Automatic Test Transfer across Applications

Andreas Rau
Saarland University
Saarbrücken, Germany
rau@st.cs.uni-saarland.de

Maximilian Reinert
Saarland University
Saarbrücken, Germany
mreinert@st.cs.uni-saarland.de

Andreas Zeller
Saarland University
Saarbrücken, Germany
zeller@cs.uni-saarland.de

CCS Concepts

•Software and its engineering → Software verification and validation; State systems; **•Information systems →** *Web applications; Document topic models;* Web crawling;

Keywords

Software Testing; Web Testing; Porting Tests

ABSTRACT

Building test suites for the Web is hard. We present a novel technique to automatically transfer and adapt existing Selenium test suites from one web application to another. By mapping functional states using topic analysis, we identify which actions yield the same result. This mapping allows for fully automatic test transfer—even across different applications in the same domain: we can take a shopping test that buys a product at Amazon and automatically adapt it to run on eBay or other eCommerce sites.

In an evaluation of test traces of 16 real world applications — encompassing more than 1200 user actions — ATTABOY was able to successfully transfer 49.3% of all test cases without any human intervention in domains such as eCommerce, knowledge bases, search engines, and news sites.

1. INTRODUCTION

What does it take to create a test suite for a web application? Developing a test case includes the creation of valid and invalid input values, identifying the correct target elements on a page and verifying that an executed action yields the correct result. The correct result is typically asserted by checking the presence of certain key elements in the web page, e.g. the presence of a button labeled with 'Proceed to Checkout'. A test suite combines these single process snippets into a "story", expressing for instance the use case scenario for buying a product. Figure 1 represents

ACM ISBN .

DOI:

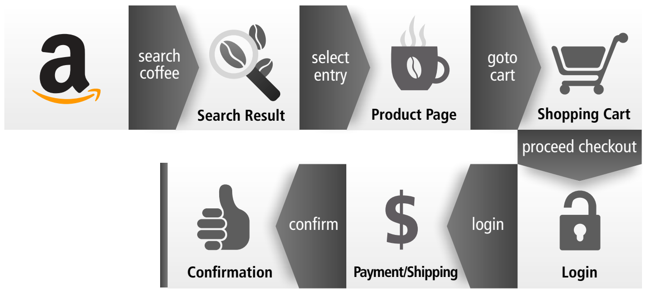

Figure 1: Test sequence for buying a product on https://www.amazon.com

a simplified view of a complete order process in Amazon, including the search for a product, over to a login procedure, passing the payment information until finally the "Purchase Complete" page pops up. The test has succeeded if the final page has been reached.

As humans, we can read a scenario like the one in Figure 1 and easily apply it again and again on Amazon. But we can also apply it on other eCommerce sites, checking whether purchasing a product works as intended. The individual steps on other sites may be slightly different, but we should always be able to start with a search and end with a confirmation of our purchase.

In this paper, we present a technique to take a scenario like the one in Figure 1 and *automatically transfer it to web sites in a similar domain,* which considerably reduces the effort for test development. Given an existing test suite for a source application A, ATTABOY crawls the target application B and generates a new test suite for B. The key idea is to leverage a set of *Natural Language Processing (NLP) techniques* to identify states and their context in the original application, and to *map* these states to similar states in the target application.

Our ATTABOY prototype executes the test suite for application A and records a *behavioral model* encompassing the applied actions and resulting states. Figure 2 shows such a mapping between two behavioral models. Once we know that two states are equivalent, we can transfer scenarios from one application to another by following similar paths from start to end. The resulting test suite for the target application B consists of the crawled actions, altogether with a new oracle telling what the supposed outcome of this action was in the source application.

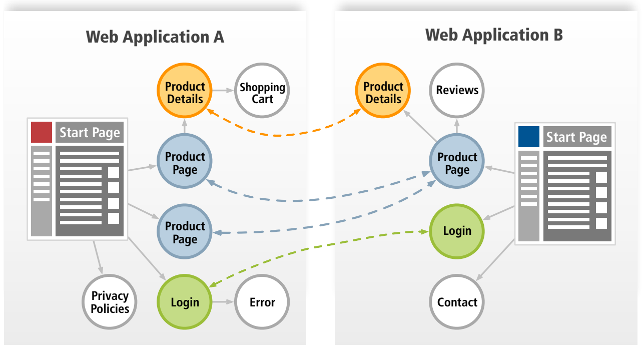

Figure 2: Simplified functional state overview on two eCommerce web applications. States with equal labels are clustered and represent the same functional state. Dashed arrows indicate a mapping to functional similar states. Nodes are labeled with their main topics.

