), the chance to randomly interact with the elements representing a feature is negligible.

Guiding the crawler to the correct feature does not require a perfect matching, but instead profits from a *ranked list of recommendations* to derive a crawling strategy. For this reason, we evaluated (Figure 5.11) the probability that the correct feature is within the top 3 ($P@3$) or top 10 ($P@10$) results. The *Precision at 10* for eCommerce shows for instance that at a recall rate of 88% semantic matching can identify features with a precision of 92%. In the domain of *search engines and mail clients* the matching is capable of identifying 77% of all features with a precision of 97%. Choosing different thresholds for the prediction impacts these values tremendously. Nevertheless, the values do not tell the direct impact on the crawler effectiveness.

To this end, we evaluate *how effective guided crawling is compared to non-guided crawling.* Let us start with a simple crawler that has no guidance (i.e., *not* using our technique). Assuming that the application has a number $|UI|$ of user interface elements the crawler can interact with, a non-guided crawler would have to test each UI element individually to find a feature:

$$\textit{number of tests without guidance} = |UI| \tag{5.2}$$

Let us now look at a guided crawler, where our guidance provides a recall of r across the UI elements in the target that are matched in the source. Within the set of $r \times |UI|$ matched features, we have a set number k of top matches, and the probability $p_{@k}$ that the correct feature is within these top matches. The number of correct matches to explore can thus be estimated as $p_{@k} \times k$; whereas the number of incorrect matches would be the converse $(1 - p_{@k}) \times (|UI| - \text{k})$. (With perfect guidance, $p_{@k} = 1.0$ would hold, and the crawler would only have to examine k matches).

If we do not find the target within the matched features, we have to search within the *unmatched* features, going through an average number of $(1 - r) \times UI$ elements. The total number of tests thus is:

$$\textit{number of tests with guidance} =$$
$$\underbrace{r \times \left(\overbrace{(p_{@k} \times k)}^{\text{correct matches}} + \overbrace{(1 - p_{@k}) \times (|UI| - k)}^{\text{incorrect matches}} \right)}_{\text{matched features}} + \underbrace{((1-r) \times |UI|)}_{\text{unmatched features}} \quad (5.3)$$

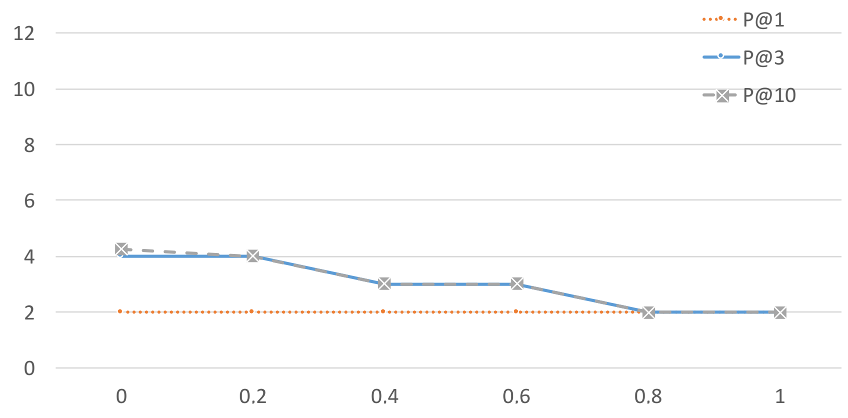
Note how the number of tests with and without guidance are equivalent in the cases of $r = 0.0$ (we have no matched features), $k = 0$ (we make no recommendation), and $p_{@k} = 0.0$ (all our top predictions are incorrect). In the extreme case of $r = 1.0$ (all features are matched), $k = 1$ (we only examine the top recommendation), and $p_{@k} = 1.0$ (our recommendation is always correct), then we have to test only one single (recommended) UI element.

With the number of tests for both unguided and guided exploration, we can now compute the average speedup as

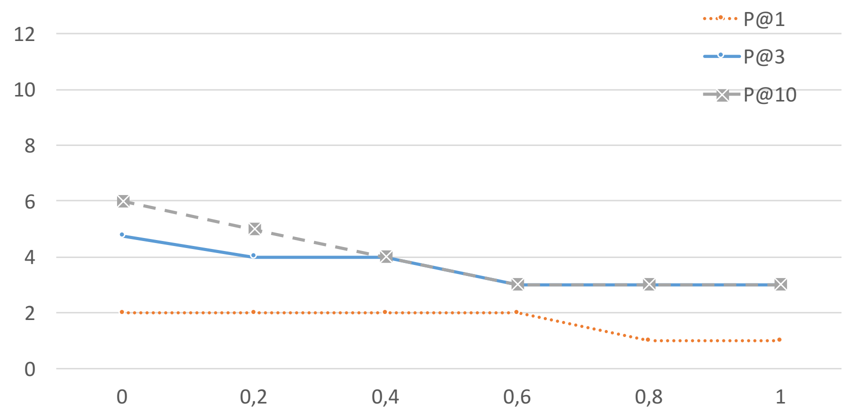
$$\textit{speedup} = \frac{\textit{number of tests without guidance}}{\textit{number of tests with guidance}} \quad (5.4)$$

Figure 5.11 shows the speedup results, as determined from our earlier evaluation result data. *The average speedup over all domains is 740%.* Testing knowledge bases, which already showed good performance in the earlier evaluation step, can even be improved by a factor of 11.75. Figure 5.11 also shows that a high threshold slows down the exploration process—that is, if backtracking and re-executing an action is not penalized in terms of execution time. A high recall is thus more important than a high precision. $P@3$ and $P@10$ outperform the direct match by orders of magnitude and are preferred when it comes to guide test generation.

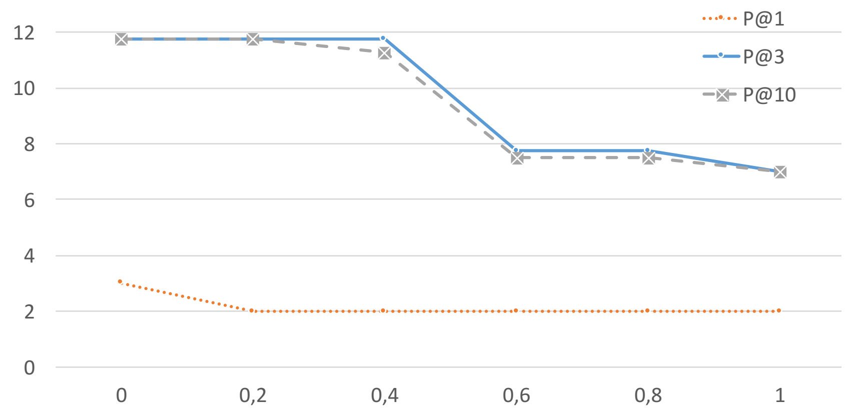
$P@3$ and $P@10$ denote the precision of the algorithm if we allow the correct result to be among the top three or top ten of all predicted values. It turns out, that this relaxation indeed improves the results significantly, which does not come to a surprise. If we lower the correctness threshold to be predicted among the top ten results, the prediction has more leeway to be correct. Figure 5.7-Figure 5.10 also show the results for $P@3$ and $P@10@$. The results for the domain *Knowledge Base* are virtually unaffected for the aforementioned reasons, but especially in the domains of *eCommerce* (increase from 61.3% to 83%) and *Search Engines/Mail Clients* (increase from 65% to 85%) the precision is drastically improved.



(a) Search engine applications



(b) eCommerce applications



(c) Knowledge base applications

Figure 5.11: Worst case averaged speed-ups for guided crawling compared to random crawling in the different domains. As in Figure 5.7, the orange (dotted) line show $P@1$, the blue (solid) line $P@3$, and the grey (dash-dot) line $P@10$; the x-axis shows the threshold applied on $\equiv_{sem}$.

In other words, by testing the top 10 results predicted by the algorithm, we have a 90% chance to test almost all features from the training data. Transferring the knowledge from the original test suite on how to interact with the UI elements (e.g. if the element should be clicked, or which values should be send) might be the next step.

These speedups are computed on a per state basis, meaning we assume the crawler already was able to somehow reach the state with the features under test. In other words, the speedups for the crawler multiply each other, the deeper we proceed within an execution trace.

This answers our research question RQ 5 about Impact on web crawling: *Can automated feature identification be used to improve web crawling?*

☞ *Semantic feature identification guides and speeds up exploration and test generation.*

☞ *For a single state,* ***the average speed up over random exploration is 740%.***

☞ *If backtracking and re-executing an action is cheap, @10 feature identification with a low threshold is the most efficient crawling strategy.*

While a conservative speedup of seven is already impressive, one should note that this speedup applies for the exploration of *one single state* only—that is, providing guidance towards all the UI elements that are directly reachable from this state. If the functionality we search is deep within the target application, *semantic guidance speeds up every exploration step along this path.* Assume that we need to cover n states before reaching our target. In the extreme case, each of these n states again provides $|UI|$ elements to explore, only one of which will lead to the target. The average speedup of guidance over n states thus would be $speedup^n$. Note that n can easily reach values of five and more; to get to the eBay payment page (Figure 5.2), for instance, one has to go through six interactions. Assuming the average speedup $speedup = 7.40$ from above, we obtain an overall speedup of up to $7.40^6 = 164,206$.

Note, though, that this is the extreme case, in which each state leads to new states never seen before—but on a site like eBay, crawling will lead to the same states again and again. While recognizing equivalent states in web pages is a problem in itself, let us assume a random crawler that is able to recognize states with a precision of 100%, and a web application in which each and every UI element yields a redundant state. Even in this worst-case comparison, the speedup would still add up for each state over the path, yielding an overall speedup for a depth of six of $7.40 \times 6 = 44$. While the actual speedup for any web application will be between the exponential and the product, depending on the application path redundancy, it is clear that the deeper the target, the higher the speedup.

It stands to reason that the target selection strategy is different if testing faces state interdependencies or is executing complex use cases that require a strict execution order. The rather lax selection ($P@10$) has to be changed to a strict selection, because an incorrect selection (and the inevitable rollback) is too expensive.

☞ *The deeper the target, the higher the guidance speedup cumulated across multiple states.*

☞ *For an application with little state redundancy, the speedup is **exponential** over the depth of the searched functionality.*

Given the complexity of business processes and modern web applications, we actually do not interpret these cumulated speedups as an improvement over existing random crawlers; instead, we see our guidance as actually *enabling* their exploration and testing—in particular as it comes to cover deep functionality. This opens the door for writing a test case once for the source application, and reusing it again and again to test the same functionality in target apps, even if it is hidden deep in the system.

5.3.4 Transferring Test Suites

In order to test the generality of our approach, we selected a set of representative web applications from three different website categories obtained from the Open Directory Project[4]. We manually created a SELENIUM test suite covering the most typical user behavior, which is, for example in the eCommerce domain, searching for products, adding them to a shopping cart and finally making a purchase. Table 5.2 shows the selected test candidates. The *Size of Test Suite* represents the number of user actions applied in the SELENIUM test suite, i.e. internal actions like *implicitWait*, *findElement* or likewise are not part of this metric as discussed before. They are not supposed to cause a visible change to the content of the website. To have equal conditions, the test cases on the other application within the domain test the same functional behavior and contain a comparable distribution of similar states. *#Functional States* hereby expresses the size of the extracted reference graph after processing the model as presented in Section 5.2.

In a second phase (see Section 5.3.5) we extended the set of test subjects by yet another domain ('news papers'). Instead of manually creating test cases, we applied random crawling to automatically generate test traces through the applications of all

[4] `dmoz.org`

Table 5.2: Overview on training candidates for manual test case translation. *Size of Test Suite* is the number of executed user actions; *#Functional States* is the number of covered functional states in the behavioral model.

	Start Page	**Size of Test Suite**	**#Functional States**
Knowledge Base	`en.wikipedia.org`	35	3
	`en.wikiversity.org`	15	9
	`en.citizendium.org`	15	7
	`wikitravel.org/en`	17	4
eCommerce	`amazon.com`	32	7
	`ebay.com`	34	6
	`homedepot.com`	17	4
	`walmart.com`	18	6
Search Engines	`google.com`	16	8
	`uk.yahoo.com`	30	7
	`bing.com`	32	7
	`duckduckgo.com`	15	4

four domains. Then we compare the test traces in terms of model similarity and the possibility to share basic test suites.

Transferring a test suite across applications is the key idea of the ATTABOY-prototype. It executes our manually created test suites against our instrumented SELENIUM server and matches the intermediate result states (after each single user action has been executed). Figure 5.12 presents the results for transferring the test suites across different domains.

On average, we were able to transfer 65% of the eCommerce test suites, 50% of the search engine test suites and 62% of the knowledge base test suites denoted by the fact that the correct functional states have been matched across the applications. Even more interesting is the distribution of the result. When translating search engines, the fluctuation of the results is almost nonexistent, meaning that although we could only translate half of the tests, all applications feature almost identical functional states

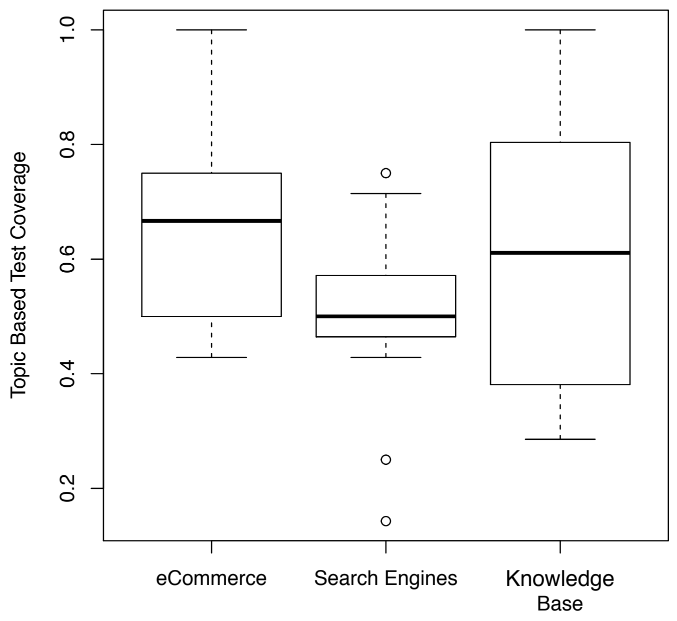


Figure 5.12: ATTABOY's results for transferring the manual test suites across eCommerce, search engine and knowledge base applications.

within the covered parts. eCommerce systems on the other hand feature a wider variety of functional states. The Walmart web shop encapsulates the search & order process in only four functional states. Amazon in comparison uses seven functional states and displays certain pieces (e.g. billing address and payment information) in different states.

Accordingly, a greedy mapping algorithm pursuing the best possible match cannot efficiently map these two test setups. Finally, knowledge data bases show an even wider range in the matching result. Although 62% of the tests were successfully transferred, the translation process of ATTABOY is not equally successful to match the states correctly through all applications. The fact that the applications are used to present different articles, causes the topic analysis to overfit on the underlying content. Increasing the ratio of article pages, might improve the result, since the functional clustering would group multiple articles. The subsequent topic extraction would collect only the common topics, thus avoiding the overfitting.

The same variance can be observed in the knowledge bases, although the overall result of 62% is pretty good, the effect of the clustering techniques comes a bit to a surprise. All four test candidates are written in the MEDIAWIKI content management system, thus sharing common styling attributes. Nevertheless, the clustering technique collapses the intermediate states on Wikipedia and Wikitravel far stronger than their competitors, e.g. due to the content distribution within articles like images, table of contents entries or likewise. The matched states are featuring the login functionality and editorial pages, but the article pages cannot be matched. Within this small subset of tests are structural analysis might yield better results or the data has to be enriched with more states covering more news articles.

This analysis allows us to answer RQ 6 on Test Transfer: *Can a given* SELENIUM *test suite be transferred to the test suite of another application within the same domain?*

☞ ATTABOY *is able to transfer an average **59%** of the **manual Selenium test suites** across all domains.*

☞ *Results show initial evidence for the feasibility of the approach. A complete test suite transfer requires further research.*

☞ *An oracle transfer is only possible by checking the state similarity. Additional properties (e.g. contents, computational values, etc.) have to manually transferred.*

5.3.5 Usage for Test Generation

So far, ATTABOY was able to transfer test suites actually designed to verify the correct behavior of the application under test across other applications with appropriate performance. Writing SELENIUM test cases requires serious effort and is error prone if the application evolves by adding new features or the structure of the pages is changed. On the other hand, the usability standards dictate the expected outcome of each action. Those standards are typically not changing drastically. If they are consistent throughout the application, we might be able to infer them from other a applications within the domain.

