EXPLORING TEXT-BASED ANALYSIS OF TEST-CASE DEPENDENCIES OF WEB APPLICATIONS

by

Camille Cobb

2012

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Web applications must be reliable as the number and popularity of web applications increases. Web applications are difficult to test because of the large input space and frequent changes. Thus, their characteristics demand an effective way of automating the test case generation process. Web application test cases often depend on what happened to the shared, persistent application state in previous test cases—I call this an inter-test-case dependency, or simply a dependency. Current test suite generation processes do not represent dependencies, and generated tests suites often violate dependencies, which negatively impacts the effectiveness of the test suite. This thesis explores the feasibility of computing dependencies from an application's resources. I propose a novel text-based approach to analyzing resources based on the insight that resources contain embedded context since they were written by human developers. In a feasibility study of five deployed web applications, I correctly identify several dependencies and show the promise of a text-based approach. I propose a process for augmenting the test case generation process to produce test suites that better uphold estimated dependencies. I identify several avenues of future work, including suggestions for improvements to the text-based methodology for estimating dependencies to improve the accuracy of dependency estimates and implementation of the proposed augmented test case generation process.

Chapter 1

INTRODUCTION

Web applications are applications that are accessed over a network, generally through a web browser on the Internet, to accomplish some task. Unlike static web sites, web applications dynamically generate many of the pages that users see based on various factors such as the user's session history or the contents of a common database. Since web applications are accessed in a web browser, they are compatible with a variety of operating systems and can be updated and maintained without requiring users to acquire or install new software. Well-known web applications include search engines like Google, online tutorial applications like Sakai and WebAssign, financial services like TurboTax and online banking, and online stores like eBay and Amazon.

Web applications are becoming increasingly common, and people are becoming more and more dependent on these applications to accomplish tasks such as managing money and buying goods. It is therefore imperative that web applications work properly and consistently, which means they must be thoroughly tested; however, testing web applications is difficult and expensive because of their large input space and frequent changes.

One approach to making the testing of web applications cost-effective and easier is to automate the testing process. Although this is promising, current automated testing methods are not efficient or accurate enough. The goal of my thesis is to improve an existing automated test case generation process by modeling dependencies between test cases.

The contributions of my work are:

1. the identification of the inter-test-case dependency problem

2. a methodology for finding dependencies based on a text-based approach to analyzing an application's resources
3. a feasibility study that shows the promise of a text-based approach to estimating dependencies
4. an augmented test case generation process to produce test suites that uphold inter-test-case dependencies.

The remainder of this thesis is organized as follows: in Chapter 2, I convey background information about web applications, the important factors in generating good test cases for web applications, previous approaches to generating test cases for web applications, and the limitations of the current state-of-the-art. Chapter 3 details the test-case dependency problem, which my thesis aims to improve. I propose a text-based approach to estimating test-case dependencies in Chapter 4 and discuss the methodology I have developed to automatically find dependencies. I designed and implemented a feasibility study, summarized in Chapter 5, which shows the promise of the text-based approach. In Chapter 6, I propose an augmented test case generation process to produce test suites that adhere to inter-test-case dependencies. The conclusions of this work and suggested areas of future work are discussed in Chapter 7.

Chapter 2

BACKGROUND

In this chapter, I expound on what constitutes a web application. I explain what makes web applications particularly difficult to test and the goals of a set of test cases for web applications. I discuss previous approaches to testing web applications and the limitations of these approaches.

2.1 Web Applications

A web application is a set of web pages and components that form a system in which user input (navigation and data input) affects the system's state. Users interact with a web application in a browser, making requests over a network using HTTP, as shown in Figure 2.1. When a user's browser transmits an HTTP request to a web application server, the application produces an appropriate response, typically an HTML document that the browser displays. A response can be either static, in which case the content is the same for all users, or dynamic such that its content depends on a user's behavior and input from the current session (session state) or the contents of the application's shared data store (application state).

2.2 Challenges and Goals of Testing Web Applications

Web applications present unique testing challenges. Often, frequent changes are made to web application code to correct errors and update the functionality of the application. These changes must be made without making the application unavailable for any significant amount of time because users expect the application to be available at all times. Additionally, web application components (the browser, server, and data store) are often geographically distributed, and errors can occur in any of these

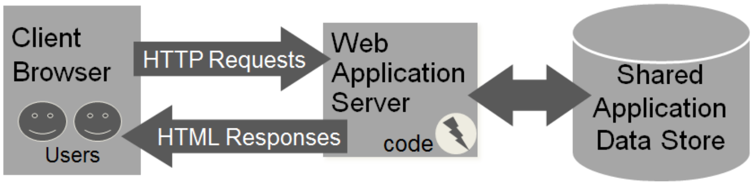


Figure 2.1: Typical structure of a web application.

components, which makes testing more difficult. Finally, web applications have a large input space. That is, there is an especially large set of user behaviors that impact the execution of the web application since responses are often generated dynamically based on the session state and application state. Since it is impractical to test every possible scenario of use, it becomes important to concentrate on testing behavior that is typical of real users. These properties of web applications make traditional software testing methods difficult, expensive, ineffective, and infeasible when applied to web applications. Thus, effective, automated, and cost effective testing solutions are needed for web applications.

The process of testing the functionality of a web application involves generating effective test cases, executing the generated test cases, and using oracles to the compare expected and actual results of the executed test cases. My work focuses on automatically generating effective test cases.

Measuring the effectiveness of a test case is nontrivial. The goal of testing a web application may be to determine how the application responds to heavy loads, ensure that there are no security vulnerabilities, or expose faults in the web application code. My research concentrates on testing the functionality of a web application, rather than testing security or excessive loads. Measuring fault detection requires faults to be seeded into the application code. Unfortunately, this introduces problems such as faults that are unrealistic or biased toward certain types of test cases. Code coverage, a measurement of how much of the application's source code will be executed when a test suite is executed, is a widely accepted and much simpler metric of a test suite's effectiveness.