After discussing related work (Section 2), this paper makes two central contributions:

Test Transfer Across Applications
A novel method for *transferring test suites across Web applications* (Section 3), as implemented in the ATTABOY prototype, building on novel techniques for visual element clustering (Section 3.1), noise reduction (Section 3.2), clustering functional states (Section 3.3), mapping states via natural language topic analysis (Section 3.4), and mapping across applications (Section 3.5).

Domain Specific Process Analysis
An evaluation (Section 4) on common process snippets across 16 representative web applications covering eCommerce systems, knowledge bases, news websites and search engines, both on manually written tests, as well as auto-generated ones. In our evaluation, ATTABOY was able to fully automatically transfer an average of 59% of manual tests, and 49.3% of automatically generated tests.

After discussing threats to validity (Section 5), we close with conclusion and future work (Section 6).

2. BACKGROUND

Modern web applications incorporate many features, are highly dynamic and interactive. Typically, they try to follow established usability standards to ease the effort for their users to familiarize themselves with the underlying functionality. In other words, it is of high interest to a service provider to avoid steep learning curves. Reusing standard usability patterns like login or address forms, shopping carts or confirmation pages allows users to deduce from experience on what they are supposed to do at any given moment. Changing element styles or reordering functional similar states can improve the user experience and may represent a unique feature giving an edge over competitors, but should not prevent the user from understanding what he is supposed to do.

Although the back-end code is hidden from the user, the web interface can still be mined for information and allows us to build a behavioral model, i.e. by specifying the single states a web application can be in and which actions are required to change this state. Literature offers a wide range of data mining and information retrieval techniques to extract the state of an application through the browser interface. Mesbah et al. [12] presented CRAWLJAX, a crawler, which creates a state model of an application by analyzing changes in the user interface. Dallmeier et al. [5] presented an alternative approach to leverage this information and introduced a methodology to identify cross-browser related issues, i.e. unwanted changes which occur if an application is executed on different browsers.

Choudhary et al. [15] map features, i.e. elements of a web page, in a cross platform setting, allowing to check the consistency of features throughout desktop and mobile platforms. Despite the hardware induced differences in mobile and desktop versions, both platforms are supposed to offer the same information and functionality, while possibly following different usability patterns. Their methodology allows to check both the presence and absence of certain features across platforms, but cannot guarantee, that the interaction with these elements produces the expected result. Furthermore, it is restricted to be executed within the same application. Structural differences and mismatches between the unique identifiers restrict the mapping capabilities across applications.

In modern development, automatically testing web applications on a system level has become increasingly important to meet the requirements of short release cycles. SELENIUM, an automation engine based on JavaScript, serves as an interface to a large set of test-browsers, translating commands to simulated user actions executed on the browser under test. The SELENIUM framework sets up a web server interface, controlled by simple HTTP commands in order to create a new browser session (i.e. a new browser instance is started and available as a web driver) and to send user commands (e.g. `GET` opens the given URL or `CLICK` clicks a previously identified target web element). As part of a test automation framework, a SELENIUM test suite can be used to check the presence of certain functionalities, control flows or likewise. In contrast to random crawlers [6, 12, 5], such a test suite is designed with domain specific knowledge and can test all user accessible functionality. Though random crawlers also test the behavior of an application, they lack this domain specific knowledge and thus usually less efficient and thorough. Furthermore, this kind of automatic testing technique has no knowledge on what the correct state of an application is supposed to be after an action has been executed.

Leveraging existing unit tests for further application analysis is not a new concept [13], but is limited to be executed on different states of the same application. This restriction is implied by the inability to identify how the current state of the target application is related to the source application. Traditional mapping techniques — based on structural element distribution or styling attributes — are insufficient. Therefore, transferring the actions of a source application to a target application cannot infer, if the produced result is actually correct.