As a consequence, RQ 7 investigates if an automatically generated test suite can also be transferred. In the second step of this evaluation, we analyze how efficient ATTABOY works when integrated with an automatic black box test generator for web applications. Random crawling solutions like CRAWLJAX [53] or CRAWLER4J [24] can be leveraged to test the behavior of web applications, although they do not provide an oracle as to whether the reached state is the one we expect after applying the crawling action. We provide this oracle by mapping the functional states across application, thus learning the expected behavior from an existing reference.

Integrating the random crawler into ATTABOY is straightforward. The crawling techniques can be run against a central SELENIUM grid, where we intercept the applied

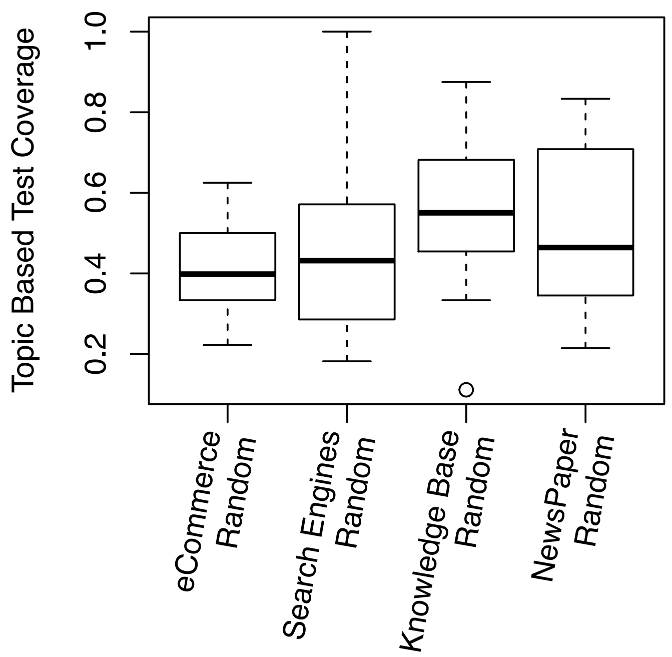


Figure 5.13: ATTABOY's results for transferring automatically generated test suites across domains.

actions and provide them as discussed earlier in form of test traces into our information retrieval method. ATTABOY takes the resulting models and compares them against the models generated in the same domain. Table 5.3 shows an overview on the generated application models created by the crawler.

A fourth category ('News web site') has been added to the test set in order to analyze the performance of ATTABOY without training it on existing tests first. Notable is the ratio in between the number of applied actions against the number of discovered functional states. Especially in the domain of search engines and knowledge databases the crawler did apply a significant number of actions, most of them resulting within the same functional state. As a side effect of randomly crawling the applications, the distribution of functional state is different compared to manually written tests. In other words, the knowledge bases are providing thousands of articles, but for instance only one distinct login or sign-up page. Here, the overfitting against the actual article content is both reduced by the noise reduction and the clustering technique. Hence, the functional state distribution is more uniform. Except Amazon and BBC, all applications show more or less the same number of functional states, while we used the same level of abstraction in the pre-processing phase.

Figure 5.13 shows the success on transferring the randomly generated test traces.

Table 5.3: Evaluation setup of automatically generated tests across four domains. Again *Size of Test Suite* is the number of executed user actions; *#Functional States* is the number of covered functional states in the behavioral model. Starred test suites have been cut short after one hour of exploration.

	Start Page	**Size of Test Suite**	**#Functional States**
Knowledge Base	`en.wikipedia.org`	35	11
	`en.wikiversity.org`	100*	11
	`en.citizendium.org`	100*	8
	`wikitravel.org/en`	63	9
eCommerce	`amazon.com`	100*	27
	`ebay.com`	138*	16
	`homedepot.com`	86	18
	`walmart.com`	100*	18
News web site	`edition.cnn.com`	21	6
	`europe.newsweek.com`	35	8
	`bbc.com`	49	14
	`nytimes.com`	100*	6
Search Engines	`google.com`	100*	7
	`uk.yahoo.com`	44	11
	`bing.com`	100*	7

On average, ATTABOY is able to translate 49.3% of all automatically generated test cases. Especially compared against the previously presented results of the manual tests on knowledge databases, which showed a high fluctuation caused by the different topics within the news articles, the random trace through the application revealed far more articles and the functional state clustering allowed us the reduce the topic overfitting on single articles. In the previous analysis, knowledge bases had a fluctuation of almost 43%, the more general tests in the random testing result has only a fluctuation of 23%, while the average result is more or less stable. In contrast, the variance in the eCommerce sector has stayed the same, although the average mapping result has dropped. Not surprisingly, the random crawling technique did not manage to login or purchase a product, both established patterns in the eCommerce sector and clearly distinguishable from other functional states. Instead, other random pages like imprints or even job opportunities have been discovered. Since the models are far from being complete, e.g. no authorized state has been reached.

RQ 7 (Test Generation): *Can we transfer automatically generated test suite across applications?*

☞ ATTABOY *is able to transfer an average of* ***49.3%*** *of the* ***automatically generated*** **SELENIUM** ***test suite*** *across all domains.*

☞ *Testing can be guided towards relevant functionality without additional manual effort. The test coverage of 49.3% indicates further potential, but also requires further research.*

5.3.6 Application Similarity

Transferring the randomly generated SELENIUM test suites across domains provides us also with a notion on how similar two applications are and how established their usability and feedback patterns are. A closer look at the random matching data in Figure 5.13 reveals that the matched functional states are common throughout the applications. The dimensions of the boxes in the plot indicate a range of approximately 20%, meaning certain functional states are found throughout all applications.

The fact that the actual content of the application can be abstracted to extract the underlying concepts shows the global acceptance of usability standards throughout the applications. For a human being, this comes as no surprise and follows the intuition of global concepts for logging in, shopping carts or sign up functions.

The test analysis in Section 5.3.5 also showed the possibility to transfer complete and complex process snippets across application boundaries. Not only is the functionality of single pages a transferable concept, but a sequence of actions leading to the same goal is transparent to further analysis.

RQ 8 Generality: *How similar are web applications regarding their features and feedback mechanisms?*

☞ *the domain of an applications is strongly correlated to the functionality it offers*

☞ *the state changes and the observable feedback is not conclusive enough to serve as an oracle, but further sentiment analysis might improve these results*

☞ *domain specific testing allows to automatize testing to a certain degree*

5.4 Threats to Validity

As any empirical study, this one faces threats to its validity. A severe challenge is the underlying model abstraction. Based on the chosen level/cutoff value, a functional state can contain the whole application (i.e. cutoff value equals 0) or almost every single action leads to a new functional state if only on element is different. The chosen representatives allowed us to manually configure the cutoff value throughout whole domains of applications, but web applications tend to be developed very fast and this behavior might change over time.

The same can be said about the noise reduction. Due to the dynamic calculation of the noise threshold, the algorithm is rather robust against changes of the application under test. Nevertheless, some information like common menu structures contains valuable classification information, e.g.filtering options on result lists, which ATTABOY considers to be noisy. Changing the test subjects or performing the same analysis on a newly generated test setup might change the results.

ATTABOY shows the ability to transfer complete test suites, even analyzing the commonalities in the resulting state models. Still, not all possible decisions within SELENIUM tests can be translated — they are Turing complete. Imagine a test case which dynamically collects all items in a shopping cart and checks whether the overall price matches the actual sum of all items. If not, it fails. ATTABOY is not able to draw the same conclusion, i.e. it would detect that the previous action indeed ended up within the shopping cart, but not that a displayed element shows the incorrect result.

5.5 Measured and Perceived Success

At this point, I would like to assess and summarize the quality of topic-driven testing for web testing. Our case studies have shown indications that we can indeed learn relevant functionality from a sample set of test applications. In the presented setup, the input for TDT was provided in terms of SELENIUM test scripts. TDT was able to learn and extract semantic patterns out of the GUI and use them for test generation in new applications.

While feature learning is successful per se, an automated test transfer is—if we are realistic—not complete or possible, yet. First of all, the training set for our experiments is too small to draw realistic conclusions for other tests. We tested common features that we could manually identify and test. The test suites cover default behavior, but anything beyond that has not been tested so far. Second, the oracle transfer is not suitable for testing. The oracle transfer currently only analyzes state similarity and control flow similarity. Anything beyond that, e.g. calculations of intermediate values, test interdependencies, or checking properties that are not visible in the GUI, is something that topic-driven testing alone cannot solve. An integration with other complementary testing techniques, such as model-based testing, or an instrumentation of the server side code might yield better results. Extending the learning base might also allow us to gain additional knowledge. We can grow our learning base by executing the tests on a broader set of applications. With this additional data, we should be able to gain more knowledge on the expected system behavior.

The main challenge in conducting the presented empirical studies was the identification of *common* functionality. The test transfer can only be as good as the initial test suite for the source applications. Missing test cases result in missing tests in the target application. Tests that cover functionality that is not present in the target applications (i.e. additional tests) are even more problematic. Intuitively, such a test should just fail. But in our experiments, we achieved only a partial test transfer in most cases. Differentiating an incomplete test transfer from a failing test or missing functionality requires manual effort. In other words, filtering existing tests for their relevance for a target AUT might require a higher effort than just writing a new test.

Regardless of the measured success for test transfer, topic-driven testing actually enables further analysis possibilities in the first place. It allows to put observations of multiple test applications into context, and find outliers. Additionally, it stands to reason that topic-driven testing can be used to strengthen the robustness of existing test suites. Since we are able to identify UI elements across applications, we can also use this capability to identify UI elements if the application evolves (regression testing) or a transfer to a different platform (cross-platform testing). In the following chapter (Chapter 6), we conduct further studies to show the generality of topic-driven testing on the Android application (cross-platform compatibility) and also the independence from the underlying test input. In this section, the *feature learning* was based on the observations from existing test cases. Meanwhile, the Android testing part takes vague natural language instructions and still is able to improve testing compared to random testing.

6 Testing with Process Snippets

Parts of this chapter have been published by Rau et al. [68]. The author of this dissertation contributed the semantic matching and natural language concepts that guided the exploration. It shows the versatility of the technique in an extended application context.

So far, we have shown that topic-driven testing can be useful to (i) re-identify functionality in web applications, (ii) learn process specifications from UI tests scripts, and (iii) identify semantic entities in a large application set. For a complete test transfer, topic-driven testing is not evolved enough and the training set is too small. Instead, it can effectively guide testing towards relevant functionality. In this chapter, the analysis will focus on Android testing, which also shows the platform independence of the new testing principle. We want to have evidence that topic-driven testing is generalizable to GUI applications in general.

Android applications have some notable and profound differences in contrast to web applications. Due to hardware specific limitations (such as screen size, or performance limitations), the UI shows less elements and potentially less noise. Furthermore, the operating system allows a finer control of security properties, since the relevant access to sensitive information sources is only available via OS-specific API. The OS thus controls access to sensitive information or also can limit the information flow to external parties.

Functional testing of real-world apps typically requires *exact specifications* of each and every action to be conducted: “Click on Button A”, “Enter City Name into Field B”. Such specifications have to be written and maintained for each and every app, again and again. In this chapter, we will investigate to what extent we can use topic-driven testing to automatize this process. Instead of repeatedly prepare specifications, we use topic-driven testing to translate a general set of use case descriptions into machine-executable statements. These statements are intended to be executed on a large set of applications at once. Our hypothesis is that this improves system penetration, and guides testing towards relevant system functions. In a nutshell, we present the novel concept of *process snippets* to considerably simplify app testing. Process snippets allow for *abstract*, *natural language* descriptions of the actions to be performed: A step such as "Add to Cart" selects the available GUI action that is *semantically closest*; this allows for the step to be applied and re-applied for several apps and contexts. Process snippets can be *composed* from each other: A snippet "Buy a Product" thus can be composed of (i) selecting a product, (ii) adding it to a cart, (iii) proceeding to checkout, and (iv) entering delivery and payment information. Once defined, such a snippet would be applicable on any standard shopping app.

Process snippets serve as the input to topic-driven testing procedure. They are translated and matched to the GUI of each application under test (AUT). We perform a set of empirical studies on fifty popular real world Android applications. Our analysis shows that our approach covers on average twice the amount core functions within twenty percent of the testing time compared to random testing.

6.1 Introduction

Automated application testing has been a common goal of many research projects on all levels of software products and processes, be it on unit level, integration level, or system testing. It does not replace the need for manual testing and inspection of the produced result but can help with tedious and simple tasks such as generating inputs and checking if the *AUT* breaks. One of the central techniques for automated application testing is *crawling,* i.e., exploring an interactive application through its user interface to discover functionality. Naïve crawlers, such as MonkeyRunner [5] for Android apps, randomly emit UI actions. They require little effort for implementation and deployment. Given sufficient time (i.e. often an infinite), a crawler can exhaust the search space of the application by interacting with all UI elements and randomly generating all possible inputs. Smarter state-of-the-art test generators, such as DM-2 [14] speed-up the exploration by analyzing the application and only trigger actions on interactive elements.

Since random crawling techniques treat every action with the same priority, they do not create valid inputs most of the time, and do not guide the exploration process towards desired *core functionality*. Moreover, complex use cases like a shopping process (such as motivating example in Figure 1.1) are unlikely to be completed by chance, as the number of explorable UI targets grows exponentially with the length of path. To guide exploration and to provide valid inputs, crawlers are given application specific or domain specific knowledge about the AUT. Such specifications, however, have to be recreated for every AUT, which requires significant (often manual) effort.

To address this problem, we build on the observation that process steps of interactive

```
# BUY
Do SEARCH // Process with 2
    ATDs
Add to Cart
Proceed to Checkout
Do LOGIN // Process with 3
    ATDs
Do ADD_ADDRESS // Process
    with 8 ATDs
Continue
Do ADD_CREDIT_CARD // Process
    with 5 ATDs
Continue
```

```
# Add_Address
Enter Name John Tester
Enter Country "United States"
Enter Street "111 Sproul Hall"
Enter City "Riverside"
...
```

```
#LOGIN
Enter username
    "testX0123@test.com"
Enter password "sample42P!"
```

Figure 6.1: Sample Snippet for Buying a product. It includes four more basic snippets: *Search*, *Login*, *Add Address*, and *Add Credit Card* and thus has a total length of 22.

applications may be *syntactically different, but semantically similar.* A button to log in may be named “Sign In”, “Log In”, or “Authenticate”; at a semantic level, they all mean the same. Our key idea is to introduce so-called *process snippets* that abstract over such syntactic differences to introduce *semantic entities* representing one step in an abstract interaction process.

Figure 6.1 shows how an abstract shopping process is expressed as a process snippet, which then can be applied on all applications in the test set. The use case description is short (i.e. only eight instructions) and references shorter use cases such as *searching for a product*, *authentication*, or *providing a valid address*. The key point is that descriptions such as “Add to Card” or “Proceed to Checkout” are *semantic* descriptions which may or may not have exact syntactic matches in the application. We interpret the use case description *semantically* by calculating the semantic text similarity [31] between the target description and the labels present in UI elements of the AUT, which allows our prototype to discover the most likely target. For example, in the “LOGIN” process snippet (referenced from the “BUY” process snippet), the field “username” matches UI elements such as “User ID”, “Login”, or “e-mail”.

Formally defined, our *process snippets* are thus a grouped list of ATDs and intuitively form a use case, such as authentication or buying a product. The instructions can be parsed to a machine executable code to guide the crawler.

Once defined, process snippets can be reused again and again for each application that follows a similar process. We have defined a set of use cases as a *repository of process snippets* and tested them on all AUTs in our test set. Even complex use cases, such as the shopping process in Figure 1.1 can be built by integrating the more basic steps. The approach is generally applicable to arbitrary GUI applications, however in the scope of this paper, we target Android apps to evaluate our technique.

To show the effectiveness of our technique, we perform an empirical evaluation on fifty of the most popular applications in the Google Play Store out of different application domains, such as shopping, travel, or food. We show that testing with process snippets is versatile enough to (1) generalize over these applications, (2) guide crawling towards core functionality, and (3) outperform random testing in terms of system penetration and functional coverage.

A part of this chapter was published in Rau et al. [68] and made the following contributions:

☞ *A new, easy, intuitive way of developing and applying UI tests. Testing with process snippets (Section 6.2) is intended to serve both industrial testing as well as academic research purposes. It allows to generate reproducible and comparable tests on a multitude of applications.*

☞ *A repository of extensible widely usable process snippets covering common use cases for further app testing.*

☞ *A runnable implementation of the presented technique for testing Android apps in a public repository, together with all required artifacts and infrastructure. It includes a repository of extensible widely usable process snippets covering common use cases for further app testing [65].*

☞ *An empirical evaluation (Section 6.3) that provides initial evidence of the effectiveness and versatility of the approach.*

The conducted empirical studies show that the topic-driven testing principle can be implemented on real problems and that it can improve state-of-the-art testing methods.