2.3 Generating Test Cases for Web Applications

Previous research has examined several approaches to generating test cases for web applications including static models, concolic testing, navigation models, user-session-based testing, and statistical usage-based models.

2.3.1 Static Models

A static model seeks to model the structure of a system, which is generally achieved by examining application code. Static models have been successful for creating test cases for many kinds of software and were also effective for testing some of the first web applications, which had fewer dynamic features compared to newer web applications. Dynamic features of modern web applications, particularly the dynamically generated pages and non-traditional control flow, make it difficult to create effective test cases based on a static model of a web application [1, 5, 11, 12]. Approaches based on modeling web applications with finite state machines (FSMs) and using coverage criteria based on FSM test sequences are not meant to represent invalid inputs and suffer from the state explosion problem, which has been partially addressed by constraining inputs [2].

2.3.2 Concolic Testing

Several groups propose applying concolic testing to web application testing to generate white-box-based test cases with the goal of achieving branch or bounded path coverage [3, 19]. Concrete and symbolic execution and constraint solving are combined to automatically and iteratively create new input values to explore additional control flow paths through a PHP script. While this approach achieves good code coverage, it is not necessarily an accurate representation of real users.

2.3.3 Navigation Models

Application navigation is a challenge unique to web applications. Unlike traditional GUI-based applications, users can circumvent the application's desired navigation constraints by utilizing the browser's features, such as the back button, location bar, bookmarks, or multiple windows. Tonella and Ricca [17] found that 47% of user sessions included an infeasible navigation, which is a navigation that does not follow any edge in their extracted model of the web application, presumably caused by browser-based navigation. Proper enforcement of navigation constraints is important and should be tested in addition to legal navigation paths through the application.

Some researchers proposed building a navigation model of the application [10, 17, 13, 18] for use in testing and application understanding, among other tasks. Wang et al. proposed essentially spidering the application from a defined start page and using a combinatorial approach to input values into the application's forms to generate the navigation model [18]. Tonella and Ricca's navigation is similarly generated by spidering the application from a start page and inputting values from equivalence classes into forms [17]. They augmented their navigation model with usage information, adding usage-based probabilities to the edges.

Neither Wang et al.'s nor Tonella and Ricca's navigation model generation is completely automated. Both require the values to be input into forms to be known beforehand. Another limitation of both approaches is that neither seems to explicitly handle navigation that may depend on different application state (e.g., if a search fails to find any matches because of the contents of the database).

2.3.4 User-Session-Based Testing

A promising approach that is more representative of real user behavior is user-session-based testing. User-session-based testing records the actual user accesses to older versions of the application and parses them into user sessions, which are then used as test cases. Each user session is a sequence of user requests in the form of resources and parameter name-value pairs, as in the example request shown in Figure 2.2. Often,

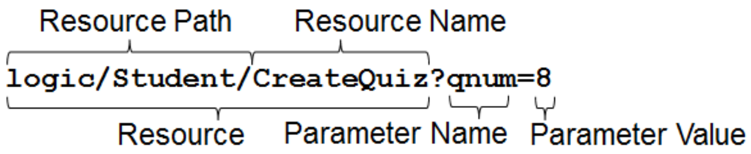


Figure 2.2: Example of a typical request.

the resource contains path information. I refer to the non-path part of the resource as the resource name. The request recorder treats hidden parameters the same as regular parameters. We say a user session begins when a request from a new Internet Protocol (IP) address arrives at the server and ends when the user leaves the web site or the session times out, after a 30 minute gap between two requests from a user [15].

While user-session based-testing is inexpensive and creates test cases that are representative of actual users, it generates too many test cases, many of which are redundant. Additionally, direct replay of collected user accesses is limited to user behavior, which might not produce good code coverage.

2.3.5 Statistical Usage-Based Models

A statistical model is based on the frequencies of the users' usage patterns, as recorded in the user sessions, which are inexpensive and simple to obtain. Usage information is important to model because user behavior does not follow exactly a statistically determined model and, when given several options, users tend not to choose between the options equally [16].

Sant et al. [13] proposed generating test cases using a model of user sessions that requires less space than the original user sessions. The model has two parts: a navigation model (called a control model by Sant et al.) that represents a user's navigation through a web application and a data model that represents the parameter values associated with these requests.

Sprenkle et al. [16] proposed modularizing navigation models and data models to allow testers to choose navigation and data models that are appropriate to the specific application being tested. Tuning the navigation model's configuration—including how requests are represented and the amount of history used—affects the resulting navigation model and generated abstract test cases [16]. A tester can generate many test cases from one abstract test case by adding parameter values generated from different data models to the abstract test cases [13]. Alternatively to using the whole set of user sessions to generate the statistical usage-based models, the original set of user sessions can be partitioned based on user privilege and used separately as input to the test case generation process to generate test cases specific to certain types of users as described in previous work with coauthors [14].

These approaches have advanced the state-of-the-art of testing web applications, but still present significant limitations. In the remainder of this thesis, I focus on one limitation, inter-test-case dependencies, and my approach to addressing this limitation.

Chapter 3

FOCUS: TEST CASE DEPENDENCIES

One property of web application test cases is that their execution is often affected by what has happened to the shared, persistent application state in a previous test case. Consider the example of an online tutorial application shown in Figure 3.1. Professors must create quizzes before students can take them. If a student attempts to access a quiz before it has been created, then the student will be unable to access the quiz and the application will correctly execute error code rather than application code. In this case, errors in the source code for accessing the quiz could not be detected, and the test suite has diminished code coverage. We call this an *inter-test-case dependency*, or simply a *test-case dependency.*

Currently, navigation models represent the *intra*-test case dependencies but fail to represent *inter*-test case dependencies. That is, the current navigation models account for the history within a single user session, which ensures that dependencies that come from session state are upheld. For example, a user must log in before accessing restricted pages. The session state contains a token representing that a user has logged in which is checked when a user attempts to access a restricted page. Since the current navigation and data models concentrate on a user's navigation of the application independent of other users [10, 17, 13, 18], they are unable to account for dependencies that come from application state—dependencies that are based on the contents of a shared data store or database. Dependencies between test cases within the generated test suite that are not upheld negatively impact the ability of the test suite to cover code and expose faults. When a dependency is not upheld, the application should execute error code. While the error case should be tested, it does not need to be tested repeatedly. Test cases executed in an order that upholds inter-test case dependencies