The question is, how can a human decide on what he is supposed to do for instance to purchase a product? How does he know that the action was successful? Understanding that a product order requires to enter 'username' and 'password' is a previously learnt behavior. The application under test dictates where in the process the information has to be provided. The outcome of the interaction is determined by the input (valid username/password) and the process (leading for instance to an authorized state). For a human user, the surrounding text implies what action is required and the success/failure can be derived from the text in the next state as well. The actual wording is of minor importance, as long as the actual meaning is clear.

NLP techniques have proven useful to analyze the underlying topic model of texts, even allowing for a classification model, if a description matches the proposed functionality. Gorla et al. [7] even showed that the description of an application can help to decide which underlying functionality (in this case API method calls) are to be expected.

The combination of the presented ideas of information retrieval and automated test case generation – i.e. crawling – together with a novel model generation and mapping technique based on topic analysis, allows Attaboy to transfer given test suites across applications and thus create new test suites for applications within the same domain.

3. MODEL GENERATION

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 [5] 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 of a page is specified in an underlying Document Object Model (DOM) in which every element of the page is expressed as a node. 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 3 presents an overview on Attaboy's process pipeline.

Considering the initial example of buying a product on Amazon (Figure 1), 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 3.1), noise reduction (Section 3.2) and functional state clustering (Section 3.3). After this pre-processing phase, we extract the main topics describing the content and functionality of a page (Section 3.4). 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.

3.1 Visual Element Clustering

Algorithm 1: Visual Element Clustering - Recursive algorithm extracting blocks of a given DOM tree.

```
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
```

The DOM of a single page may consist of thousands or hundreds of thousands elements, i.e. nodes. Before we can identify common web elements or structures throughout the entire web application, we group individual web elements within a single state based on their visual coherence. These groups are then mapped throughout the web applications. Instead of comparing hundreds of thousands single elements we only have to match a few hundred groups. In addition, we also use the resulting segments in order to identify noise in the next step. Although the DOM already structures elements hierarchically, this structure may not follow the actual presentation within the browser, is hard to generalize and shows unsatisfactory results [16].

Literature offers a large amount of studies, which focus on webpage segmentation. Kohlschütter et al. [9] introduced a densitometry approach for page segmentation based on text density. The area of application is restricted to the purpose of information retrieval where a large amount of text is present. Cai et al. [4] present a vision-based page segmentation algorithm to extract the page structure based on its visual perception. Their technique depends on visual separators and a set of predefined heuristic rules. To generate the final vision-based structure, many iterations are required. Improved versions of the VIPS algorithm [1, 16] circumvent this limitation and represent the basic idea of our visual element clustering solution.

We first remove all invalid nodes, i.e. nodes with no visual representation like `STYLE`, `SCRIPT` or `META`, out of the DOM. After that—starting with `BODY` as the root node—the tree is traversed recursively. For each node, the algorithm (see Algorithm 1) checks, if it is in between predefined boundaries $isValid()$, i.e. that an element is within the viewport of the page and has no empty dimensions. Afterwards, the node properties are checked against a set of extended VIPS rules (see Table 1). 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. Furthermore, we allow setting the maximum number of children per block and a maximum DOM tree depth for the analysis.