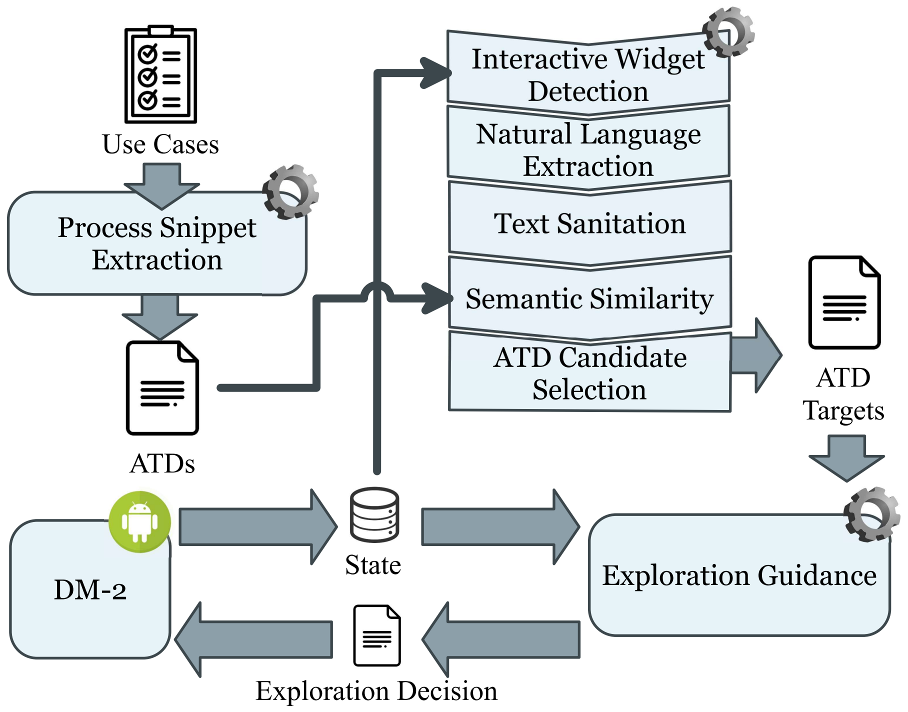


Figure 6.2: Overview of the processing steps for process snipped guided exploration.

6.2 Testing with Process Snippets

This section describes our technique for translating natural language use case descriptions (i.e. process snippets) into executable crawl rules. Figure 6.2 shows a high-level overview of the technique, which we discuss in further detail in this chapter. In the first step (Section 6.2.1), we mine process snippets from a set of input files into a list of Action-Target-Data tuples. Our prototypes resolve simple one-line instructions and complex interdependent process snippets into an instruction list, which is composed of the type of action, the target descriptor and optional input data. In the second step Section 3.1.2, the set of given test applications (i.e. a set of installable APKs) is deployed to a target Android device and started. Now, each presented UI screen is analyzed to extract interactive widgets together with their descriptive labels. These labels are analyzed in order to decide which exploration action should be prioritized. Finally, Section 6.2.3 describes the third step, which is to guide the crawler to maximize the core functionality coverage. Essentially, our prototype tries to find suitable targets for the provided ATDs. If there is no such target, i.e. the probability for a match is too low, the exploration strategy falls back to randomly explore the application until another suitable target is found.

6.2.1 Process Snippet Extraction

In this first step, we extract a machine-comprehensible list of instructions out of a set of use case descriptions written in natural language. It allows us to describe complex use cases such as “buying a product” or “book this flight” in a short recipe, a so-called *process snippet*. Process snippets describe short use cases of an application. Their main advantages are that they are easy to write and are human understandable. Figure 6.1 shows a sample snippet for buying a product, which is only eight lines of text. Process snippets are abstract descriptions of how a certain use case should be completed in an application. They are not explicitly developed for one target application. Instead, they are versatile enough to be applied to multiple applications. The given *buy* snippet consists of simple clicks and is building upon other sub-snippets such as *Login* and *Adding Address*.

Developing these snippets is straightforward. With a general understanding how shopping is done, a human can easily write step-by-step list of instructions, roughly comparable to a how-to description. Sub-processes, such as the authorization, can be encapsulated in their own process snippets and then be referenced and reused in other use cases. While parsing a given list of process snippets, each line is treated as a single Action-Target-Data triple or a previously defined process snippet.

Each line starts with an action (e.g. a verb such as *click, goto, enter, or do*) followed by a target descriptor in natural language and optional data in form of a string. By splitting each line into verb, noun and an optional object descriptor, a list of these statements is parsed into a list of executable ATDs. These will guide the crawler through the application use cases. Actions with no data (i.e. no object in the parsed instruction) are typically translated into simple clicks or long-clicks. Input fields or drop-down menus require a data argument. This characteristic will later narrow down the target space during execution. Rather than providing a perfect syntactic match (e.g. “Add product to Amazon shopping cart”), it is beneficial to keep descriptors abstract (e.g. “Add to Cart”). This allows them to be applied on a broader set of applications.

6.2.2 ATD Candidate Selection

To derive the best matching ATD for any interactive widget, we have to compute the *semantic similarity* to all specified ATDs as described in Section 3.2. Note, that not all action types may be applicable to a specific widget, e.g. you cannot enter text on a button, but only into input fields. Taking this into account, the set of ATDs is filtered by its specified action, i.e. ATDs with optional data (i.e. intended to data into the application) are only allowed for editable widgets and *click* or *goto* only for (long-)click-able widgets.

From this filtered list, the ATDs are sorted in descending order by the calculated semantic similarity value. These may now be executed on the desired interactive widget. This methodology also allows us to steer the exploration towards the desired functionality. Imagine the current application screen does not contain any use case represented in our test set. The exploration now either can fall back to a default random strategy or identify a potential target which is somewhat semantically similar to one of our use cases. By calculating the semantic similarity to all ATDs we also identify “Create your Amazon account” in the provided example as a potential start point for other use cases. In case there is no better match for completing the “Authentication” use case, our strategy will select this candidate. Only if all candidates identified by the semantic similarity are already explored (i.e. the calculated similarities are below the specified threshold t), we fall back to one of the unmapped targets by random (i.e. we ignore the cut-off value). At this point, there is no ATD target in the current state and we have to discover the use case in the application first. This is essential to allow for deeper exploration progress if the process description is incomplete or imprecise.

6.2.3 Guiding Exploration

So far, we have explained how to determine ATD candidates for a given UI screen. Now, we show how to guide a crawler to explore states, related to our process snippets.

We implemented testing with process snippets as an exploration strategy in DM-2 [14] to replace its default random strategy. Furthermore, we extended the code base such that DM-2 supports the presented extensions for *accessible* properties and made it capable to interact with out-of-screen elements. DM-2 is able to distinguish different states based on the currently visible UI elements and assigns an unique identifier *state-id* to each state. It also assigns a unique *widget id* to each widget, which allows us to determine which ATD candidates we already interacted with and which are still unexplored. The higher the similarity value of an ATD candidate is, the better the chances that executing the action leads to the desired result. However, if we solely choose the next action by the highest similarity that would mean we would always do the same action in the same state. We do not only want to explore that single ATD candidate but rather prioritize our target options. This is done by adding a weighting function based on how often we already interacted with this candidate. Additionally, whenever we see a state for the very first time we first apply all enter text ATDs and press enter. In most processes, the data fields have to be filled before the user can progress into the next state. The ATD candidates for each state are managed in a priority queue, whereas the priority is computed based on the candidates weight and similarity value.

Finally, we keep track of the novelty of the discovered UI screens to detect if we are stuck in a certain use case. In case the process snippet descriptions are incomplete or a mismatch occurs, we might get stuck in certain UI screens. Instead, we restart the application and start over if the text content of the UI screen did not change for the last ten actions.

6.3 Evaluation

We integrated our approach into the already mentioned DM-2 framework [14]. DM-2 is an extensible state-of-the-art app crawling framework with a default random exploration strategy. DM-2 allows us to analyze the UI hierarchy of every screen, as well as the state model of the AUT, and define our own exploration strategy. DM-2's default random strategy is extracting all actionable targets in an application state and randomly selects the next target to interact with. DM-2 crawls through an application by randomly clicking keyboard elements if an input field is focused. First of all, this is slower than sending all keys at once. Secondly, it rarely creates valid inputs. For fairness reasons, we extended this basic strategy to integrate a dictionary of inputs. DM-2 can fill input fields by either randomly generating a string or picking an input from a provided list. For the following evaluation, this list of inputs contains all *data* entries of the parsed process snippets. DM-2 can thus fill input fields with legit values only by chance, but has no deeper knowledge or understanding what these inputs mean. We henceforth refer to this strategy as the "*random*" strategy.

To evaluate the effectiveness of testing with process snippets, we conducted a set of empirical studies on fifty top applications of the Google Play Store from the domains business, communications, education, food, shopping, social, and travel. These domains thematically group applications regarding their intended use cases and are offered by the Play Store itself. Details of the tested application are presented in Table 6.1. Out of this test set, six apps are incompatible with DM-2 and are dropped from the analysis. After presenting the evaluation setup, we present the empirical analyses, which compare the new approach against random testing with a dictionary. The analyses focus on the following research questions:

RQ 9 (Testing Core Functionality) *Do process snippets improve the testing of core functionality? How does it compare in terms of efficiency?*

We analyze the performance of testing with process snippets (the topic-driven approach) and compare it in practice on a diverge set of fifty popular real world Android appli-

Table 6.1: Averaged benchmark overview on AUTs grouped by their domain for both random runs *r*, and guided *p*. *depth* indicates how far the exploration reached into the application. *widgets* is the number of covered widgets, *states* the number of covered app states and the number of actions executed per exploration on average

	App Name	$depth_p$	$depth_r$	$widgets_p$	$widgets_r$	$states_p$	$states_r$
eCommerce	com.amazon.mShop	17	17	1298	1120	171	204
	com.joom	8	24	521	679	178	185
	com.thehomedepot	17	25	941	837	169	177
	com.walmart.android	26	44	1469	1800	193	265
	com.ebay.mobile	8	7	1423	531	93	38
Food	com.hellofresh	22	20	633	753	206	193
	com.india.foodpanda	20	11	371	259	122	77
	com.global.foodpanda	5	6	33	169	15	25
	com.mcdonalds.app	9	8	404	483	44	43
	com.starbucks.mobilecard	9	9	372	143	117	65
	hu.viala.newiapp	1	5	32	92	3	29
	otlob.UI	20	18	680	776	143	178
Travel	com.booking	28	45	1162	1411	219	357
	com.cleartrip.android	18	15	1068	884	186	132
	com.takatrip.android	13	27	1593	8583	115	330
	io.wifimap.wifimap	12	27	653	961	74	182
	com.tourego.tourego	7	7	242	317	19	41
	com.railyatri.in.mobile	12	12	1167	935	152	101
	net.skyscanner	12	6	863	458	81	37
	com.google.earth	9	16	1039	903	142	139
Social	com.blued.international	2	1	45	24	8	3
	com.f2f.Gogo.Live	5	6	73	84	57	61
	com.facebook.katana	11	18	233	352	98	146
	com.facebook.lite	8	13	757	1282	49	98
	com.instagram.android	6	12	342	450	48	78
	com.pinterest	3	2	61	59	13	16
	com.snapchat.android	7	8	137	374	66	75
	com.tumblr	12	9	2151	751	87	49
	com.yy.hiyo	13	21	289	540	84	136
	com.zhiliaoapp.musically	9	15	256	1655	43	142
Education	com.ted.android	15	22	279	435	49	121
	com.duolingo	12	26	476	619	111	220
	com.youdao.hindict	28	22	1701	2041	185	196
	com.microblink.photomath	6	11	139	214	43	59
	com.memrise.android	4	11	71	240	9	52
Business	cl.taxibeat.driver	2	7	36	197	7	18
	com.application.onead	5	10	87	160	20	67
	com.facebook.pages.app	1	1	61	10	9	4
	com.mobisystems.office	18	14	946	464	154	87
	com.mydawa.MyDawa	3	6	41	82	9	23
	com.netqin.ps	5	6	42	45	15	17
	com.ubercab.driver	7	9	1368	1132	71	82
	com.vtg.app.mymytel	5	6	65	75	21	37
	team.dev.epro.apkcustom	27	20	543	263	160	130

cations. We check which core functionality is tested by chance (random testing) and measure the impact of our approach.

RQ 10 (Testing With Use Cases) *How does process snippet testing compare against random testing in terms of application coverage?*

Traditional software testing measures the performance of testing procedures by calculating the achieved coverage (e.g. covered lines of code, or covered blocks). In contrast, we are only able to observe the visible UI changes (black-box testing). As a replacement to code coverage, we count the number of states covered in the applications as well as the distinct UI elements that are displayed during an exploration.

In other words, we collect all UI states that we have ever seen during all explorations and then calculate the coverage of the individual runs.

RQ 11 (Use Case Generality) *How general are generated process snippets?*

Process snippets are intended to replace the need to repeat the effort to generate tests over and over again. We want to measure the reusability of process snippets and their applicability to a wide variety of applications.

6.3.1 Evaluation Setup

For the evaluation, we implemented a set of 64 process snippets, which are executed on every test subject. While this number seems fairly small, the process snippets encapsulate 362 testable actions (i.e. ATDs). Every application is tested on real Android devices (i.e. a set of Google Pixel phones running on stock Android 8) until the state model is either exhausted, i.e. DM-2 reports that there exist no further unexplored widgets, or a total time of thirty minutes has elapsed.

Borges et al. [14] have shown that DM-2 reaches 96% of its maximum coverage within 15 minutes. Accordingly, we consider thirty minutes to be a reasonable execution time to explore and test an application. In order to prevent overfitting in the random exploration, we repeated the random dictionary testing procedure ten times (ten times cross validation) and averaged the exploration results. This also counteracts the highly dynamic nature of the applications and allows us to compare the results on a stable basis. Thus, the random exploration can take five hours per application in total.

6.3.2 Testing Core Functionality

The core contribution of this paper is to introduce a new testing procedure which is supremely suited to test and follow a pre-defined set of use cases. In order to show the effectiveness of our approach, we manually verified each application and identified the correct interactive widgets for each ATD which is executable in the AUT. This ground truth serves as a reference to calculate the coverage and discovery speed of both random exploration as well as the process snippet exploration.

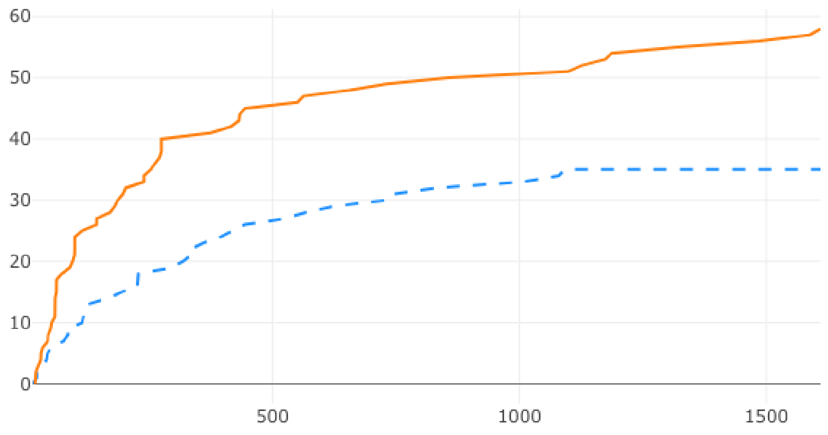
Figure 6.3 and Figure 6.4 summarize the results for each of the domains under test, i.e. the number of correctly tested core functions that are part of the set of process snippets we use for our evaluation. As already mentioned, we averaged the results over the repeated ten runs.

Immediately, one can see that snippet testing outperforms random tremendously in almost all categories and applications. The eight test apps of the food domain (Figure 6.3b) nicely demonstrate the potential of the demonstrated technique. Not only does it execute almost double the number of the core functions correctly, but also covers two thirds in the first four minutes. In contrast, random testing takes five times longer until it reaches this number. The likelihood of clicking the correct button or inserting the correct input text by chance is considerably low. Moreover, following the provided use cases allows more complex interaction patterns and actually allows to test deeper functionality. The eCommerce applications (Figure 6.6) and social applications (Figure 6.3c) show comparable results and can in the eCommerce domain actually proceed to insert valid credit card data or addresses. Notable outliers to the overall success of process snippet testing can be seen in the domains of travel apps (Figure 6.4b) and business apps (Figure 6.4c). For travel apps, the exploration charts show almost identical results when averaged over all applications. A manual inspection of the results showed us that even though the Play Store assigns apps into the travel domain the virtually share no use cases. There are star gazing apps, as well as applications for booking a taxi. In this domain, process snippets cannot play to its strength and cover more core functionality. The predefined set of use cases is too small.