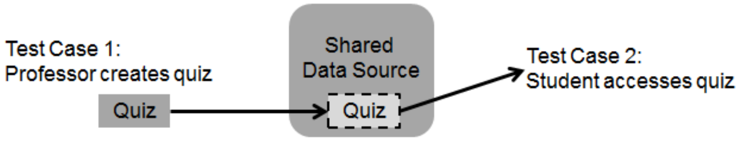


Figure 3.1: Example of a Dependency.

execute application code that represents valid, although not necessarily correct, functionality and likely increases the amount of code covered. In previous work evaluating automatically generated test suites, our lab group observed that this was a significant problem that prevented effective test suites in terms of code coverage.

Thus, the goal of my thesis project is to estimate inter-test-case dependencies automatically and adapt the test case generation process to uphold dependencies.

3.1 Problem

Inter-test-case dependencies result from changes to the shared data store (application state). Since these changes are caused by clients' requests to the server, an inter-test-case dependency is actually a dependency between requests, which make up a test case. These requests can be in the same or different test cases. Dependencies between two requests in the same test case can be represented by current navigation models regardless of whether the request impacts session state or application state. I focus on finding dependencies between requests in different test cases.

A request can result in data being written, read, edited, or deleted (killed) in the shared data store or could be independent of the application state. Although more sophisticated relationships between writes, reads, edits, and kills could be explored more extensively in future work, I estimate dependence based on the intuitive assumption about dependencies that (a) data cannot be read, edited, or killed until it has been written to the application state and (b) once it has been deleted (killed) from the application state, data can no longer be read or edited. Figure 3.2 represents this dependence relationship. Although an edit or kill could also involve a read to the shared

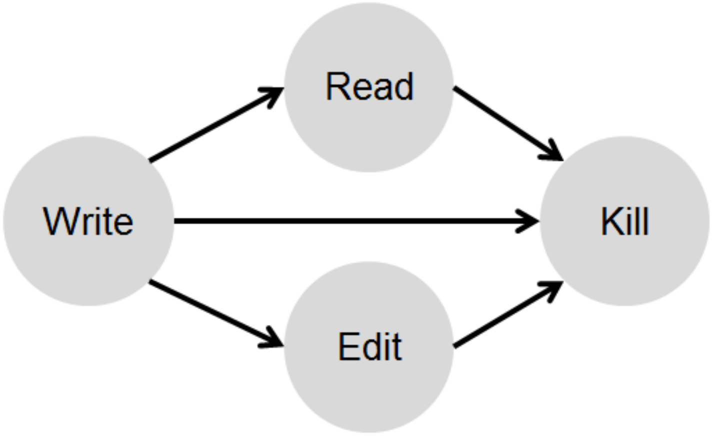


Figure 3.2: Dependence relationship between writes, reads, edits, and kills to the same data in shared application state.

data store, this does not impact the dependence relationship and is not represented separately.

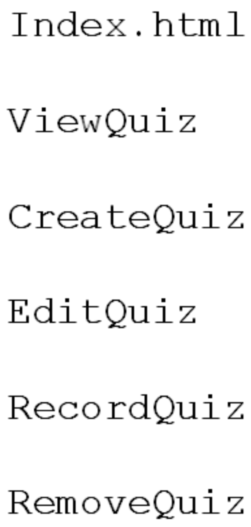
Manually determining all dependencies is tedious and error prone even for someone who is very familiar with the application. Thus, an automated approach to estimating dependencies is necessary. In order to automatically determine dependencies between requests, we must know two things: (1) if a request results in a write, read, edit, or kill and (2) what specific portion of the data store is being affected by the request. With this information, we can augment an automated test case generation process to uphold dependencies in generated test suites.

Chapter 4

TEXT-BASED APPROACH TO ESTIMATING DEPENDENCIES

My hypothesis is that a text-based approach to analyzing test cases can be used to estimate test-case dependencies, which can be incorporated into a data model to generate test cases that are more representative of actual usage and better adhere to inter-test-case dependencies.

I propose a novel text-based approach to estimating test-case dependencies. My key insight is that *URL requests contain embedded content, which may be leveraged to identify dependencies between test cases without requiring source code and at a low cost*. *Resources* are likely to contain useful information since they are written by a human web application developer so that the source code is easier to develop and easier to understand later. For example, a human can easily make assumptions about dependencies between the set of resources shown in Figure 4.1: `ViewQuiz` is dependent on `CreateQuiz` because a human could infer that `CreateQuiz` writes a quiz to the shared data store and `ViewQuiz` reads the quiz from the shared data store. This example illustrates the intuition behind my approach. The text content of resource names often reveals the developer's intent. In the previous example, the verb "create" suggests a write to the application state, and the direct object "quiz" establishes what is being written to the application state. I call words, generally verbs, that suggest whether a request is a write, read, edit, or kill *action words*. Direct objects in the resource name reveal which portion of the shared data store is being affected. A shared direct object among a set of requests suggests a dependency among those requests. Although resource names often have intuitive meaning to humans, it is difficult for a computer to automatically detect this meaning and assume dependencies.



```
Index.html

ViewQuiz

CreateQuiz

EditQuiz

RecordQuiz

RemoveQuiz
```

Figure 4.1: A human could easily assume dependencies between these resources.

4.1 Observation 1: Static Extensions Imply Independence

Many resource names end in static extensions like `html`, `ico`, `js`, and `pdf`. This suggests that the request results in a read to a static file and will not change or be affected by the application state. With very high confidence, we can assume that requests whose resource names end in a static extension are *independent* of any other requests. Once a request is known to be independent, it can be ignored for the rest of the dependency estimation process.