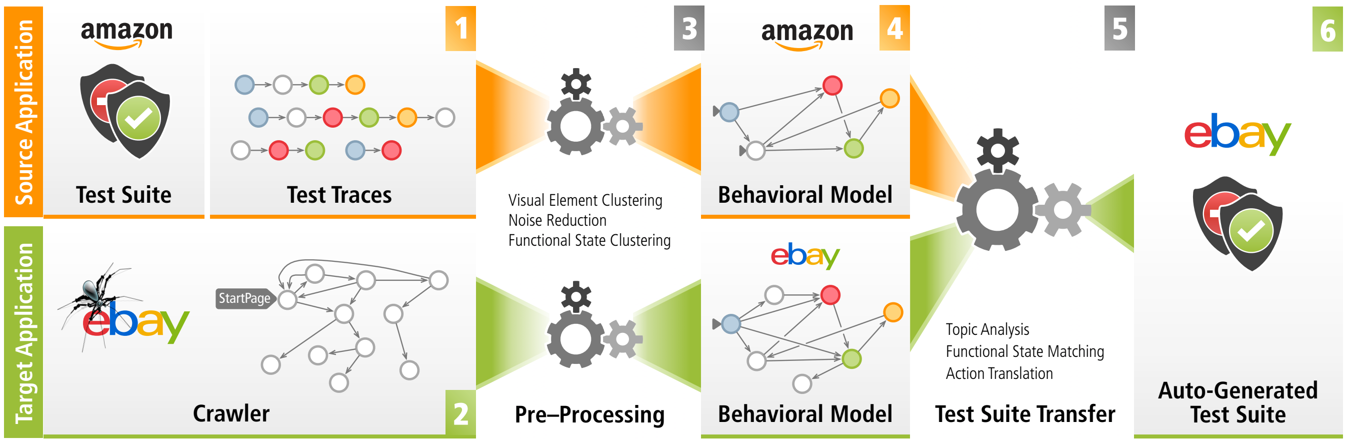

Figure 3: **Attaboy Process Pipeline** for automatic test suite generation. A given reference test suite is executed (1) and processed (3). Meanwhile a random test suite for the target application is created (2) and processed as well. Afterwards the resulting behavioral models (4) are transferred by 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).

Table 1: VIPS rules for element clustering. If the node property is fulfilled for a certain node, the listed action is executed. *'NB'* creates a new basic block. *'T'* - traverse recursively and analyze child nodes

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

3.2 Noise Reduction

With our page segmentation functionality in place, the next step is to filter out common blocks with redundant information throughout the complete state model of the application. Noise — in our model reoccurring structures, which offer no classification power over the underlying context or functionality of a page — is a common problem in the field of information retrieval and web data mining. Noise is typically introduced by default templates or advertisements. Service providers for instance have to add imprints to the footer (due to legal regulations) or include navigation elements to every single page of their application.

Kohlschütter et al. [8] presented a tool that automatically identifies templates and effectively removes content which is not related to the main content. As with their previously mentioned page segmentation algorithm, this approach is optimized for information retrieval, where a large amount of text is present. Yi et al. [17] instead build a so called Site Style Tree (SST), which basically is a merged DOM tree of a web application. The SST is built by adding the DOM for each page and joining it with the existing tree. For each new page, the page count is incremented for equal nodes. Noise is identified by checking the number of the nodes. The higher the frequency, the more likely it represents noise. The algorithm needs around 500 pages of a given web application pages as training data before it can effectively classify noise. Although we could train or model by randomly crawling the web application, we also want the possibility to generate or model on smaller traces produced by small SELENIUM test suite featuring less than fifty user actions.

Alassi et al. [2] introduced a VIPS based algorithm able to cope with a much smaller set of training data. We compute the noise value for all block-pairs of all states based on their *structural similarity*. The structural similarity is computed by traversing both blocks in depth-first manner and count equal nodes. Two nodes are considered equal if they have the same relative XPath. The normalized block similarity is computed by

$$S_{Block} = \frac{2 * M}{S_1 + S_2} \qquad (1)$$

where M is the number of equal nodes and S_n is the number of nodes in each block. The final noise value for a single block is the average of all block matchings .

The next step is to find a suitable cut-off value, to decide whether a block is considered noise. Instead of using a global cutoff value for the whole application, we compute a dynamic threshold (see Algorithm 2) that is dependent on the average and maximum noise value per state. Noise is typically not uniformly distributed throughout the application. Start pages typically contain navigational elements that are not present if the user is prompted for credentials on a separate login-page. The dynamic threshold allows for a more fine-grained analysis of the states, thus minimizing the effect of removing elements with valuable data for the later topic classification. Still, a domain specific *tuning value* t can be provided to handle domain specific difference of the noise distribution. Thus, t can be manually defined in the training phase of each reference model.