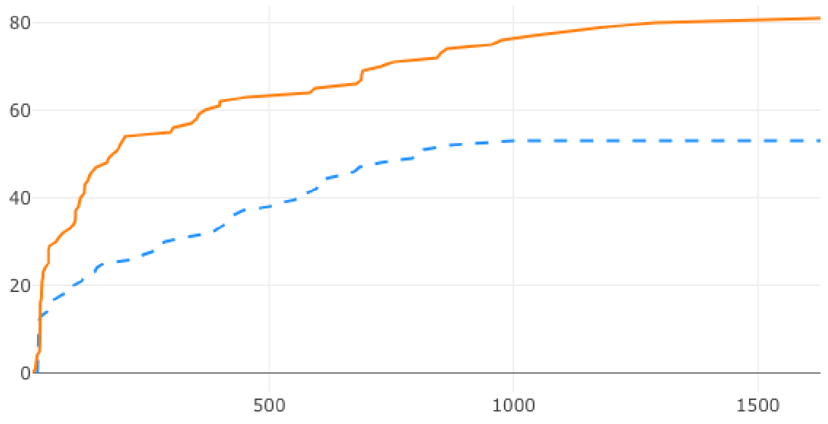
RQ 9 (Testing Core Functionality): Do process snippets improve the testing of core functionality? How does it compare in terms of efficiency?

☞ ***Yes. On average, testing with process snippets outperforms random testing in both function coverage and in effectiveness***

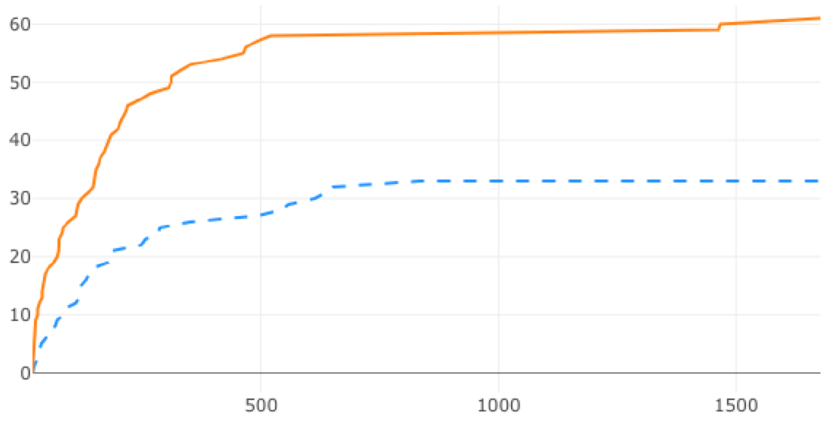
☞ ***Testing with process snippets discovers 30% more core functions in 20% of the time.***



(a) eCommerce

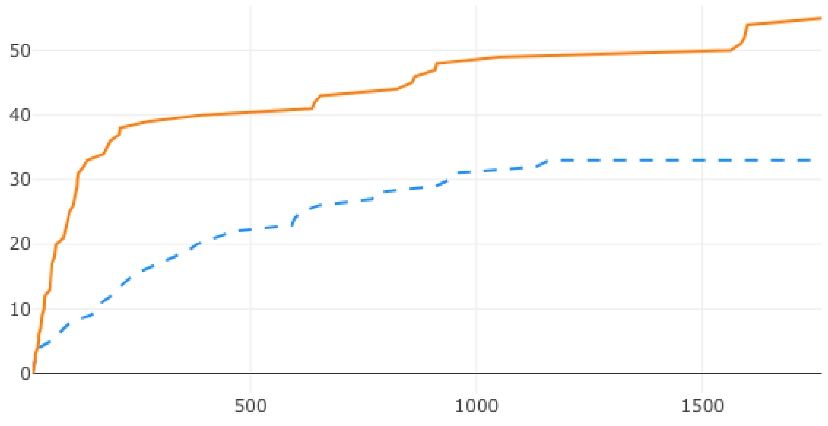


(b) Food

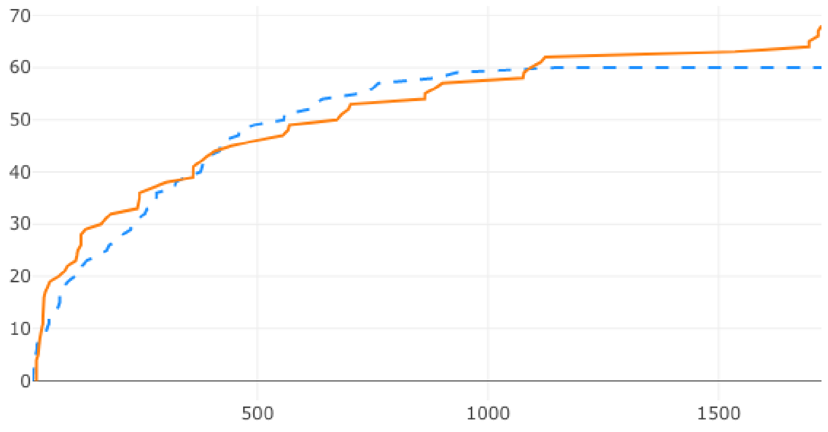


(c) Social

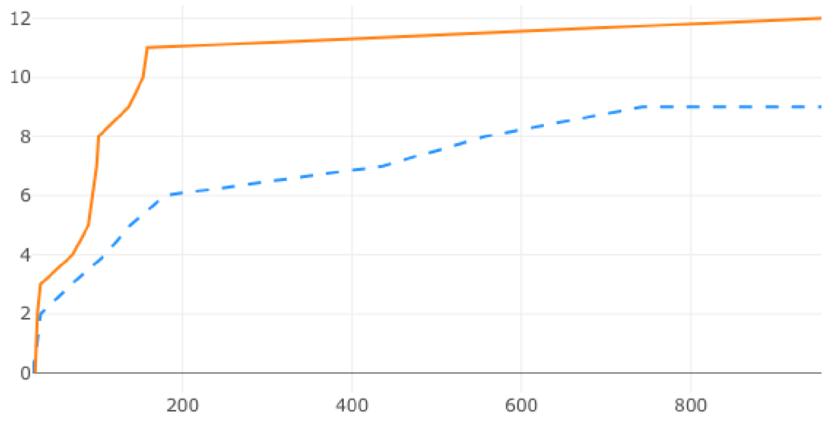
Figure 6.3: Averaged evaluation results for covering core functionality in the tested domains, showing how many ATDs (*y*-axis) where covered at which time (*x*-axis, values are in seconds). Process snippet runs are represented with solid orange lines; the blue dashed lines show the random runs.



(a) Education



(b) Travel



(c) Business/Office

Figure 6.4: Averaged evaluation results for covering core functionality in the tested domains, showing how many ATDs (*y*-axis) where covered at which time (*x*-axis, values are in seconds). Process snippet runs are represented with solid orange lines; the blue dashed lines show the random runs.

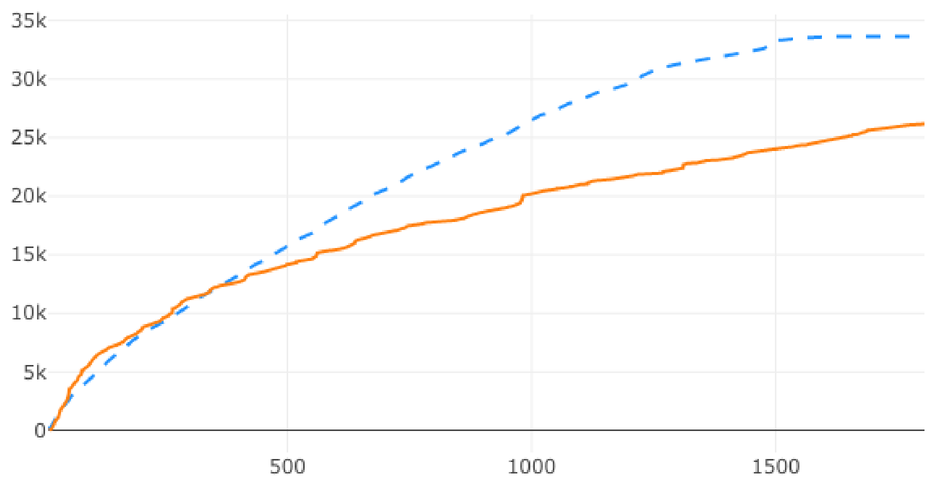


Figure 6.5: Average total widget coverage of all applications. Displays for both exploration types how many unique widgets (y-axis) where covered at a certain time (x-axis in seconds). Blue-dashed line are random runs, orange-solid line are TDT tests.

6.3.3 Efficiency of Testing with Process Snippets

In this section we evaluate how effective process snippets are for testing purposes and how it compares in terms of application coverage. Since all AUT are treated as a black box and most applications in the test set dynamically load content into WebViews objects, coverage cannot be measured by traditional code coverage metrics, such as line coverage or block coverage. Instead, we collect all exploration paths of all runs to build a supermodel. Using this model we can calculate the exploration depth and the number of discovered elements (i.e. widgets, which either have content or are interactive).

Table 6.1 shows the general exploration results and app statistics generated by both testing procedures. The value *depth* is based on the supermodel analysis and denotes the length of the longest sequence, which is necessary to cover a unique state in the application model (i.e. there exists no shorter path to cover this state). In other words it expresses the system penetration which is achieved during testing. The *widgets* columns describe how many relevant UI elements were discovered during the exploration and *states* describes how many states were revealed during the exploration. Together, these values can be interpreted as an alternative to traditional coverage metrics. The *actions* columns describe how many actions the crawler executed on average. Together with the overall execution time, this metric is representing the effectiveness of the crawler and

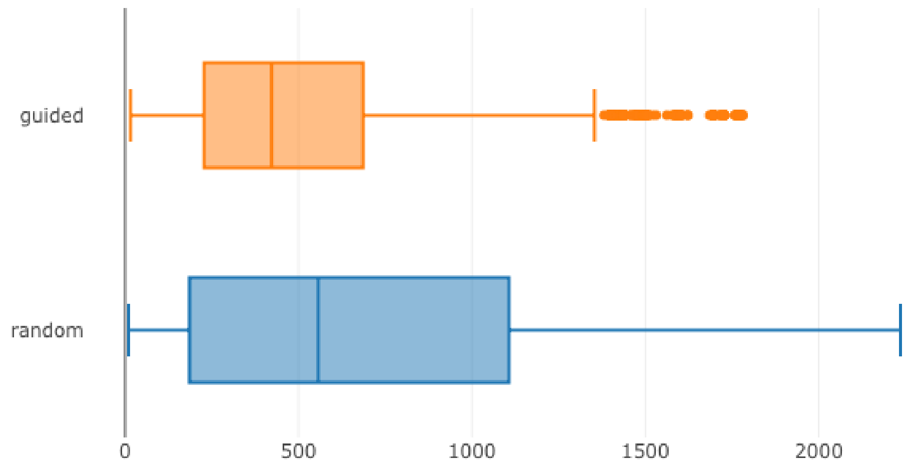


Figure 6.6: Average distribution of discovery times of unique widgets: performance analysis comparing the averaged results of baseline testing (random with dictionary, dashed blue) against guided (testing with process snippets, solid orange). The *x*-axis shows the exploration timeline in seconds.

allow us to argue about the effort to required to reach core functionality or maximize the app coverage.

Overall, none of the testing procedures is outperforming the other in all categories or applications. Random testing is on average more successful in terms of pure widget/state coverage. The previous analysis showed that testing with process snippets reaches *deeper* core functionality. Still, this has no measurable impact on actually testing *more* functionality. A *broad* testing procedures allows to discover many application parts (i.e. widget coverage). To emphasize this, Figure 6.5 shows the overall results of all runs averaged over the ten repetitions for all applications under test. It shows for both test methods how many interactive widgets, or widgets with text content were discovered at which time. Guided testing concentrates on fulfilling the specified process snippets. Instead, random testing clicks as many different widgets as fast as possible. The tested real world applications feature large exploration spaces and rapidly clicking through them reveals lots of widgets, states, and reachable functionality.

Another interesting analysis is the distribution in which different runs reveal new widgets over time (see Figure 6.6) The random runs discover new widgets in a more or less normal distribution. In contrast, the box plot shows that the process snippet runs reach a 75% quartile within ten minutes, half the time the random runs take. Repeating the guided exploration also shows less divergence in the discovered widgets, i.e. the runs are more deterministic. On the downside—when considering absolute coverage—the process snippet exploration is too focused on following process snippets. While the main goals, i.e. (i) testing core functionality, and (ii) creating comparable exploration traces, are achieved, these numbers also show potential for further improvements: testing with process snippets reaches a peak after about ten minutes of exploration. If the budget allows for further testing, an alternative approach (fall back to random) can improve coverage.

One final word must be said about measuring the crawling efficiency in terms of execution time. Execution time can be highly subjective and is heavily impacted by the performance of the test infrastructure, e.g. the test device, network latencies, or system load. By repeating the exploration ten times and averaging the results over all test runs, the effect of the latter should be minimized. External factors such as network latencies or system load should not be present in different runs. For completeness, we also evaluated the number of test actions that were executed by the individual exploration strategies. Table 6.2 shows the number of test actions for both random runs ($actions_r$) and guided test runs $actions_p$. Bold values indicate significant differences in the number of executed actions. Averaged, the differences are insignificant, though. Only one fifth of the applications shows major differences (i.e. the bold values), but there is no clear evidence indicating that random testing or guided testing are on average executing less actions during the exploration and thus is more efficient. In pure number of actions, none of the approaches is superior. Only in relation to the achieved widget coverage or achieved functionality coverage one can argue about efficiency. The previous observations do not change. Guided testing does cover more core functions, random testing covers more widgets—with a comparable number of actions.

These observations allow us to answer RQ 10 (Testing With Use Cases): How does process snippet testing compare against random testing in terms of application coverage?

☞ *Testing with process snippets is inferior to random testing in terms of pure application coverage.*

☞ *Random testing allows for **broad** system tests (cheap), while snippet testing concentrates on **deep** functionality (expensive).*

☞ *The number of executed crawl actions is comparable in both crawling strategies.*

Table 6.2: Averaged benchmark overview on the tested application set for random runs (r) and guided runs (p). *actions* shows the number of exploration actions executed. *correct* and *incorrect* describe the percentage of correctly vs. incorrectly executed ATD actions. Number in braces are the absolute values.

App Name	$actions_p$	$actions_r$	*correct*	*incorrect*
com.amazon.mShop	842	834	87% [398]	12% [57]
com.joom	597	**951**	100% [139]	0% [0]
com.thehomedepot	945	1019	62% [86]	37% [52]
com.walmart.android	629	852	90% [274]	9% [30]
com.ebay.mobile	633	652	85% [126]	14% [22]
com.hellofresh	1165	858	94% [454]	5% [27]
com.india.foodpanda	1094	1019	84% [378]	15% [69]
com.global.foodpanda	**1310**	824	87% [590]	12% [82]
com.mcdonalds.app	789	649	87% [174]	12% [25]
com.starbucks.mobilecard	**1114**	808	78% [471]	21% [155]
hu.viala.newiapp	**1573**	1294	53% [230]	46% [199]
otlob.UI	1144	758	100% [174]	0% [0]
com.bookin	742	881	93%[125]	6% [9]
com.cleartrip.android	661	522	75% [57]	25% [19]
com.takatrip.android	594	**1110**	90% [224]	9% [24]
io.wifimap.wifimap	477	719	98% [267]	1% [4]
com.tourego.tourego	121	**766**	88% [39]	12% [5]
com.railyatri.in.mobile	693	882	100%[75]	0% [0]
net.skyscanner	645	**1203**	88% [31]	11% [4]
com.google.earth	404	326	84% [65]	15% [12]
com.blued.international	410	429	0% [0]	0% [0]
com.f2f.Gogo.Live	750	758	87% (309)	12% (43)
com.facebook.katana	1291	1167	59% [209]	40% [144]
com.facebook.lite	934	904	0% [0]	0% [0]
com.instagram.android	**1422**	795	78% [70]	21% [19]
com.pinterest	51	409	75% [66]	25% [11]
com.snapchat.android	1108	1342	96% [816]	4% [34]
com.tumblr	724	928	71% [117]	29% [46]
com.yy.hiyo	720	875	80% [83]	19% [20]
com.zhiliaoapp.musically	768	688	79% [59]	20% [15]
com.ted.android	232	270	100% [92]	0% [0]
com.duolingo	903	913	90% (448)	9% (47)
com.youdao.hindict	618	785	88% [103]	11% [14]
com.microblink.photomath	351	432	100% [167]	0% [0]
com.memrise.android	29	678	90% [10]	10% [1]
cl.taxibeat.driver	630	366	0% [0]	0% [0]
com.application.onead	1316	1175	100% [5]	0% [0]
com.facebook.pages.app	**609**	346	50% [150]	50% [150]
com.mobisystems.office	614	757	100% [105]	0% [0]
com.mydawa.MyDawa	807	**1371**	99% [671]	1% [3]
com.netqin.ps	**562**	81	100% [260]	0% [0]
com.ubercab.driver	754	**922**	60% [90]	40% [61]
com.vtg.app.mymytel	1278	1091	97% [167]	2% [5]
team.dev.epro.apkcustom	691	810	93% [42]	20% [15]

6.3.4 Generalization Analysis

Generating process snippets is supposed to be a one time effort and should improve testing for apps independent of their domain or the syntactic content of their natural language text. The initial testing effort for developing test cases for non-trivial apps is significant. Developing use cases is comparably trivial, since it requires only a basic understanding how a system is supposed to work. In a way, it is comparable to defining acceptance test descriptions. While conducting the previously presented studies (especially generating the SELENIUM test suites in Section 5.3), the main effort was spent on identifying *common* application behavior, since we followed the assumption that a test transfer was the ultimate goal.