4.2 Observation 2: Action Words Suggest a Write, Read, Edit, or Kill to the Application State

Action words within a resource name suggest whether the request results in a write, read, edit, or kill to the application state. Because of the ambiguity of words' meanings and the limited context from resources alone, I propose manually identifying action words that can be classified as write, read, edit, or kill with high confidence as a first step and consider the challenges to automating this step later.

In a preliminary study (see Chapter 5), I found a relatively small set of action words, which was essentially universal across a broad range of applications. The classification of action words is the only manual aspect of my proposed methodology for estimating dependencies, and since action words are universal, a tester would generally not need to perform this manual step. Although incorrect classification of action words could severely impact the effectiveness of this approach, a tester could easily configure the set of classified action words to fit a specific application—presumably with only basic knowledge of the application's source code or domain (ex. banking, tutorial, etc.). I expect that manually configuring the set of action words would also enable testers to estimate dependencies for applications written in languages other than English.

4.3 Observation 3: Direct Objects Reveal the Portion of the Data Store Being Affected

Direct objects within a resource name reveal which portion of the shared data store a request affects. As with action words, current software tools are insufficient in determining which words are direct objects. Unlike the action words, direct objects are application-specific, which necessitates an automated approach to finding direct objects. Direct objects within a resource name are only useful once we know how the request is changing the application state, so we look for direct objects in resources that have been classified as write, read, edit, or kill. As a first approach, I assume that every non-action word in a resource is a direct object—I refer to this set of words as "potential direct objects." A word could be assumed to be a direct object based on its part of speech or the word ordering within the resource, but this is difficult to determine and not necessarily unambiguous. My approach results in the most inclusive set of direct objects.

Not every potential direct object suggests which portion of the application state is being accessed. For example, words like "to" and "in" may be common among several resources without suggesting a dependency. Linguists refer to these unimportant words as "stop words" and have established widely accepted lists of the most common stop words. We do not consider a stop word to be a direct object. In computer science and, specifically, in the web application domain, there are several additional stop words. For example, a servlet is a commonly used Java class that receives client requests and generates appropriate responses, so the word “servlet” has no significant meaning in a web application context. Within a given application, the name of the application is also treated as a stop word. The application’s name would presumably not be revelatory of which part of the data store is being accessed since it refers to the application itself. Finally, the meaning of words in the resource path is different from the meaning of words in the resource name, so we do not consider words in the resource path direct objects. When a potential direct object is found in only one unique resource name, it may reveal what portion of the data store is being affected, but since no other resource names suggest access or changes to the same portion of the shared data store, it cannot suggest a dependency and is not considered a direct object.

As with the set of action words, a tester could configure the stop words or even the final set of direct objects to be application-specific by specifying stop words.

4.4 Key Insight: Action Words and Direct Objects Imply Dependencies

Given a set of resources classified as write, read, edit, or kill and a set of direct objects, we can create groupings of resources that access or change the same portion of the shared data store and assume that the dependency pattern from Figure 3.2 holds within each grouping of resources that have a direct object in common.

Optionally, if a direct object is found in a resource name that does not contain one of the manually classified action words, a tester could manually examine the resource to determine whether a dependency estimate could be made within the context of the request. An area of future work is to explore the possibility of iteratively determining the meaning of unclear action words and finding more direct objects to make additional dependency estimates.

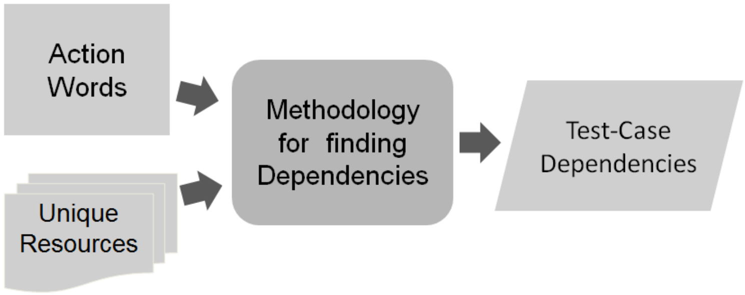


Figure 4.2: Methodology for estimating dependencies.

4.5 Methodology for Determining Dependencies

I have developed a methodology for estimating dependencies that leverages the above insights. The set of unique resources for an application and the manually identified and classified action words are taken as input, and a set of test-case dependencies are output as shown in Figure 4.2. The intuition behind this methodology is to handle the resources for which we have the most confidence in the classification first.

1. Classify resources with known static file extensions as independent.
2. Identify resources that contain one of the manually identified and classified action words.
3. Identify direct objects within the resources from step 2.
4. Assume dependencies between resources from step 2 with shared direct objects from step 3.
5. Optionally, use the direct objects from step 3 to make more assumptions about dependencies.

For the set of resources in Figure 4.1 and a set of action words that includes view (read), create (write), edit, and remove (kill), this methodology will estimate the same dependencies that a human would assume, as shown in Figure 4.3. The circles in this figure represent nodes in the dependency graph for these resources. In step 1 of the methodology, the resource `index.html` would be classified as independent since it ends in the static extension `html`. Next, in step 2, the resources `ViewQuiz`,

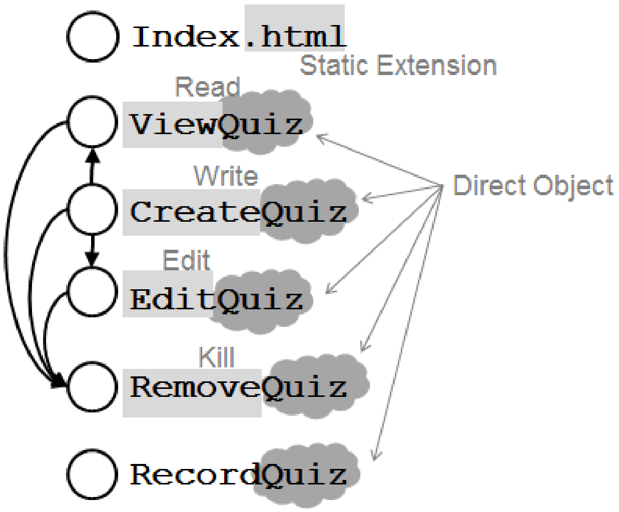


Figure 4.3: A set of resource names with action words, static extensions, and direct objects found.