Finally, advertisements represent a significant source of

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

Algorithm 2: Dynamic Noise Threshold - Algorithm to compute a dynamic cutoff value for noise classification.

```
Input:
$s$, page of web app
$t = \{t \in \mathbb{N} | 0 < t \leq 100\}$, tune value
Result: $T$, noise threshold
begin
    Let $\mu \leftarrow s.getAvgNoise()$
    Let $m \leftarrow s.getMaxNoise()$
    T = (m/100) * t
    if $T < \mu$ then
        $T = ((m - \mu)/2) + \mu$
    return $T$
end
```

noise to modern web applications. They typically are not stable, meaning that based on the behavior of the user they change over time. Even a simple refresh of a page reloads the embedded advertisement. Some advertisement originates from the application itself, e.g. recommendations to other products or articles. It is part of the template and are covered by the dynamic threshold analysis. Cross-origin advertisements on the other hand, are hard to analyze. Embedded iframes pointing to external sources are therefore completely ignored in our state extraction and translated to a non-empty noise block with the dimensions of the iframe.

3.3 Functional State Clustering

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 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 reuse Equation 1 for S_{Block}, where M denotes the number of matching classes (the intersection) and S_n the respective size of a set. A configurable 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.

3.4 Natural Language Processing (NLP)

Using the pre-processed input data, we collect the text content of all non-noisy nodes within a state to determine

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

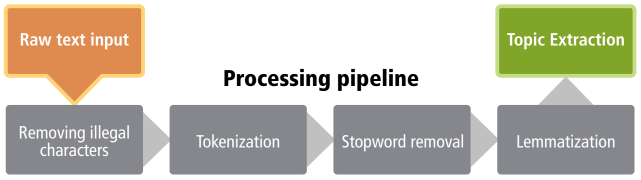

Figure 4: Topic extraction pipeline implemented in Mallet. Starting with the raw text input of a webpage, resulting in a list of topics

the overall topic of a given state. Standard NLP techniques allow to process the textual content and extract the main topics. Blei et al. [3] introduced the Latent Dirichlet allocation (LDA), a probabilistic topic model to annotate document archives with thematic information. Probabilistic topic model algorithms are able to discover topics in large and unstructured document collections without any prior annotation or labeling of the documents. LDA assumes that the order of words in a document is irrelevant. Thus, a *topic* is defined as a distribution over a fixed vocabulary.

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[3] framework [11], 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.

We implement a sequence of text processing steps as Mallet `pipes` as depicted in Figure 4. 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, we integrate a morphological stemming technique presented by Minnen et al. [14] called Morpha Stemmer[4] for lemmatization. Reducing the inflectional forms (e.g. the words 'speak', 'spoke' and 'speaking' are substitutes of the word 'speak') improves topic extraction and accelerates the semantic comparison. 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

[3]MAchine Learning for LanguagE Toolkit
[4]https://github.com/knowitall/morpha

'overfitting', we fall back to our previous functional state clustering. We cluster all states of a web application, collect the LDA topics for each state and return the intersection. The resulting topic set is an abstraction of all states within a cluster.

3.5 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. 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.

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 3 by applying the pre-processing methods presented earlier.

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 [10]. DISCO calculates the semantic text similarity of the topic sets. Figure 5 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 exactly 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.

4. EVALUATION

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[5]. 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 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 3.

[5]dmoz.org

Table 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

In a second phase (see Section 4.2) we extended the set of test subjects by yet another domain ('news papers'), but instead of manually creating test cases, we applied random crawling to automatically generate test traces through the applications of all four domains and compared them in terms of model similarity and the possibility to share basic test suites.

4.1 Transferring Test Suites

RQ1 Can a given SELENIUM test suite be transferred to the test suite of another application within the same domain?