In the study about process snippets executed on Android apps, the assumptions are weaker. Essentially, we generated a set of use case descriptions and more or less executed *all of them* on the test subjects, regardless whether they were actually designed for the AUT or not. The most important observations so far are the efficiency improvements as well as the coverage of core functionality compared to random testing. In addition to that, we investigate how many process snippets we can effectively execute and measure prediction precision, i.e. if process snippet is executed on the correct targets.

We look into the data presented in Table 6.2. The *correct* / *incorrect* columns indicate our manually inspected results on the execution success of the ATDs supplied for testing. The great majority of applications show very high success rates with values above 85%. There are some applications that could not be tested such as the `com.facebook.lite`, `com.blued.international`, or the `cl.taxibeat.driver` app. We were not able to discover executable ATD targets for semantic testing. The required functionality is bound to an existing account, e.g. a Facebook account, that requires additional authentication steps. In these scenarios, we fall back to the random exploration strategy.

For the majority of the remaining applications, the semantic interpretation of the process snippets instructions has a high precision and thus can improve testing. In a nutshell, this means that guiding the *exploration due to our ATD analysis is very successful*. Especially the domains eCommerce and food apps show high results across the board. These results conform with the previous observations about core function coverage (Figure 6.3a and Figure 6.3b). The high number of executions (absolute numbers) together with the high precision is correlated to the efficiency of testing core functionality. The domains ‘education’ and ‘social’ are not far off, but the individual results are more diverse. This effect is in turn observable in the overall results about the core function coverage. The average advantage of the guided testing is not as good as in the domains eCommerce and food.

The absolute numbers (i.e. the number in square brackets) are quite diverse as well and range between $[5,590]$. A high number indicates that the guided testing repeatedly executed many ATD actions. For the majority of the applications the guided approach effectively matches a high number and thus positively affects the core functionality coverage.

For the remaining applications with low absolute execution numbers, we manually inspected the application states and narrow down these differences to three main reasons:

Miscategorization. The majority of applications with low absolute values do not contain most of the use cases tested. While Google labels an application such as *skyscanner* as a travel app, it does not feature typical use cases such as booking a flight.

Language. The semantic similarity for the target identification is too low, i.e. is cut off by our similarity metric. In some applications the interactive elements do not have a descriptive text but are just presenting an icon instead, the language is changed from English to another language, or there is no semantic match.

Application Complexity. Some of the use cases are located deep in the application, or are unreachable due to two-factor authentication or comparable security measures. These limitations are out of scope for our prototype.

Despite the differences in the absolute execution numbers, the high precision values also indicate that extending the set of process snippets (i.e. developing new use cases) would be beneficial for testing. Even if a certain function does not exist in the target application, topic-driven testing would rather not execute it, if it cannot be clearly identified. The high precision implies that a use case is only executed if the correct target is present.

The developed set of use cases was arguably small and only covered 64 use cases. The use cases are publicly available [68] and lay a foundation for further development. Mining a broad scope of use case descriptions into process snippets enables us to test more application behavior. Thus, topic-driven testing can serve both academical and industrial purposes.

These observations allow us to finally answer **RQ 11 (Use Case Generality)**: *How general are the generated process snippets?*

☞ *Although not all applications feature the complete set of use cases, the target descriptors can be interpreted and executed semantically with a high average precision of 85%.*

☞ *Further completion of the set of process snippets would be beneficial for further testing. The low number of falsely executed process snippets (high precision) indicates that we would spend more time on the test execution and a negative effect of having more test cases is not to be expected.*

☞ *Extending the list of use cases would allow for better system penetration and cover more functionality.*

6.4 Threats to Validity

As for every empirical study, there are certain threats to its validity to be faced. Most of them overlap with the studies about employing topic-driven testing on web applications. First and foremost threated is the feature/use case selection itself. We selected 64 use cases for the evaluation, which cover distinct functionality. This set is of course far from complete, which introduces bias into all our evaluation. The evaluation cannot argue about the behavior and use cases that is not in the test set. There might exist use cases that cannot be expressed in natural language texts. We can also not argue that we can semantically match further use case descriptions.

Additionally, there is no way to measure how many features exist in the AUT and calculate the test coverage in the real world applications. They are all treated as a black-box and the testing framework can only observe changes on the client side. The evaluation is not intended to produce a test environment, but instead is intended to steer testing towards desired functionality. We have seen that the absolute number for the correct ATD execution differed and the manual inspection could only analyze the main reasons for this shortcoming. The generality of the feature transfer might be a threat for the same reasons. To reduce bias in the presented dataset, we captured a significant set of typical features within the applications based on the most common use cases or testing scenarios. The high precision of executing ATDs on the target subjects gives enough credible evidence to support the assumption that we also can transfer other process snippets.

RQ 9 (Testing Core Functionality) concluded that our guided exploration discovers more of the targeted relevant core functionality in a shorter time period. The presented analysis again neglects the fact that the tested use case set is not complete. This might lead to a better evaluation of the guided procedure. Imagine that there exist more of the simplistic use case descriptions, such as “Go to Startpage”. It might require a simple click on a button that is present on every page. What are the chances of this button to be clicked by a random crawler? The answer is: very likely. Choosing similar use cases that require a broad exploration instead of complex interaction schemes might positively affect the performance and efficiency of random crawling. In defense of the topic-driven testing-approach it can only be said that integrating a random approach into our technique is still possible. Remembering the exploration time of the unique widgets (Figure 6.6) and core functions, the data clearly indicates that the highest benefit of test guidance is achieved in the first few minutes of the exploration. The remainder of the exploration time can be used for further—broad—explorations.

Although the experiments were run on fifty real world applications out of different domains, all featuring a distinct functionality, the scope of this analysis is far from being complete. There exist domains, say for instance *file sharing tools*, which offer features for writing, storing, sharing, or printing documents, all functionalities not expressed in the given application set. Furthermore, certain features are just not represented with natural language content and therefore cannot be used by this approach for analysis purposes. A common example would be the representation of buttons with icons or images, video players, or other overlaying guiding methods. Currently, these functions would only be tested by chance, i.e. by falling back to the underlying default strategy. In our experiments, the chosen use cases mostly are represented with descriptive labels next to the target element and there exist description texts for most icons.

Anything beyond this behavior is out of the scope of this analysis at this moment. Topic-driven testing could be extended to manage these limitations, though. A possible solution might be the analysis of alternative tags in images (i.e. *alt*), which is often displayed for the sake of accessibility. An alternative to this is learning image descriptions, i.e. by training a neural network on a dataset of labeled images [37].

Finally, the speedups we report are all based on models, making simplified assumptions about how crawlers do and can operate. Our models are as conservative as possible, modeling and reporting worst-case scenarios for our technique. Real-world crawlers operating on real-world web applications may perform differently, sporting heuristics to focus on specific UI elements first, and leverage similarity between pages to perform better. All these optimizations are orthogonal to our approach and could be combined with it; we kept them out of our evaluation for better control.

6.5 Process Snippets for Testing

At this point, I would like to discuss the observations made during the empirical studies for applying topic-driven testing on the domain of Android testing. By changing the input to the procedure from existing test suites into even more general use case description, topic-driven testing can effectively guide a crawling procedure towards relevant core function. It outperforms basic random crawling both in terms of speed as well as overall coverage. Our topic-driven technique covers twice the amount of the to be tested functions in a fifth of the time.

Topic-driven testing guides testing towards *deep* functionality and is superior to random testing in this aspect. While random testing explores an app quickly and covers broad set of functions, targeting the exploration towards relevant (deep) functionality is not possible. Since the guided approach reaches the majority of its coverage in the first minutes an integration with existing testing approaches seems promising, if exploration time is of minor interest. Industrial testing typically intended to be efficient—reach and test relevant functionality fast. The presented testing procedure achieves that goal on a broad set of applications, but industrial approaches require further research to get more reliable results.

For academic purposes though, topic-driven testing can already serve as an efficient test generator. First of all, testing can be steered towards relevant functionality. Second, the process snippets are executed on the targets they are designed for—as indicated by the high precision. Further tests are likely also executed on the same targets.

Finally, the produced exploration traces are more deterministic than those produced by random explorations. Putting these benefits together, a wide range of applications can be tested with minimal manual effort. The produced results are reproducible and the generated application models can be compared to learn common behavior. Especially an analysis that is focused on identifying common or uncommon behavior or testing an assumption on a large set of applications is made possible through semantic testing.

7 Conclusion and Open Research Challenges

The tremendous increase in the number of app releases every day requires further improvement and research of automated testing procedures. Apps have to be available, executable, testable, and maintained on multiple platforms at once. Figuratively, every day there is a new device on the market with different hardware, screen size, hardware buttons, or sensors. In the end though, all these devices and applications are used by human users. The performance, functionality, and usability of an application is judged by the user. This dissertation presented *topic-driven testing* as a novel testing technology that tests applications close to the human experience. Central to the presented technique is a semantic interpretation of the application functionality as well as test instruction. This stands in contrast to existing state-of-the-art techniques that interpret test instructions and the structure of the application by syntactic means.

In Software Engineering, *abstraction* and *reuse* have been the two most prominent drivers of productivity in the past two decades. *Process snippets* apply these two principles in the domain of app testing. Taking a list of natural language use case description to guide exploration to the desired functions, testing with process snippets is by far superior to random testing in both system penetration and covering relevant core functionality. Compared to random exploration, our evaluation on fifty of the most popular Android apps out of six domains shows that testing with process snippets can cover 30% more functions within 20% of the time. Representing abstract process steps, process snippets can be easily reused to test one application after another.

In my personal opinion, topic-driven testing has the potential to be the foundation of a whole new testing procedure. It can be integrated into systematic UI testing and is complementary to state-of-the-art frameworks. Existing general purpose app testing frameworks heavily depend on random actions for exploration. While powerful, they rarely test the core functionality and are not robust against even minor app changes. Compared to random testing, there is still some manual effort included in setting up the testing procedures. But especially the results in the section about *testing with process snippets* indicates that this effort scales nicely, if the testing is executed on a large set of applications. For the majority of the applications, testing with process snippets is far more efficient then random testing. The test setup and creation of testable snippets is a matter of hours. Thus, topic-driven testing can have an impact on industrial testing methods.

In a nutshell, this thesis makes the following contributions to improve mining and understanding software features and automated user interface testing:

Context Analysis of Interactive Elements Applying visual clustering and analyzing the alignment of text elements in relation to interactive elements allows to link the available UI actions with their descriptive texts.

Semantic Feature Identifcation This dissertation describes a new methodology to infuse features with their semantic meaning. Each interactive element is assigned to a certain position in the word vector space that is determined by its descriptive texts in the googleNewsVector.

By observing the word vector space of the descriptive texts in five hundred web applications, the analysis identifies areas and groups of semantically close descriptions from which we can automatically learn common features in these applications. Using semantic text similarity as a similarity metric, these features can be effectively identified and matched across applications.

The developed libraries to calculate the semantic similarity are publicly available and can be integrated into other test frameworks:

`https://github.com/amrast/webtest`

Automatic Test Transfer We presented ATTABOY, a prototype tool for transferring web application SELENIUM tests across applications. ATTABOY executes a given SELENIUM test suite and mines the features of the GUI in the source application. The gathered knowledge is transferred to other applications using semantic text similarity as a metric. The transferred knowledge is specifically tailored on the one applications the tests were originally written for. While a complete test transfer (especially with an oracle) is not achieved, the empirical evaluation shows the potential to guide testing towards relevant functionality *seven times faster than random testing*. This comparison might not be fair, but the execution and setup costs to guide testing in the target application using ATTABOY is comparable to random testing—virtually non-existent.

ATTABOY is not capable of achieving a full test transfer, it can merely check if the application reaches similar UI states during the test case execution. This is not sufficient for a full test automation. The main limiting factor is that topic-driven testing cannot automatically transfer or generate an oracle required for testing, so far.

Testing with Process Snippets I presented a novel methodology, called *testing with process snippets*, which takes a list of natural language use case description to guide exploration to the desired functions. Testing with process snippets was implemented into the existing Android testing framework DM-2. Evaluated on fifty real-world application from the Google Play Store, testing with process snippets is by far superior to random testing approaches in both system penetration and covering relevant core functionality.

The empirical study shows that 30% more core functions are tested within only 20% of the execution time, i.e. within a few minutes. While random testing explores more software parts (i.e. reaches a higher widget coverage), process snippets can effectively control the exploration and reliably guide it to relevant system parts. The remaining 90% of the exploration time can still be used to fall back to a default random strategy, if coverage maximization is the intended goal. In addition, our evaluation shows initial evidence that our methodology is applicable on a large set of domains and their domain specific use cases.

Program/Application Understanding Our analysis offers insights into how applications are generally structured and how they integrate functionality. The semantic concepts that express features and their corresponding UI elements are grouped together in the work vector space and are uniformly expressed independent of the application. While the position (*x*-*y* coordinates) and the descriptive texts can vary significantly, the inner workflows of a use case are typically similar in order to facilitate the usability of an application.

In summary, this dissertation makes advances in the state-of-the-art UI testing by introducing a novel way to express similarities between applications and their functionality. The semantic interpretation of features allows to integrate and develop techniques complementary to existing state of the art tools. To enable researchers and developers to benefit from this work, the source code, data sets, and empirical evaluation tools used and developed in this thesis are made publicly available for download.

7.1 Lessons Learned

I would like to summarize a few general, informal, and subjective observations made during the implementation and evaluation of the central tools developed in this dissertation. My personal experience with industry in Germany indicates that companies try to integrate software testing procedures into their continuous integration and release workflow, but the adaptation of automated software testing tools is at the very beginning. Especially in the area of acceptance testing, an automated testing solution is still in its infancy. For most parts—not only in acceptance testing—testing is still done with a high manual effort. In my personal opinion, the effort to setup decent and effective test infrastructures is too costly at the moment, since testing is not integrated *while* or *before* a product is developed, but afterwards. In any case, software is then tested from the point of view of a developer rather than an end-user.

Random testing techniques are widely used. Fuzzing, i.e. probe a system with random inputs, is also a technique used in industry to test a system. Random UI testing though does not produce reliable and reproducible results and is unsuitable for many important testing goals, such as regression testing. Nevertheless, setting up and integrating a decent testing procedure into the development cycle can be costly. It is often substituted by manual testing, which in turn can be error prone.

ATTABOY

With this motivation in mind, ATTABOY has been developed. Reducing the initial cost of generating software tests by learning from existing systems and their test suites was the key idea behind ATTABOY. The prototype was evaluated on real-world subjects in order to get a realistic view on the potential in an industrial test setting.

In my opinion, the strength of ATTABOY is not the test generation for new applications. An effective test generation requires further research. The incomplete oracle generation is a major limitation. Especially in the area of regression and multi-platform testing, ATTABOY can still play to its strengths. It guides testing towards relevant core functionality faster. Getting reasonably large tests suites is still a problem though.

Testing with Process Snippets

While there is a lot of potential for future research to improve ATTABOY, we put the main focus on an even simpler test setup and showed that the principle of topic-driven testing is independent of the underlying test platform or test device. The results in Chapter 6 tell their own story. With minor effort, testing can be reliably guided towards relevant core functionality. The generated traces are not purely random, but more stable and reliable instead.