`CreateQuiz`, `EditQuiz`, and `RemoveQuiz` would be classified as indicating a read, write, edit, and kill, respectively. In this case, `Quiz` is the only non-action word in these resources and would be identified as a direct object in step 3, indicating that these resources access or change the portion of the application state representing a quiz. The dependencies indicated by the arrows in Figure 4.3 would be assumed in step 4 of the methodology and follow the dependence relationship proposed in Figure 3.2. The resource `RecordQuiz` in this example also contains the direct object `Quiz`. In step 5, a tester could manually determine that the context of the word "record" implies that it is an action word indicating a write to the application state, thereby assuming an additional dependency.

4.6 Alternative Approach: Monitoring Data Accesses

An alternative approach to estimating test-case dependencies involves instrumenting application code to monitor data accesses or changes and executing test cases to determine the type of access or change and the portion of the data store that each request maps to. This approach could be explored in future work, but would likely have many of the same limitations as the text-based approach. For example, a dependency is likely to be violated in any test suite executed for this purpose (since the dependencies are still unknown). If a request violates a dependency, the usual access or change to the data store could not occur, and a single resource would be mapped to different accesses or changes to the data store. Determining which of the mappings represent correct functionality and therefore imply a dependency would be difficult.

Static analysis of the source code could be performed to identify the parts of the code that result in accesses to the database, and an application's resources could be mapped to its source code to estimate dependencies; however, static analysis of the source code is costly, and mapping resources to source code is not straightforward. This method would likely result in overestimates of inter-test-case dependencies, which would substantially limit the amount of user behavior that could be represented in test suites without violating dependencies and could result in diminished code coverage.

Chapter 5

FEASIBILITY STUDY, RESULTS AND OBSERVATIONS

I designed an empirical study to determine the feasibility of the text-based approach. The goal was to answer these research questions:

1. Can we automatically parse resources into words? The answer to this question tells us if a text-based approach to estimating dependencies between resources is possible.

2. Can we manually identify action words in resources that indicating a write, read, edit, or kill? If so, is the set of action words a reasonable size—large enough to have breadth in a variety of applications but manageable to manually find and reasonable to store?

3. How common are action words in resources across a wide range of applications? If they are universal, then manually identifying action words is not inhibitive to the proposed methodology.

4. How many resources can be automatically classified as independent or as indicating a write, read, edit, or kill using action words and static extensions? How often are those resources used in the collected user sessions? If action words are found in common resources, the dependencies found with the text-based approach are likely to handle dependencies between the most common requests.

5. Can we automatically identify direct objects in resources?

6. Are direct objects universal across a wide range of applications? If direct objects are universal, we can use a list of direct objects rather than going through the process of programmatically identifying direct objects.

7. Can we correctly assume dependencies based on classified action words and direct objects? If not, what additional information is required to assume dependencies?

Subject	# of Classes	NCLOC	# of Unique Resources
Masplas	9	609	20
Book	11	5279	29
CPM	76	7430	83
Logic	106	10704	90
Logicv2	135	16491	120
DSpace	291	29430	215

Table 5.1: Subject application characteristics

5.1 Subjects

I studied five publicly deployed web applications on servers administered by Sprenkle et al.'s research group [16, 14]. The applications are written in Java using servlets and JSPs and consist of a backend data store, a Web server, and a client browser. Since my testing techniques are language-independent—requiring resources but not source code for testing, these techniques can easily be extended to other web technologies.

I used 9 subject user-session sets from user requests to the applications, collected by Sprenkle et al. [16, 14]. The applications were of varying sizes (1K-50K non-commented lines of code), technologies, and representative of web application activities and usages: an e-commerce bookstore (Book) [8]; a course project manager (CPM) used as part of computer science courses at the University of Delaware; an online symbolic logic tutorial (Logic and a significantly revised version, Logicv2) used as part of philosophy courses at Washington and Lee University; and a customized digital library (DSpace) used by our research group to make our publications easily searchable and accessible [6]. Table 5.1 summarizes the applications' code characteristics including the number of unique resources in each application.

Book was the only application for which an email was sent to local newsgroups asking for volunteer users. These user requests were also used by Sant et al. [13]. For the remaining applications, users accessed the applications naturally, i.e., they were not solicited for experiments. Accesses for each application were collected over a long period of time: CPM: 5 academic semesters, Logic: 2 academic semesters, DSpace: 3.5 years.

Subject	# User Sessions	# Requests	% Lines Cvd
Masplas	169	1107	89%
Book	125	3564	61%
CPM	890	12352	78%
Logic	497	16,179	80%
Logicv2	374	16,052	78%
DSpace1	1087	12,277	74%
DSpace2	5012	14,110	46%
DSpace3	3853	15,126	45%
DSpace4	7687	38,155	49%
Total	19,525	127,827	–

Table 5.2: User session set characteristics

I had access to user sessions, that is the original user accesses converted into user sessions using Sprenkle et al.'s framework [15], useful in answering question 4. Before processing user accesses, accesses from IP addresses that are known to be spiders, bots, or malicious were removed to reduce the noise from non-users and better create models of human users' navigations. The DSpace user sessions are partitioned by the time periods in which they were collected to provide more sets of user session subjects to model and compare.

Table 5.2 shows the characteristics of the collected user sessions, in terms of the number of user sessions (totaling over 19,000 sessions), the number of user requests (totaling nearly 128K), and the percent of application code covered by the user sessions using Cobertura [4]. I report line coverage to show that the user sessions cover a large portion—but not all—of the application.

5.2 Experiment: Methodology and Results

5.2.1 Parsing Unique Resources

To answer **question 1**, I created a script to automatically split the unique resources into words by splitting at camel cased transitions and on non-alphanumeric characters (ex. `/`, `_` and `?`). These scripts took on the order of a second to execute for each application.