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 6 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 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

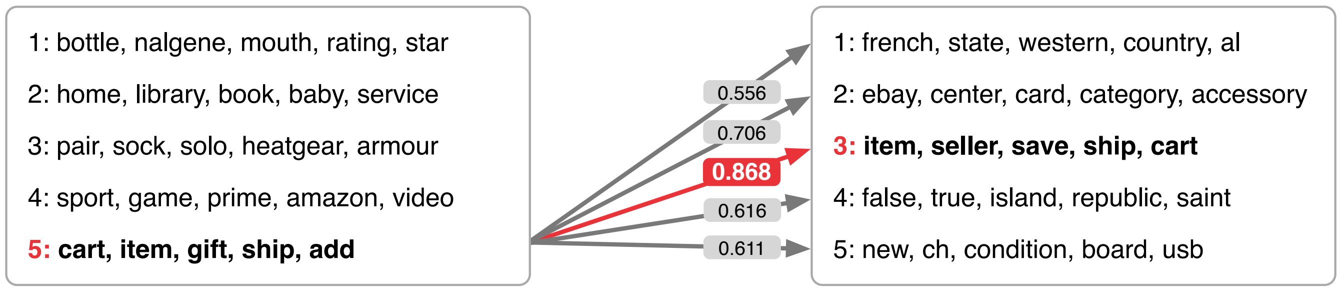
1: bottle, nalgene, mouth, rating, star
2: home, library, book, baby, service
3: pair, sock, solo, heatgear, armour
4: sport, game, prime, amazon, video
5: cart, item, gift, ship, add
0.556
0.706
0.868
0.616
0.611
1: french, state, western, country, al
2: ebay, center, card, category, accessory
3: item,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

Figure 5: 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

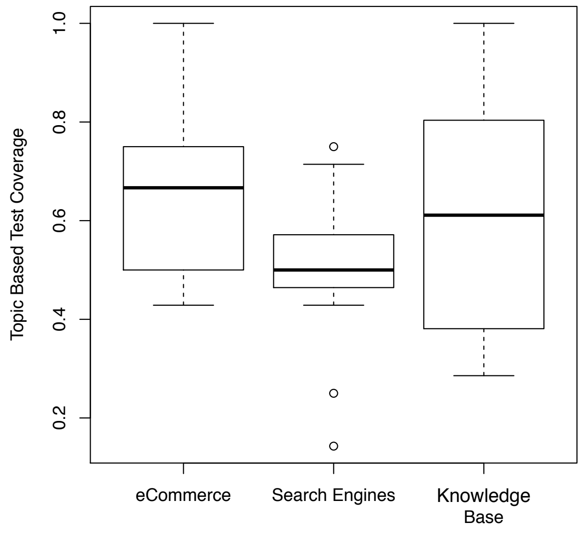

Figure 6: ATTABOY's results for transferring the manual test suites across eCommerce, search engine and knowledge base applications.

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.

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

4.2 Usage for Test Generation

RQ2 Can we transfer automatically generated test suite across applications?

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, RQ2 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 [12] or CRAWLER4J [6] 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 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 3 shows an overview on the generated application models created by the crawler.

Table 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 Website	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
	duckduckgo.com	—	—

A fourth category ('News Website') 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 7 shows the success on transferring the randomly generated test traces. 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

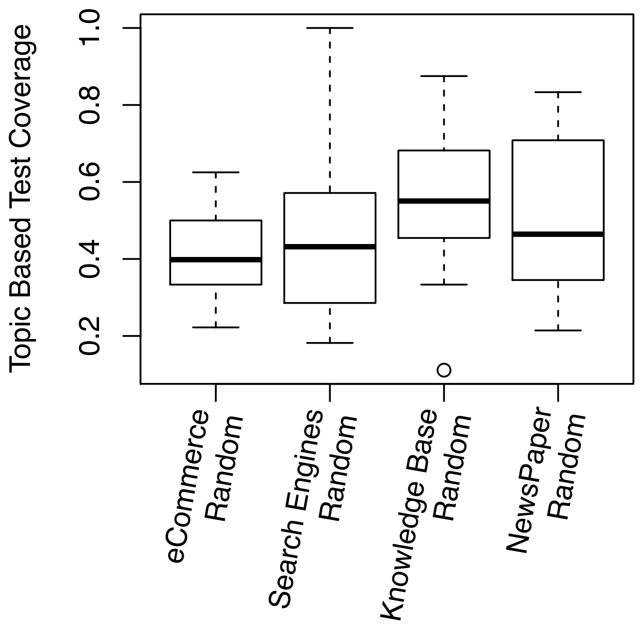

Figure 7: ATTABOY's results for transferring automatically generated test suites across domains

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.