Random testing covers more UI states and there are good examples (e.g. fuzzing) where random testing is a suitable method. More traditional testing that is part of a deployment cycle needs to be reliable, though. We want certain parts of the application to be tested well and there should be the option to follow certain use cases. This dissertation already put enough emphasis on the motivation and reasons, why random testing is incapable of achieving this goal. The results show, that topic-driven testing can improve testing in this area.

7.2 Open Research Challenges

There is one central final question unanswered. How far is topic-driven testing evolved or in other words how far is it solving the ongoing problem of testing?

Modern interactive applications offer so many interaction opportunities that automated exploration and testing is practically impossible without some guidance towards relevant functionality. We therefore propose a testing procedure that uses semantic similarity to link UI elements to their semantic meaning. By doing so, testing can reuse existing tests from other applications to effectively guide exploration towards semantically similar functionality or can follow simple human understandable instructions that express use cases. This method is highly effective: Rather than spending hours or days exploring millions of redundant paths, our guidance allows to discover deep functionality in only a few steps. In the long run, this means that one needs to write test cases for only one representative application in a particular domain (“select a product”, “choose a seat”, “book a flight”, etc.) and automatically reuse and adapt these test cases for any other application in the domain, leveraging and reusing the domain experience in these test cases.

Despite these advances, topic-driven testing does not solve the problem of automated testing mainly due to the limitations introduced by the missing oracle translation, yet. On top of that I would like to point out further research opportunities that are enabled by the topic-driven testing. Central to this research is the capability to identify features in the GUI of an application and match these in other applications regardless of the underlying platform or technology. This is a central contribution of this dissertation. The empirical evaluations show satisfying results and thus lay a foundation for further research.

Future work and research could be focused on the following topics:

Oracles. As of now, process snippets only represent interaction steps, but do not check the resulting app state; they share this "oracle problem" with the bulk of test generation approaches. Our matching approach is able to match actions that lead to specific functionality; however, we cannot check whether the particular action was successful or not. Although this problem is shared by all test generators (which had therefore better be called *execution generators*), we have the opportunity not only to reuse actions, but also the oracles in existing test cases. The challenge here is to identify, match, and adapt appropriate criteria for success or failure.

The semantic approach of process snippets also offers opportunities for generic oracles, expressing conditions such as "Order completed" which would match the wide variety of "Thank you" messages indicating success, but not "Verify credit card info", "Order declined", and other states indicating failure. We are currently investigating how to learn *common behavior* from a large set of application, i.e. derive common observations after a process snippet has been executed. Since process snippets are general enough to be tested on a large base, we can learn the expected behavior.

In my personal opinion, the oracle problem cannot be completely solved. A standard example would be the putting random items into a shopping cart and calculating the total price. An automated testing procedure would not only need to learn the meaning of *sum*, but also check positional properties, e.g. the sum is below the list of items. In essence, this example can be made arbitrary complex and explains why a complete automation is unlikely. Instead, I see the possibility for a semi-automated process. As of now, we can generate better executions with the proposed process snippets. Likewise, there is the possibility to identify which part of the exploration graph serves which use case. By infusing this semantic meaning into an exploration trace a developer can easily identify the shopping cart and integrate further test steps or an oracle.

Outlier Detection. Process snippets also can be useful for detecting *outliers*—that is, applications whose process deviates from the norm. If an application uses process steps (or labels for these steps) that are highly unusual, such deviations could be detected automatically; this also applies to oracles, as sketched above. In the long run, classifying applications by their support for common process snippets could lead to quick detection of user interaction issues—without having to run expensive experiments with humans.

Our previously conducted experiments [7] shows the potential of our technique to detect security and privacy issues. By learning the purpose of UI elements, we can learn which access to sensitive information is *expected* and which is suspicious. Extending this idea towards other policies seems to be straightforward. Observing how a majority of applications behaves (e.g. internal behavior, resource access, device load, timing properties, or styling) allows to for outlier detection and maybe automated sandboxing.

Other GUI Frameworks. Right now, our work focuses on Android apps and web applications. However, we are planning to generalize it to other GUI domains. Web applications are a particularly interesting target, as a web page typically holds more textual and contextual information, which a process snippet can semantically match. We have seen that the impact of more text (web application) in contrast to a small screen size (mobile device) is less significant than expected. This raises the question if (a) the testing can be generalized for all UI applications and (b) if we can learn or find differences that are introduced due to incompatibility of a device with the back-end.

Alternate Input Models. So far, our approach leverages test suites or process snippets as a source to guide test generation. Each input expresses a single use case. Human experience, though, is formed from many examples, and experience with multiple user interfaces helps interacting with the next one. We are looking into techniques that allow us to further abstract over multiple examples to guide test generation. Process snippets could make great helpers for *voice-driven systems.* Eventually, one should be able to successfully tell a smart speaker "Go to Expedia and book me a flight to Xi'an, China" and have the speaker automatically find the correct UI interactions based only on process snippets—regardless of whether specific commands or command patterns ("skills") are already installed. Likewise, we can imagine assistive technologies and automated services do the right thing on interfaces that were originally designed for humans.

The presented testing methodology was designed to be further reused for both academic and industrial purposes. We imagine it to be used for quickly generating reproducible and comparable exploration models. In order to achieve this, all the data referred to in this paper is publicly available for download and experimentation. The package includes our set of testable process snippets, the exploration graphs, the application APKs, the ground truth and the test data to reproduce our results.

Feature Detection and Program Understanding. Basic UI design visually groups together elements that belong to the same feature to reduce the difficulty of a target selection task (*Fitt's law* [23]). While these element groups are tight, the position of each group is not fixed. We mined the semantic meaning of interactive UI elements (Chapter 3) and showed that their describing labels are also close in the word vector space. The denseness of these features in the vector space over five hundred applications shows the generality.

Using this method the other way around, we can re-identify features in applications and even explain them in natural language. This is the basis for program understanding. Exploration graphs of applications tend to grow quickly into hundreds of states and thousands of edges, which goes beyond human understanding. I see the potential of topic-driven testing to gather program understanding and explain these exploration graphs or at least parts of them. With this information, a developer can gain understanding on how different components of a system are bound together, where additional testing effort should be spent, or even identify unexpected application behavior.

Glossary

DROIDMATE-2 A testframe to test Android applications through the GUI developed by Jamrozik et al. [34] and later extended by Borges et al. [14].

SELENIUM is a software testing framework for web applications. It offers a HTTP interface to communicate and control a variety of test browsers for testing without the need for browser specific commands.

`xpath` is a unique identifier which identifies a UI element in the DOM.

Action Target Data Tuple Action Target Data Tuples describe machine executable statements. *Action* typically describes the type of actions which should be executed (e.g. click, hover, etc.), *Target* the interactive element on which the action should be executed upon, and *Data* an optional data string which serves as input.

Android An operating system developed by Google Inc. which is mainly run on mobile devices such as smart phones, smart watches or tablets.

Android Debug Bridge (ADB) is a software interface for Android systems and devices used to connect them to a computer over USB or a wireless connection. It allows to execute commands on the device and transfer data between device and computer.

application programming interface (API) is a set of clearly defined methods defining the communication between various system components.

Application Under Test (AUT) Describes a software application which is tested.

Asynchronous JavaScript and XML (AJAX) is a set of web development techniques. Typically, the code of a web application is split into the server part (which is run on the back-end server of the application provider) and a client part, which is executed in the browser of the user. It allows to refresh the content by running client-side computations instead of requesting a new HTML page for every page..

cascading style sheet (CSS) is a style sheet language used for describing the presentation parameters of a markup language, e.g. HTML.

Continuous Integration In software engineering, continuous integration (CI) is the practice of merging all developer working copies to a shared mainline several times a day to prevent integration problems. It is typically combined with automatically running tests on each code change.

Document Object Model The Document Object Model (DOM) is an interface that unifies HTML, XHTML, or XML documents in a logical tree structure. Each *node* contains objects, i.e. elements with attached information or event handlers. In web applications the HTML structure that represents GUI can be expressed as a DOM document.

finite state machine (FSM) is a mathematical model of computation that can be expressed as a finite graph. Each *node* in the graph represents a state of a system, while the *edges* describe the actions to transfer the state from on state to another..

Frames per Second Frames per Second (FPS) describes how often the pixels on a screen are rendered per second.

Graphical User Interface The interface of a program shown to the user. It replaces abstract command structures into a graphical representation and allows users to interact with the program through graphical icons and visual indicators.

HTTP archive record (HAR) is a file format that stores network communication between client and remote host. Each outgoing and incoming network request is stored as an entry and contains information about the type of request, the request content (e.g. payload, headers, etc) as well as the response (including response code, MIME-type etc).

Human Machine Interaction (HMI) is a research area that investigates the ways in which between humans interact with computers and vice versa..

information retrieval (IR) describes the process of obtaining information from a set of resources, documents or texts.

Internet of Things Internet of things (IoT) is the network of physical devices, vehicles, home appliances, and other items embedded with electronics, software, sensors, actuators, and connectivity which enables these things to connect, collect and exchange data.

Latent Dirichlet allocation (LDA) is a generative statistical model used for topic extraction of documents.

Levenshtein distance is a string metric to measure the difference between two sequences. The Levensthein distance measrues how many single-character edits (i.e. insertions, deletions or substitutions) are required to transform one word to the other..

MAchine Learning for LanguagE Toolkit (Mallet) framework [50], a popular Java-based machine learning tool kit.

Natural Language Processing (NLP) is a subfield of computer science that investigates how computers process and analyze large amounts of natural language data and includes tasks such as speech recognition, natural language understanding, and natural language generation..

Part-of-Speech Tagging Is a standard natural language processing technique that identifies the role of words (be it verb, object, etc.) in a given document..

singular value decomposition (SVD) is a factorization method for a matrix (be it real or complex). It is a generalization of the eigendecomposition that allows for an analysis of non-symmetric matrices..

System Under Test (SUT) Describes a software system which is tested..

t-Distributed Stochastic Neighbor Embedding (t-SNE) is a technique for dimensionality reduction that is especially suited for the visualization of high-dimensional datasets..

term frequency-inverse document frequency (tf-idf) is a numerical statistic that reflect the importance of a word in a document. In information retrieval this measure is widely used as a weighting factor to extract the relevant topics including, or generate text summarizations of documents..

topic-driven testing (TDT) is a novel testing technique presented in this dissertation. It integrates NLP analysis into UI testing to allow for more precise testing procedures and allows to compare applications based on the UI..

UI Automator UI Automator is an Android UI testing framework suitable for cross-app functional UI testing across system and installed apps..

User Interface The interface of a program software that a human uses to interact with a system.

Vision-based Page Segmentation (VIPS) is a technique that allows to extract the semantic structure for a web page. It groups together DOM elements due to their visual proximity it allows for efficient information retrieval..

Web Application TEst generator (WATEG) is an automated testing tool for Web applications presented by Thummalapenta et al. [77] that verifies a pre-defined set of business rules on an AUT..

word vector model (word2vec) is a model trained on text corpora. Each word is modeled as an *n* dimensional vector in a vector space. Words which are occuring together are placed into neighbouring regions in this vector space.

word2vec-GoogleNews-vector The word2vec-GoogleNews-vector is a publicly available pre-trained 300-dimensional english word vector model. It has been trained on Google News data that featured more than 3 billion running words. The resulting corpus size of the GoogleNews-vectors-negative300.bin.gz [29] models more than 3 million words..

World Wide Web Consortium (W3C) is an international community to develop Web standards..

Acronyms

DROIDMATE-2 A testframe to test Android applications through the GUI developed by Jamrozik et al. [34] and later extended by Borges et al. [14]. DROIDMATE-2. .

Action Target Data Tuple Action Target Data Tuples describe machine executable statements. *Action* typically describes the type of actions which should be executed (e.g. click, hover, etc.), *Target* the interactive element on which the action should be executed upon, and *Data* an optional data string which serves as input. Action Target Data Tuples. .

Android Debug Bridge (ADB) is a software interface for Android systems and devices used to connect them to a computer over USB or a wireless connection. It allows to execute commands on the device and transfer data between device and computer. android debug bridge. .

application programming interface (API) is a set of clearly defined methods defining the communication between various system components. application programming interface. .

Asynchronous JavaScript and XML (AJAX) is a set of web development techniques. Typically, the code of a web application is split into the server part (which is run on the back-end server of the application provider) and a client part, which is executed in the browser of the user. It allows to refresh the content by running client-side computations instead of requesting a new HTML page for every page.. Asynchronous JavaScript and XML. .

cascading style sheet (CSS) is a style sheet language used for describing the presentation parameters of a markup language, e.g. HTML. cascading style sheet. .

Continuous Integration In software engineering, continuous integration (CI) is the practice of merging all developer working copies to a shared mainline several times a day to prevent integration problems. It is typically combined with automatically running tests on each code change. Continuous Integration. .

DFS Depth First Search.

Document Object Model The Document Object Model (DOM) is an interface that unifies HTML, XHTML, or XML documents in a logical tree structure. Each *node* contains objects, i.e. elements with attached information or event handlers. In web applications the HTML structure that represents GUI can be expressed as a DOM document. Document Object Model. .

finite state machine (FSM) is a mathematical model of computation that can be expressed as a finite graph. Each *node* in the graph represents a state of a system, while the *edges* describe the actions to transfer the state from on state to another.. finite state machine. .

FMAP Feature Matching Across Platforms.

Frames per Second Frames per Second (FPS) describes how often the pixels on a screen are rendered per second. Frame per Second. .

GUI Graphical User Interface.

GUI Graphical User Interface.

HTTP archive record (HAR) is a file format that stores network communication between client and remote host. Each outgoing and incoming network request is stored as an entry and contains information about the type of request, the request content (e.g. payload, headers, etc) as well as the response (including response code, MIME-type etc). HTTP archive record. .

Human Machine Interaction (HMI) is a research area that investigates the ways in which between humans interact with computers and vice versa.. human machine interface. .

information retrieval (IR) describes the process of obtaining information from a set of resources, documents or texts. information retrieval. .

Internet of Things Internet of things (IoT) is the network of physical devices, vehicles, home appliances, and other items embedded with electronics, software, sensors, actuators, and connectivity which enables these things to connect, collect and exchange data. Internet of Things. .

Latent Dirichlet allocation (LDA) is a generative statistical model used for topic extraction of documents. Latent Dirichlet Allocation. .

MAchine Learning for LanguagE Toolkit (Mallet) framework [50], a popular Java-based machine learning tool kit. MAchine Learning for LanguagE Toolkit. .

Natural Language Processing (NLP) is a subfield of computer science that investigates how computers process and analyze large amounts of natural language data and includes tasks such as speech recognition, natural language understanding, and natural language generation.. Natural Language Processing. .

Part-of-Speech Tagging Is a standard natural language processing technique that identifies the role of words (be it verb, object, etc.) in a given document.. Part-of-Speech Tagging. .

singular value decomposition (SVD) is a factorization method for a matrix (be it real or complex). It is a generalization of the eigendecomposition that allows for an analysis of non-symmetric matrices.. singular value decomposition. .

System Under Test (SUT) Describes a software system which is tested.. application under test. . system under test. .

t-Distributed Stochastic Neighbor Embedding (t-SNE) is a technique for dimensionality reduction that is especially suited for the visualization of high-dimensional datasets.. t-Distributed Stochastic Neighbor Embedding. .

term frequency-inverse document frequency (tf-idf) is a numerical statistic that reflect the importance of a word in a document. In information retrieval this measure is widely used as a weighting factor to extract the relevant topics including, or generate text summarizations of documents.. term frequency-inverse document frequency. .

topic-driven testing (TDT) is a novel testing technique presented in this dissertation. It integrates NLP analysis into UI testing to allow for more precise testing procedures and allows to compare applications based on the UI.. topic-driven testing. .

Vision-based Page Segmentation (VIPS) is a technique that allows to extract the semantic structure for a web page. It groups together DOM elements due to their visual proximity it allows for efficient information retrieval.. Vision-based Page Segmentation. .

WATEG Web Application TEst generator.

word vector model (word2vec) is a model trained on text corpora. Each word is modeled as an *n* dimensional vector in a vector space. Words which are occuring together are placed into neighbouring regions in this vector space. word vector model. .

word2vec-GoogleNews-vector The word2vec-GoogleNews-vector is a publicly available pre-trained 300-dimensional english word vector model. It has been trained on Google News data that featured more than 3 billion running words. The resulting corpus size of the GoogleNews-vectors-negative300.bin.gz [29] models more than 3 million words.. word2vec-GoogleNews-vector. .