In our applications, only 15 of the 557 total unique resources contained words that arguably should have been split and were not. Fourteen of those contained compound words that represented a single concept—ex. "mydspace" and "callpapers." Only one unique resource—`masplas05/submitfile.jsp`—contained a word (submitfile) that definitely should have been split but was not. No resources were incorrectly split when they should not have been.

For the purposes of this study, I manually corrected the "submitfile" parsing error, but tools such as AMAP [9] could be used to automatically expand abbreviations, and Samurai [7] could be used to automatically split words that were not split with camel casing and non-alphanumeric characters if necessary.

Since Camel casing and punctuation correctly split words in all but a few resources, this fundamental part of the text-based approach can be implemented at a very low cost, answering question 1.

5.2.2 Manually Identifying Action Words

The first part of my methodology for estimating test-case dependencies is manually identifying and classifying action words as write, read, edit, or kill. My goal is to find a set of action words that can be classified with high confidence in any context.

To answer **question 2**, I began by looking at the parsed resource names for all of the applications except Logicv2, which was added to the set of applications later. I noted words that seemed to have an obvious classification; however, many of these words' classifications no longer seemed obvious without the context of the resource. For example, the word `format` suggests an edit in the context of the resource `dspace/dspace-admin/format-registry` but could also be used as a noun. To view the words out of context, I wrote a script that generated the list of unique words in each application. Even without context information, identifying and classifying words as write, read, edit, or kill was difficult and somewhat error prone. For example, I initially assumed that the word `record` indicated a write, but I later realized that it should not be considered an action word because it is used as a noun in the book application (ex. in the resource `bookstore/EditorialRecords.jsp`). Some words clearly

Subject	# Action Words	# Write	# Read	# Edit	# Kill
Masplas	3	2	1	0	0
Book	2	0	2	0	0
CPM	9	3	2	2	2
Logic	13	4	4	3	2
Logicv2	12	4	4	3	1
DSpace	11	2	6	2	1
Total	22	5	10	3	4

Table 5.3: Action words found in each application

indicate an interaction with the application state but have an unclear classification. For example, `login` must indicate at least a read, since logging in requires the password to be checked against the database but could also include an edit if information about the login is recorded to the database. I did not include these words in the set of action words, but future work could explore classifying these to produce a conservative dependency estimate. Other words seemed ambiguous in the initial pass at identifying and classifying action words but were actually acceptable action words. For example, `view` could be used as a noun or a verb but indicates a read in either case. Even the final set of action words that was used to estimate dependencies contained a word that was not used as my classification predicted: rather than implying a change to the application state (i.e., an edit) `change` was used in the Logic application to proceed to the next question in a quiz and therefore meant a read. In future work, a researcher could examine the resources in a larger set of applications to find a more exhaustive set of classified action words, which could be given to testers so that they do not need to manually identify or classify action words themselves. Thus, manually identifying and classifying action words is possible, albeit very difficult, which answers question 2.

Answering **question 3**, I found and classified 22 action words in the resources of the subject applications; the breakdown of how many write, read, edit, and kill words are in the unique resources of each application is shown in Table 5.3. Figures 5.2, 5.1, 5.3, and 5.4 show which action words were contained in the applications, broken down by the classification of the action words. The grey words are in an application's resources but not in the collected user access logs (i.e., none of the resources containing

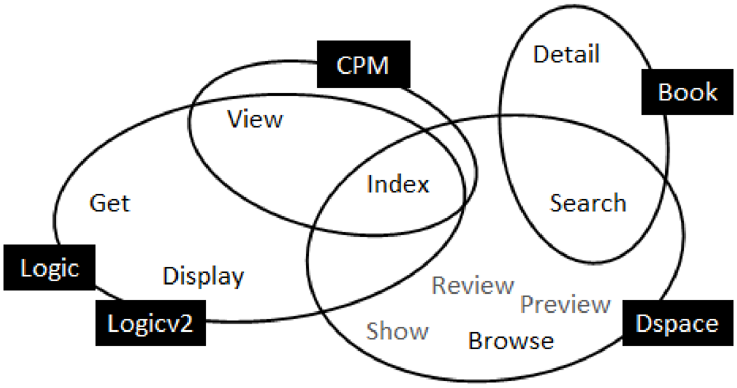


Figure 5.1: Manually identified "read" action words and which applications they are in.

the gray action words were accessed by real users). The high degree of overlap between action words in various applications suggests that action words are universal across a variety of applications.

Some action words may seem to be conspicuously absent. For example, `read`, `write`, `download`, and `store` could easily be classified with high confidence as read, write, write, and write, respectively; however, none of the resources that I examined contained these words, and I chose to only include action words that were actually present in the resources for the examined applications. I believe that a slightly larger sampling of applications would provide a more exhaustive list of the most common action words. In fact, just adding Logicv2 to the set of applications that I manually examined for action words would have revealed the action word `read`. I did not include the words `undo` or `import`, because, like `login`, their classification is ambiguous. It is unclear if `undo` is an edit or a kill, and import could be a write or a read depending on whether it indicates an external or internal import.

All of these issues point to the need for a configurable set of action words. With basic knowledge of an application, in the context of a specific application, testers could identify additional action words with a clearly defined classification or remove action words that were incorrectly classified.

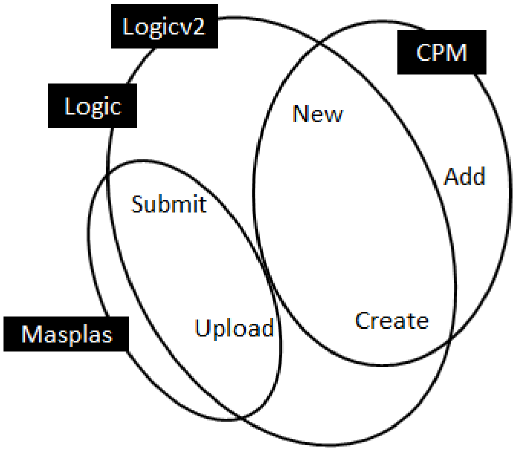


Figure 5.2: Manually identified "write" action words and which applications they are in.