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

4.3 Application Similarity

RQ3 How similar are web applications regarding their features and feedback mechanisms?

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 7 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 4.2 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.

5. THREATS TO VALIDITY

As any empirical study, this one faces threats to its validity:

Abstracting the application model
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.

Noise Reduction
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.

Generality of Selenium Test Suites
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.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we presented ATTABOY—a novel approach to transfer web application test suites across applications. ATTABOY maps functional states using topic analysis to identify which actions yield similar results, and leverages this mapping to transfer test suites. This approach is effective: On average, ATTABOY translates 59% of domain specific use cases (manual SELENIUM tests) and 49.3% of the overall behavior (random test case generation) across applications within the same domain. With this, ATTABOY lays a foundation for further research in mining, comparing and analyzing web applications.

Besides addressing general concerns such as robustness and ease of use as well as the specific issues raised in Section 5, our future work will be to continue to integrate NLP features to identify commonalities and discrepancies across multiple applications. This will open several interesting research opportunities:

Usability Analysis
The first and most noteworthy research area would be to learn common usability scenarios by exploring multiple applications within a domain. Finding outliers in the behavioral models would allow us to classify the usability of an application based on *learnability*, *memorability*, *performance* and *error proneness*.

Information Flow
Privacy has become a severe challenge. Knowing what information is required to access certain functionality within the application is straightforward. From a security perspective, it is interesting to learn if a user can reach a certain state without providing this information or if a competitor allows another methodology to reach the same goal.

Automated Test Case Generation
Learning domain specific knowledge is essential to actually apply the learned specifications across domains. Leveraging unit tests has already been done [13], but we would like to investigate if it is possible to actually extend existing test generators with knowledge transferred from other applications in the domain. Random crawling techniques face the problem to generate correct input for domain specific fields. Transferring this knowledge might allow us a better coverage of the application's behavior.

To learn more about ATTABOY and to access all test suites and data described in this paper, visit our project page

7. REFERENCES

[1] M. E. Akpinar and Y. 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.

[2] D. Alassi and R. Alhajj. Effectiveness of template detection on noise reduction and websites summarization. *Inf. Sci.*, 219:41–72, Jan. 2013.

[3] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent Dirichlet Allocation. *Journal of Machine Learning Research*, 3(4-5):993–1022, 2012.

[4] D. Cai, S. Yu, J. R. Wen, and W. Y. Ma. VIPS: a visionbased page segmentation algorithm. *Beijing Miciosoft Research Asia*, pages 1–29, 2003.

[5] V. Dallmeier, M. Burger, T. Orth, and A. Zeller. WebMate: Generating test cases for web 2.0. *Software Quality. Increasing . . .* , 2013.

[6] Y. Ganjisaffar. Crawler4j website. https://github.com/yasserg/crawler4j, March 2016.

[7] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. Checking app behavior against app descriptions. In *ICSE '14: Proceedings of the 2014 International Conference on Software Engineering*, pages 292–302. ACM Press, June 2014.

[8] C. Kohlschütter, P. Fankhauser, and W. Nejdl. Boilerplate Detection using Shallow Text Features. *Text*, pages 441–450, 2010.

[9] C. Kohlschütter and W. Nejdl. A densitometric approach to web page segmentation. *Proceedings of the 17th ACM conference on Information and knowledge management*, pages 1173–1182, 2008.

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

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

[12] A. Mesbah, A. van Deursen, and S. 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.

[13] A. Milani Fard, M. Mirzaaghaei, and A. Mesbah. Leveraging existing tests in automated test generation for web applications. *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE '14*, pages 67–78, 2014.

[14] G. Minnen, J. Carroll, and D. Pearce. Applied morphological processing of English. *Natural Language Engineering*, 7(03):207–223, 2001.

[15] S. Roy Choudhary, M. R. Prasad, and A. Orso. Cross-platform feature matching for web applications. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 82–92, New York, NY, USA, 2014. ACM.

[16] C. S. Win. Web Page Segmentation and Informative Content Extraction for Effective Information Retrieval. *International Journal of Computer & Communication Engineering Research (IJCCER)*, 2(2), 2014.

[17] L. Yi, B. Liu, and X. 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*, page 296, 2003.