World Wide Web Consortium (W3C) is an international community to develop Web standards.. World Wide Web Consortium. .

Bibliography

[1] M. Elgin Akpinar and Yeliz Yesilada. Vision based page segmentation algorithm: Extended and perceived success. In *Revised Selected Papers of the ICWE 2013 International Workshops on Current Trends in Web Engineering - Volume 8295*, pages 238–252, New York, NY, USA, 2013. Springer-Verlag New York, Inc. URL: `http://dx.doi.org/10.1007/978-3-319-04244-2_22`, `doi:10.1007/978-3-319-04244-2_22`.

[2] Inc Alexa Internet. Alexa top 500 web pages. `http://www.alexa.com/topsites/category/Top`, 2017. Accessed: 2017-08-04.

[3] Manuel Álvarez, Alberto Pan, and Juan Raposo. Using clustering and edit distance techniques for automatic web data extraction. *Web Information Systems ...*, page 13, 2007.

[4] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. Using GUI ripping for automated testing of Android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 258–261, New York, NY, USA, 2012. ACM. URL: `http://doi.acm.org/10.1145/2351676.2351717`, `doi:10.1145/2351676.2351717`.

[5] Android Developers. Monkeyrunner. *https://developer.android.com/studio/test/monkeyrunner/*, August 2012.

[6] Fiora T. W. Au, Simon Baker, Ian Warren, and Gillian Dobbie. Automated usability testing framework. In *Proceedings of the Ninth Conference on Australasian User Interface - Volume 76*, AUIC '08, pages 55–64, Darlinghurst, Australia, Australia, 2008. Australian Computer Society, Inc. URL: `http://dl.acm.org/citation.cfm?id=1378337.1378349`.

[7] Vitalii Avdiienko, Konstantin Kuznetsov, Isabelle Rommelfanger, Andreas Rau, Alessandra Gorla, and Andreas Zeller. Detecting behavior anomalies in graphical user interfaces. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 201–203. IEEE, 2017.

[8] Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *the 2013 ACM SIGPLAN international conference*, pages 641–660, New York, New York, USA, 2013. ACM Press.

[9] Ziv Bar-Yossef and Sridhar Rajagopalan. Template detection via data mining and its applications. *WWW*, page 580, 2002.

[10] Marco Baroni, Georgiana Dinu, and Germán Kruszewski. Don't count, predict! A systematic comparison of context-counting vs. context-predicting semantic vectors. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 238–247, Stroudsburg, PA, USA, 2014. Association for Computational Linguistics.

[11] Giovanni Becce, Leonardo Mariani, Oliviero Riganelli, and Mauro Santoro. Extracting widget descriptions from GUIs. In Juan de Lara and Andrea Zisman, editors, *Fundamental Approaches to Software Engineering*, pages 347–361, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[12] F. Behrang and A. Orso. Test migration between mobile apps with similar functionality. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 54–65, 2019. `doi:10.1109/ASE.2019.00016`.

[13] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent Dirichlet Allocation. *Journal of Machine Learning Research*, 3(4-5):993–1022, 2012. `arXiv:1111.6189v1`, `doi:10.1162/jmlr.2003.3.4-5.993`.

[14] Nataniel P Borges Jr, Jenny Hotzkow, and Andreas Zeller. DroidMate-2: A platform for Android test generation. In *2018 33nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Sep 2018. `doi:10.1145/3238147.3240479`.

[15] Andreas Bruns, Andreas Kornstadt, and Dennis Wichmann. Web application tests with selenium. *IEEE software*, 26(5), 2009.

[16] Deng Cai, Shipeng Yu, J R Wen, and W Y Ma. VIPS: a vision-based page segmentation algorithm. *Beijing Microsoft Research Asia*, pages 1–29, 2003. URL: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.118.638{&}amp;rep=rep1{&}amp;type=pdf`, `doi:MSR-TR-2003-79`.

[17] Soumen Chakrabarti, Martin van den Berg, and Byron Dom. Focused Crawling - A New Approach to Topic-Specific Web Resource Discovery. *Computer Networks*, 31(11-16):1623–1640, 1999.

[18] Wontae Choi, George Necula, and Koushik Sen. Guided GUI testing of Android apps with minimal restart and approximate learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '13, pages 623–640, New York, NY, USA, 2013. ACM. URL: `http://doi.acm.org/10.1145/2509136.2509552`, `doi:10.1145/2509136.2509552`.

[19] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated Test Input Generation for Android - Are We There Yet? *ASE*, pages 429–440, 2015.

[20] Valentin Dallmeier, Martin Burger, Tobias Orth, and Andreas Zeller. WebMate: Generating test cases for web 2.0. *Software Quality. Increasing …*, 2013. URL: `http://link.springer.com/chapter/10.1007/978-3-642-35702-2{_}5`.

[21] Günes Erkan and Dragomir R Radev. Lexrank: Graph-based lexical centrality as salience in text summarization. *Journal of artificial intelligence research*, 22:457–479, 2004.

[22] Mattia Fazzini, Martin Prammer, Marcelo d'Amorim, and Alessandro Orso. Automatically translating bug reports into test cases for mobile apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 141–152, New York, NY, USA, 2018. ACM. URL: `http://doi.acm.org/10.1145/3213846.3213869`, `doi:10.1145/3213846.3213869`.

[23] Paul M Fitts. The information capacity of the human motor system in controlling the amplitude of movement. *Journal of experimental psychology*, 47(6):381, 1954.

[24] Yasser Ganjisaffar. Crawler4j website. `https://github.com/yasserg/crawler4j`, March 2016.

[25] Anne Geraci. *IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries*. IEEE Press, Piscataway, NJ, USA, 1991.

[26] Yihong Gong and Xin Liu. Generic Text Summarization Using Relevance Measure and Latent Semantic Analysis. *SIGIR*, pages 19–25, 2001.

[27] Shuai Hao, Bin Liu, Suman Nath, William G J Halfond, and Ramesh Govindan. PUMA. In *the 12th annual international conference*, pages 204–217, New York, New York, USA, 2014. ACM Press.

[28] Gang Hu, Linjie Zhu, and Junfeng Yang. Appflow: Using machine learning to synthesize robust, reusable ui tests. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 269–282, New York, NY, USA, 2018. Association for Computing Machinery. URL: `https://doi.org/10.1145/3236024.3236055`, `doi:10.1145/3236024.3236055`.

[29] Google Inc. Google news data set: Googlenews-vectors-negative300.bin.gz. `https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit?usp=sharing`, Last Accessed on 2018-10-29.

[30] Aminul Islam and Diana Inkpen. Semantic text similarity using corpus-based word similarity and string similarity. *ACM Transactions on Knowledge Discovery from Data*, 2(2):1–25, 2008. URL: `http://www.site.uottawa.ca/~diana/publications/tkdd.pdf`, `doi:10.1145/1376815.1376819`.

[31] Aminul Islam and Diana Inkpen. Semantic text similarity using corpus-based word similarity and string similarity. *ACM Trans. Knowl. Discov. Data*, 2(2):10:1–10:25, July 2008. URL: `http://doi.acm.org/10.1145/1376815.1376819`, `doi:10.1145/1376815.1376819`.

[32] Melody Y. Ivory and Marti A Hearst. The state of the art in automating usability evaluation of user interfaces. *ACM Comput. Surv.*, 33(4):470–516, December 2001. URL: `http://doi.acm.org/10.1145/503112.503114`, `doi:10.1145/503112.503114`.

[33] Ivar Jacobson. *Object-oriented software engineering: a use case driven approach*. Pearson Education India, 1993.

[34] Konrad Jamrozik, Philipp von Styp-Rekowsky, and Andreas Zeller. *Mining sandboxes*. ACM, New York, New York, USA, May 2016.

[35] Konrad Jamrozik and Andreas Zeller. Droidmate: a robust and extensible test generator for android. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*, pages 293–294. ACM, 2016.

[36] Anjali Ganesh Jivani et al. A comparative study of stemming algorithms. *Int. J. Comp. Tech. Appl*, 2(6):1930–1938, 2011.

[37] Andrej Karpathy and Li Fei-Fei. Deep visual-semantic alignments for generating image descriptions. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[38] Christian Kohlschütter and Wolfgang Nejdl. A densitometric approach to Web page segmentation. *Proceedings of the 17th ACM conference on Information and knowledge management*, pages 1173–1182, 2008. `doi:10.1145/1458082.1458237`.

[39] Peter Kolb. Disco: A multilingual database of distributionally similar words. *Proceedings of KONVENS-2008, Berlin*, 156(2003):37–44, 2008.

[40] Harold W Kuhn. The Hungarian method for the assignment problem. *50 Years of Integer Programming 1958-2008*, pages 29–47, 2010.

[41] Tessa Lau, Clemens Drews, and Jeffrey Nichols. Interpreting written how-to instructions. *IJCAI International Joint Conference on Artificial Intelligence*, 10443:1433–1438, 2009.

[42] M. Leotta, A. Stocco, F. Ricca, and P. Tonella. Using multi-locators to increase the robustness of web test cases. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, April 2015. `doi:10.1109/ICST.2015.7102611`.

[43] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. DroidBot: a lightweight UI-Guided test input generator for Android. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 23–26. IEEE, April 2017.

[44] Jun-Wei Lin, Farn Wang, and Paul Chu. Using semantic similarity in crawling-based web application testing. In *Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on*, pages 138–148. IEEE, 2017.

[45] Surabhi Lingwal. Noise Reduction and Content Retrieval from Web Pages. *International Journal of Computer Applications*, 73(4):24–30, 2013.

[46] Hans Peter Luhn. The automatic creation of literature abstracts. *IBM Journal of research and development*, 2(2):159–165, 1958.

[47] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for Android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 224–234. ACM, 2013.

[48] Nataniel P. Maintainers: Borges Jr and Jenny Hotzkow. Droidmate 2 - online repository. `https://github.com/uds-se/droidmate`, 2019.

[49] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[50] Andrew Kachites McCallum. Mallet: A machine learning for language toolkit. http://www.cs.umass.edu/ mccallum/mallet, 2002.

[51] Ali Mesbah, A. van Deursen, and D. Roest. Invariant-based automatic testing of modern Web applications. *IEEE Transactions on Software Engineering*, 38(1):35–53, jan 2012. URL: `http://dl.acm.org/citation.cfm?id=1555037http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5728834`, `doi:10.1109/TSE.2011.28`.

[52] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web*, 6(1):1–30, mar 2012. URL: `http://dl.acm.org/citation.cfm?doid=2109205.2109208http://crawljax.com/publications/`, `doi:10.1145/2109205.2109208`.

[53] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB)*, 6(1):3:1–3:30, 2012.

[54] Rada Mihalcea and Paul Tarau. Textrank: Bringing order into text. In *Proceedings of the 2004 conference on empirical methods in natural language processing*, 2004.

[55] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013. URL: `http://arxiv.org/abs/1301.3781`.

[56] Gary Miner, John Elder IV, and Thomas Hill. *Practical text mining and statistical analysis for non-structured text data applications*. Academic Press, 1 edition, January 2012.

[57] Guido Minnen, John Carroll, and Darren Pearce. Applied morphological processing of english. *Natural Language Engineering*, 7(3):207–223, 2001.

[58] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[59] Narayanan Palani. *Selenium Webdriver: Software Automation Testing Secrets Revealed Part 2*. eBooks2go, 2016.

[60] Roger S Pressman. *Software engineering: a practitioner's approach*. Palgrave Macmillan, 2005.

[61] Gang Qian, Shamik Sural, Yuelong Gu, and Sakti Pramanik. Similarity between euclidean and cosine angle distance for nearest neighbor queries. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, SAC '04, pages 1232–1237, New York, NY, USA, 2004. ACM. URL: `http://doi.acm.org/10.1145/967900.968151`, `doi:10.1145/967900.968151`.

[62] Quantcast. Quantcast Top 500 Web page Ranking. (visited: 2018-04-10). URL: `https://www.quantcast.com/top-sites/`.

[63] Václav Rajlich and Norman Wilde. The role of concepts in program comprehension. In *Program Comprehension, 2002. Proceedings. 10th International Workshop on*, pages 271–278. IEEE, 2002.

[64] Andreas Rau. Topic-driven testing. In *2017 IEEE/ACM 39th International Conference Software Engineering Companion (ICSE-C)*, pages 409–412. IEEE, 2017.

[65] Andreas Rau. Testing with process snippets data repository with code, snippets and evaluation data, 2018. `https://www.dropbox.com/sh/klxanq5w0os6f9u/AACB9M5GoZpLYtWySTk6q7nPa`, Last Accessed on 2018-10-23.

[66] Andreas Rau, Jenny Hotzkow, and Andreas Zeller. Efficient GUI test generation by learning from tests of other apps. In *ACM/IEEE 40th International Conference on Software Engineering: Companion Proceedings*, pages 370–371, New York, New York, USA, 2018. ACM Press.

[67] Andreas Rau, Jenny Hotzkow, and Andreas Zeller. Transferring tests across web applications. In *International Conference on Web Engineering (ICWE)*, pages 50–64. Springer, 2018.

[68] Andreas Rau, Jenny Hotzkow, and Andreas Zeller. “Add to Cart, Proceed to Checkout”: Composing Tests from Natural Language Snippets. *ACM SIGSOFT 28th International Symposium on Software Testing and Analysis (ISSTA 2019)*, 2019 Under Submission.

[69] Andreas Rau, Maximilian Reinert, and Andreas Zeller. Automatic test transfer across applications. Technical report: `https://www.st.cs.uni-saarland.de/projects/attaboy/`, Chair of Software Engineering, Saarbrücken, Germany, 2016.

[70] Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA.

[71] Davi de Castro Reis, Paulo Braz Golgher, Altigran Soares Silva, and AlbertoF Laender. Automatic web news extraction using tree edit distance. In *Proceedings of the 13th international conference on World Wide Web*, pages 502–511. ACM, 2004.

[72] Shauvik Roy Choudhary, Mukul R. Prasad, and Alessandro Orso. Cross-platform feature matching for Web applications. *Issta*, pages 82–92, 2014. URL: `http://dl.acm.org/citation.cfm?id=2610384.2610409`, `doi:10.1145/2610384.2610409`.

[73] Raimondas Sasnauskas and John Regehr. Intent fuzzer: crafting intents of death. In *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*, pages 1–5. ACM, 2014.

[74] Sara Sprenkle, Lori Pollock, Holly Esquivel, Barbara Hazelwood, and Stacey Ecott. Automated oracle comparators for testing Web applications. In *Software Reliability, 2007. ISSRE’07. The 18th IEEE International Symposium on*, pages 117–126. IEEE, 2007.

[75] Josef Steinberger and Karel Jezek. Using latent semantic analysis in text summarization and summary evaluation. *Proc. ISIM*, 4:93–100, 2004.

[76] Suresh Thummalapenta, Pranavadatta Devaki, Saurabh Sinha, Satish Chandra, Sivagami Gnanasundaram, Deepa D Nagaraj, Sampath Kumar, and Sathish Kumar. Efficient and change-resilient test automation: An industrial case study. *Pro-*

ceedings - International Conference on Software Engineering, pages 1002–1011, 2013.

[77] Suresh Thummalapenta, K Vasanta Lakshmi, Saurabh Sinha, Nishant Sinha, and Satish Chandra. Guided Test Generation for Web Applications. *Proceedings of the 35th International Conference on Software Engineering - ICSE 2013*, pages 162–171, 2013.

[78] Suresh Thummalapenta, Saurabh Sinha, Nimit Singhania, and Satish Chandra. Automating test automation. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 881–891, Piscataway, NJ, USA, 2012. IEEE Press. URL: `http://dl.acm.org/citation.cfm?id=2337223.2337327`.

[79] Kristina Toutanova, Dan Klein, Christopher Manning, William Morgan, Anna Rafferty, Michel Galley, and John Bauer. Stanford log-linear part-of-speech tagger, 2000.

[80] Laurens Van Der Maaten and G E Hinton. Visualizing high-dimensional data using t-sne. *Journal of Machine Learning Research*, pages 2579–2605, 2008.

[81] Mario Linares Vásquez, Kevin Moran, and Denys Poshyvanyk. Continuous, Evolutionary and Large-Scale - A New Perspective for Automated Mobile App Testing. *Proceedings of the Seventh IEEE International Conference on Computer Vision*, cs.SE, 2018.

[82] Karane Vieira, Altigran Soares da Silva, Nick Pinto, Edleno Silva de Moura, João M B Cavalcanti, and Juliana Freire. A fast and robust method for web page template detection and removal. *CIKM*, page 258, 2006.

[83] W3C. Http archive format specification [online]. 2018. URL: `https://w3c.github.io/web-performance/specs/HAR/Overview.html`.