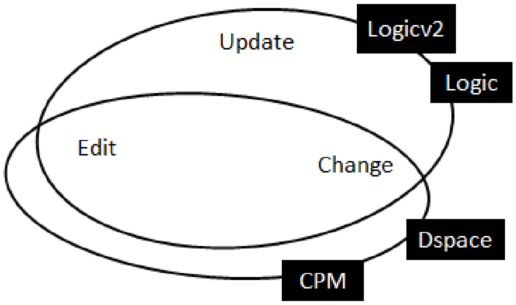


Figure 5.3: Manually identified "edit" action words and which applications they are in.

Thus, action words are relatively universal across a variety of applications, and manually identifying and classifying action words is not inhibitive to the proposed methodology for estimating dependencies, answering question 3.

5.2.3 Classifying Resources by Static Extensions and Action Words

To answer **question 4**, I wrote scripts to automatically classify resources in each application based on the static extensions or action words they contained. Given lists of static extensions and write, read, edit, and kill action words, the scripts generated a text file for each application with a list of resources denoted as independent, write,

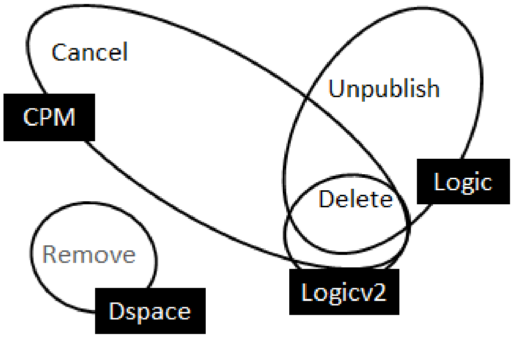
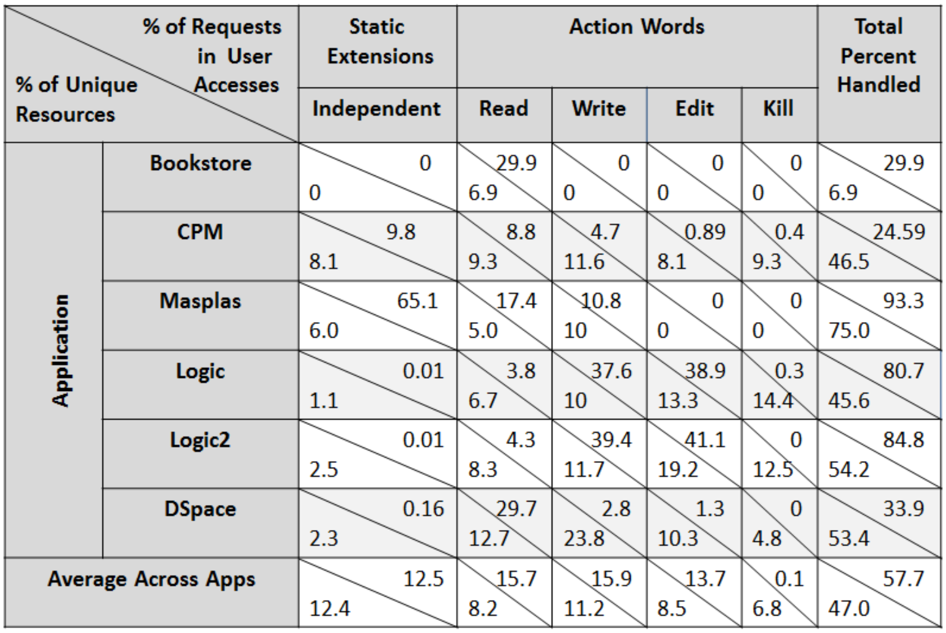


Figure 5.4: Manually identified “kill” action words and which applications they are in.

read, edit, kill, or unclassified. The execution of these scripts took on the order of a second for each application.

Table 5.4 shows the percent of unique resources and the percent of requests in the original set of user sessions containing a static extension or action word. The total percent handled indicates the percent of unique resources and the percent of requests that contain either a static extension or an action word—that is, the maximum percent of resources for which a dependency could be found. Although the lists of resources and collected user sessions that I analyzed had been scrubbed of many of the resources or requests with with static file extensions, classifying resources with static extensions as independent handles a sizable portion of the resources and user requests even in the scrubbed files—on average, 12.4% of unique resources and 12.5% of user requests in the collected user accesses. Overall, this small set of action words is present in a large percentage of the resources and user requests across all of these applications—on average, 47% of unique resources and 57.7% of user requests, at least 6.9% of unique resources and 24.59% of the user requests, and as much as 75% of the unique resources and 93% of the user requests. This adds further assurance that action words are universal across a variety of applications. Since these action words are present in such a large percentage of the requests in user accesses, the resources containing action words are commonly used.



% of Requests in User Accesses % of Unique Resources		Static Extensions	Action Words				Total Percent Handled
		Independent	Read	Write	Edit	Kill	
Application	Bookstore	0 0	29.9 6.9	0 0	0 0	0 0	29.9 6.9
	CPM	9.8 8.1	8.8 9.3	4.7 11.6	0.89 8.1	0.4 9.3	24.59 46.5
	Masplas	65.1 6.0	17.4 5.0	10.8 10	0 0	0 0	93.3 75.0
	Logic	0.01 1.1	3.8 6.7	37.6 10	38.9 13.3	0.3 14.4	80.7 45.6
	Logic2	0.01 2.5	4.3 8.3	39.4 11.7	41.1 19.2	0 12.5	84.8 54.2
	DSpace	0.16 2.3	29.7 12.7	2.8 23.8	1.3 10.3	0 4.8	33.9 53.4
Average Across Apps		12.5 12.4	15.7 8.2	15.9 11.2	13.7 8.5	0.1 6.8	57.7 47.0

Table 5.4: Percent of unique resources and requests in user accesses that contain a static extension or action word.