[84] World Wide Web Consortium (W3C). Html 5.1 specification for interactive elements, 2016. URL: `https://www.w3.org/TR/2016/WD-html51-20160412/interactive-elements.html`.

[85] Chaw Su Win and Mie Mie Su Thwin. Web Page Segmentation and Informative Content Extraction for Effective Information Retrieval. *International Journal of Computer Communication Engineering Research (IJCCER)*, 2(2), March 2014.

[86] Q. Xin, F. Behrang, M. Fazzini, and A. Orso. Identifying features of android apps from execution traces. In *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 35–39, 2019. doi:10.1109/MOBILESoft.2019.00015.

[87] Wei Yang, Mukul R. Prasad, and Tao Xie. A grey-box approach for automated gui-model generation of mobile applications. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*, FASE'13, pages 250–265, Berlin, Heidelberg, 2013. Springer-Verlag. URL: http://dx.doi.org/10.1007/978-3-642-37057-1_19, doi:10.1007/978-3-642-37057-1_19.

[88] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. Droidfuzzer: Fuzzing the android apps with intent-filter tag. In *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*, page 68. ACM, 2013.

[89] Lan Yi, Bing Liu, and Xiaoli Li. Eliminating noisy information in Web pages for data mining. *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '03*, pages 296–305, 2003.

[90] L. Yujian and L. Bo. A normalized levenshtein distance metric. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(6):1091–1095, June 2007. doi:10.1109/TPAMI.2007.1078.

Appendices

Evaluation Data - Quantcast Top 500 US Pages

For the feasability study we retrieved a subset of the Quantcast Top Million U.S. Web Sites. The most recent rankings are estimated as of Dec 05, 2019. This data is subject to terms of use described at `https://www.quantcast.com/learning-center/quantcast-terms/website-terms-of-use/`

Rank	Site	Rank	Site
1	google.com	27	realtor.com
2	facebook.com	28	linkedin.com
3	amazon.com	29	ranker.com
4	youtube.com	30	quizlet.com
5	en.wikipedia.org	31	urbandictionary.com
6	twitter.com	32	wordpress.com
7	yahoo.com	33	spotify.com
8	ebay.com	34	usps.com
9	nytimes.com	35	legacy.com
10	reddit.com	36	glassdoor.com
11	yelp.com	37	chase.com
12	target.com	38	cnet.com
13	walmart.com	39	giphy.com
14	buzzfeed.com	40	stackexchange.com
15	paypal.com	41	cnbc.com
16	wikia.com	42	whitepages.com
17	bing.com	43	bustle.com
18	apple.com	44	etsy.com
19	pinterest.com	45	quora.com
20	netflix.com	46	thepennyhoarder.com
21	adobe.com	47	instagram.com
22	espn.com	48	rumble.com
23	live.com	49	nbcnews.com
24	craigslist.org	50	npr.org
25	bestbuy.com	51	groupon.com
26	weather.com	52	webmd.com

Rank	Site	Rank	Site
53	capitalone.com	79	eonline.com
54	dailymail.co.uk	80	lifehacker.com
55	imdb.com	81	247sports.com
56	fandango.com	82	fedex.com
57	imgur.com	83	healthyway.com
58	theguardian.com	84	wellsfargo.com
59	businessinsider.com	85	drudgereport.com
60	tripadvisor.com	86	thehill.com
61	go.com	87	wikihow.com
62	gizmodo.com	88	icloud.com
63	goodreads.com	89	wsj.com
64	microsoft.com	90	latimes.com
65	cbsnews.com	91	xfinity.com
66	macys.com	92	snopes.com
67	ups.com	93	gfycat.com
68	att.com	94	politico.com
69	zillow.com	95	uproxx.com
70	pagesix.com	96	usatoday.com
71	bankofamerica.com	97	hulu.com
72	nydailynews.com	98	topix.com
73	merriam-webster.com	99	bedbathandbeyond.com
74	indeed.com	100	opentable.com
75	nfl.com	101	twentytwowords.com
76	lowes.com	102	thekitchn.com
77	fashionbeans.com	103	rollingstone.com
78	bleacherreport.com	104	jcpenney.com

Rank	Site	Rank	Site
105	slate.com	131	thoughtcatalog.com
106	today.com	132	wayfair.com
107	priceline.com	133	office.com
108	answers.com	134	overstock.com
109	nih.gov	135	newegg.com
110	people.com	136	homedepot.com
111	foodnetwork.com	137	healthline.com
112	allrecipes.com	138	airbnb.com
113	cheatsheet.com	139	sbnation.com
114	discover.com	140	spanishdict.com
115	definition.org	141	yellowpages.com
116	deviantart.com	142	mapquest.com
117	vimeo.com	143	mercurynews.com
118	comcast.net	144	jalopnik.com
119	aol.com	145	southwest.com
120	cbssports.com	146	topix.net
121	azlyrics.com	147	americanexpress.com
122	dmv.org	148	messenger.com
123	costco.com	149	expedia.com
124	theverge.com	150	gamefaqs.com
125	variety.com	151	avclub.com
126	nickiswift.com	152	vanityfair.com
127	deadspin.com	153	viralthread.com
128	citi.com	154	seriouseats.com
129	medicalnewstoday.com	155	rare.us
130	usnews.com	156	romper.com

Rank	Site	Rank	Site
157	walgreens.com	183	instructables.com
158	github.com	184	androidcentral.com
159	llbean.com	185	thechive.com
160	biblegateway.com	186	informationvine.com
161	stackoverflow.com	187	lifewire.com
162	nbc.com	188	mentalfloss.com
163	nbcsports.com	189	magiquiz.com
164	creditkarma.com	190	outlook.com
165	office365.com	191	looper.com
166	miamiherald.com	192	roblox.com
167	hometalk.com	193	startribune.com
168	mayoclinic.org	194	toysrus.com
169	kotaku.com	195	jezebel.com
170	city-data.com	196	theweek.com
171	thesun.co.uk	197	weather.gov
172	usmagazine.com	198	trulia.com
173	narvar.com	199	knowyourmeme.com
174	intuit.com	200	spectrum.net
175	dailycaller.com	201	shmoop.com
176	hp.com	202	cinemablend.com
177	imore.com	203	barnesandnoble.com
178	urbo.com	204	worldlifestyle.com
179	vox.com	205	getitfree.us
180	sfgate.com	206	redd.it
181	iheart.com	207	tomsguide.com
182	irs.gov	208	ebates.com

Rank	Site	Rank	Site
209	boredpanda.com	235	rawstory.com
210	cafemom.com	236	discordapp.com
211	dickssportinggoods.com	237	gamestop.com
212	t-mobile.com	238	thepiratebay.org
213	ca.gov	239	howtogeek.com
214	newarena.com	240	aa.com
215	ancestry.com	241	coupons.com
216	theroot.com	242	gamespot.com
217	medium.com	243	polygon.com
218	gofundme.com	244	edmunds.com
219	reuters.com	245	express.co.uk
220	adp.com	246	medicinenet.com
221	washingtonexaminer.com	247	sears.com
222	trend-chaser.com	248	nextdoor.com
223	gap.com	249	axs.com
224	msnbc.com	250	cvs.com
225	kbb.com	251	joinhoney.com
226	yourtango.com	252	drugs.com
227	apartmenttherapy.com	253	delish.com
228	verizon.com	254	dailymotion.com
229	blogger.com	255	redbubble.com
230	dropbox.com	256	popsugar.com
231	inspiremore.com	257	kohls.com
232	eventbrite.com	258	dell.com
233	flickr.com	259	ibtimes.com
234	staples.com	260	siriusxm.com

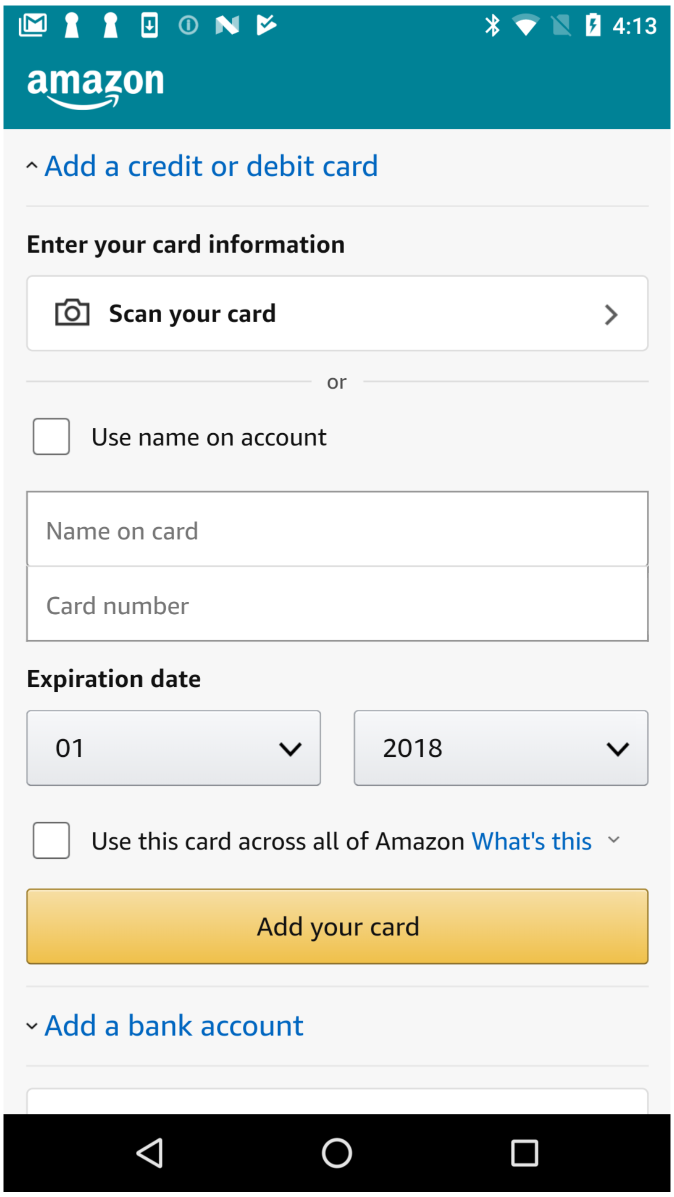
Rank	Site	Rank	Site
261	bravotv.com	287	nesn.com
262	lifedaily.com	288	petfinder.com
263	custhelp.com	289	excite.com
264	ikea.com	290	techradar.com
265	tmz.com	291	offers.com
266	ajc.com	292	chron.com
267	nasdaq.com	293	forever21.com
268	worldation.com	294	dominos.com
269	patch.com	295	bizrate.com
270	consumerreports.org	296	force.com
271	enotes.com	297	sportingnews.com
272	pizzabottle.com	298	v2profit.com
273	seccountry.com	299	wunderground.com
274	delta.com	300	coinbase.com
275	inverse.com	301	experian.com
276	cbslocal.com	302	battle.net
277	tasteofhome.com	303	ew.com
278	patient.info	304	bandcamp.com
279	tomshardware.com	305	booking.com
280	hollywoodreporter.com	306	macrumors.com
281	kiwireport.com	307	healthcare.gov
282	denverpost.com	308	patreon.com
283	ask.com	309	leafly.com
284	chicagotribune.com	310	tributes.com
285	distractify.com	311	archive.org
286	ibt.com	312	sciencealert.com

Rank	Site	Rank	Site
313	exdynsrv.com	339	pbs.org
314	bettycrocker.com	340	aliexpress.com
315	searchencrypt.com	341	syf.com
316	wired.com	342	autozone.com
317	pizzahut.com	343	cabelas.com
318	cargurus.com	344	decider.com
319	washingtontimes.com	345	playstation.com
320	starbucks.com	346	cbs.com
321	slideshare.net	347	godaddy.com
322	epicurious.com	348	flightaware.com
323	autotrader.com	349	kayak.com
324	bestdeals.today	350	blackboard.com
325	retailmenot.com	351	dillards.com
326	bodybuilding.com	352	nordstrom.com
327	telemundo.com	353	zappos.com
328	united.com	354	fortune.com
329	michaels.com	355	cars.com
330	liveleak.com	356	timeanddate.com
331	rottentomatoes.com	357	mashable.com
332	howstuffworks.com	358	spokeo.com
333	usaa.com	359	sprint.com
334	com	360	instructure.com
335	icbook.com	361	brit.co
336	mega.nz	362	emedicinehealth.com
337	ssa.gov	363	wikimedia.org
338	dailykos.com	364	grtyb.com

Rank	Site	Rank	Site
365	netfind.com	391	khanacademy.org
366	greatist.com	392	umblr.com
367	superuser.com	393	hgtv.com
368	chegg.com	394	wellhello.com
369	theonion.com	395	victoriassecret.com
370	ulta.com	396	draxe.com
371	thelist.com	397	cookingclassy.com
372	fidelity.com	398	hotels.com
373	redfin.com	399	gunbroker.com
374	spendwithpennies.com	400	mundohispanico.com
375	amctheatres.com	401	qz.com
376	shopify.com	402	bloomingdales.com
377	food52.com	403	mirror.co.uk
378	chewy.com	404	beenverified.com
379	scribd.com	405	theblaze.com
380	directv.com	406	splinternews.com
381	marriott.com	407	timeout.com
382	evite.com	408	nationalgeographic.com
383	popculture.com	409	victoriabrides.com
384	state.gov	410	liftable.com
385	synchronycredit.com	411	ny.gov
386	history.com	412	seattletimes.com
387	usbank.com	413	justanswer.com
388	zimbio.com	414	kym-cdn.com
389	kansascity.com	415	harborfreight.com
390	sharepoint.com	416	qvc.com

Rank	Site
417	dailysnark.com
418	thefreedictionary.com
419	education.com
420	nj.com
421	duckduckgo.com
422	workingmothertv.com
423	ask.fm
424	newyorker.com
425	pcgamer.com
426	lifebuzz.com

Android UI Widget Hierarchy


4:13

amazon

Add a credit or debit card

Enter your card information

Scan your card

or

Use name on account

Name on card

Card number

Expiration date

01

2018

Use this card across all of Amazon What's this

Add your card

Add a bank account

Appendix B Figure 1: Payment screen of the Amazon app.

Listing 1: Extract of the widget hierarchy presented by droidmate for the login screen in Appendix B Figure 1

```
<?xml version='1.0' encoding='UTF-8' standalone='yes' ?>
<hierarchy rotation="0">
<node index="0" text="" resource-id=""
    class="android.widget.FrameLayout"
    package="com.amazon.mShop.android.shopping" content-desc=""
        checkable="false" checked="false"
    clickable="false" enabled="true" focusable="false" focused="false"
        scrollable="false"
    long-clickable="false" password="false" selected="false"
        bounds="[0,0][1080,1794]">
<node index="0" text="" resource-id=""
    class="android.widget.LinearLayout"
    package="com.amazon.mShop.android.shopping" content-desc=""
        checkable="false" checked="false"
    clickable="false" enabled="true" focusable="false" focused="false"
        scrollable="false"
    long-clickable="false" password="false" selected="false"
        bounds="[0,0][1080,1794]">
<node index="0" text="" resource-id="android:id/content"
    class="android.widget.FrameLayout"
    package="com.amazon.mShop.android.shopping" content-desc=""
        checkable="false" checked="false"
    clickable="false" enabled="true" focusable="false" focused="false"
        scrollable="false"
    long-clickable="false" password="false" selected="false"
        bounds="[0,63][1080,1794]">
<node index="0" text=""
    resource-id="com.amazon.mShop.android.shopping:id/drawer_layout"
    class="android.support.v4.widget.DrawerLayout"
        package="com.amazon.mShop.android.shopping"
    content-desc="" checkable="false" checked="false" clickable="false"
        enabled="true"
    focusable="false" focused="false" scrollable="false"
        long-clickable="false" password="false"
    selected="false" bounds="[0,63][1080,1794]">
<node index="0" text=""
    resource-id="com.amazon.mShop.android.shopping:id/alx_intercept_layout"
    class="android.widget.RelativeLayout"
        package="com.amazon.mShop.android.shopping"
    content-desc="" checkable="false" checked="false" clickable="false"
        enabled="true"
    focusable="false" focused="false" scrollable="false"
        long-clickable="false" password="false"
    selected="false" bounds="[0,63][1080,1794]">
<node index="0" text="" resource-id=""
    class="android.widget.RelativeLayout"
    package="com.amazon.mShop.android.shopping" content-desc=""
        checkable="false" checked="false"
    clickable="false" enabled="true" focusable="false" focused="false"
        scrollable="false"
    long-clickable="false" password="false" selected="false"
        bounds="[0,63][1080,1794]">
...
</hierarchy>
```