One remaining open question is whether we could address the possibility of requests with static extensions that generate files with static extensions. For example, the resource `logic2/Professor/ExportGrades.xls` generates a spreadsheet from students' grades in the data store.

A large percentage of unique resources and user requests in collected access logs across all of the subject applications that I examined contain a manually identified action word, which suggests that the text-based approach will estimate dependencies between resources that are representative of the most common user behavior.

5.2.4 Identifying Direct Objects

To answer **question 5** I took a semi-automated approach to finding direct objects because it was easier given the format of the data (i.e., the intermediate text file representations generated to be interpreted by a human). I expect the process could be fully automated by generating more consistent intermediate file (or by not generating intermediate files and just passing the data directly between scripts). I

Subject	# Potential Direct Objects	# Direct Objects
Masplas	4	1
Book	3	0
CPM	18	9
Logic	15	7
Logicv2	28	11
DSpace	74	53

Table 5.5: Direct objects found in each application

wrote scripts that found all of the words in the resources classified as write, read, edit, or kill that were not action words and refer to these as potential direct objects. These scripts executed in less than one second for each application. Then, I manually deleted potential direct objects that were stop words and potential direct objects that were only in one resource name until I was left with only actual direct objects. The stop words I used were: servlet, catch, to, for, by, and all file extensions.

Table 5.5 shows the number of potential direct objects and direct objects found for each application. The number of direct objects is small compared to the total number of unique resources, and the elimination process for potential direct objects significantly reduces the number of direct objects found. Since the number of direct objects for each application is much smaller than the number of unique resources, the direct objects are much easier for a tester to examine than the entire set of unique resources and seems to indicate that words are commonly reused throughout an application, thereby defining the vocabulary of the application.

For each application, the direct objects are some of the most common words in the collected user sessions. This shows that resources containing these direct objects, which this methodology is likely to find dependencies for, are commonly used. The direct objects identified for CPM, Logic, Logicv2, and DSpace made sense given the domain of the application. For example, `quiz` is a direct object in Logic and Logicv2, online tutorial applications, and `collection` is a direct object in DSpace, a document repository. Although no direct objects are found for Book and only one direct object was found for Masplas, this is not surprising since both are small applications and since Book was automatically generated, meaning that our intuition that a human developer would create resource names with intuitive meaning does not necessarily hold for this application. This methodology successfully identifies direct objects in the subject applications, answering **question 5**.

As I expected, the direct objects are not universal, answering **question 6**. There is only a small amount of overlap of direct objects in different applications (except between Logic and Logicv2, which is not surprising) and the overlapping direct objects seem relatively generic or domain-specific for applications within similar domains: `File` is a direct object in Masplas, Logicv2, and DSpace; `course` is a direct object in Logicv2 and CPM; `password` is a direct object in Logic, Logicv2, DSpace; `group` is a direct object in CPM and DSpace; and the direct object `question` in Logic and Logicv2 is the singular of the direct object `questions` in DSpace.

Thus, the proposed methodology for finding direct objects, which could easily be automated, find direct objects that are application- or domain-specific.

5.2.5 Assuming Dependencies

I wrote a script that creates a text file with the unique resources—classified as write, read, kill, or edit—grouped by the direct object contained in the resource name for each application. This script executed in less than one second. I manually assumed dependencies between these grouped resources to answer **question 7**. The number of direct objects determines the number of groupings of resource names and, thereby, the number of sets of dependent resources.

My methodology found several dependencies that were confirmed to be correct based on a manual check, confirming question 7, that we can correctly assume dependencies based on classified action words and direct objects. In Logic, for example, this methodology correctly estimated that the resource `Logic/Admin/CreateStudent` must occur before and cannot occur after `Logic/Admin/DeleteStudent`, since the latter resource represents a kill to the direct object `quizzes` being created by the former resource.

Subject	# of Dependent Resources Identified	% of Unique Resources
Masplas	2	10
Book	0	0
CPM	28	34
Logic	36	40
Logicv2	60	50
DSpace	105	49

Table 5.6: Number of dependent resources identified in unique resources for each application

Step 5 of the methodology for estimating dependencies, making assumptions about the dependency of resources containing one of the identified direct objects but no action word, would reveal several additional dependencies. For example, given that `quiz` is a direct object in the Logic application, a tester could easily determine that `Take` implies an access to the `quiz` data in the application state based on the context of the resource `Logic/Student/TakePracticeQuiz` and assumed dependencies between other resources that contain the direct object `quiz`.

Figure 5.6 shows the number of unique resources identified within a set of dependent resources for each application, and the percent of unique resources in each application for which a dependence relationship was found. The resources found to be dependent is significantly smaller than the entire set of unique resources and would be much easier for a tester to manually examine to confirm whether the estimated dependence relationships are correct.

The largest set of dependent resources is 17, found in Logic and DSpace (i.e., 17 unique resources contained an action word and a shared direct object). Currently, we assume that all of the reads within one of those sets of dependent resources are independent of one another and have the expected dependence relationship with all of the writes, kills, and edits. We make the same assumption for all of the writes, kills, and edits as well. Whether this assumption is valid is an open question that should be addressed in future work. For example, must all of the writes for one piece of data occur before any of the reads, edits, or kills for that data or can a piece of data be read, edited, or killed after just one write?

Other open questions that should be answered in order to more accurately estimate dependencies include:

1. What do we do when there are multiple action words in one resource? If they are both the same classification? For example, in the resource `dspace/pubs/submit/upload-file-list.jsp` in DSpace, both `submit` and `upload` are writes to the shared data store. If they are not the same classification? For example, in the resource `dspace/pubs/submit/cancel.jsp` in DSpace, `submit` is a read, but `cancel` is a kill.
2. What if a resource shows up in multiple groupings of dependent resources? For example, the resource `scheduler/servlet/ViewGroupGradesServlet` in CPM contains two direct objects—`Group` and `Grades`—and is, therefore, part of two groupings of dependent resources.

Currently I assume that there are no dependencies between different direct objects. Future work could examine relationships between direct objects. In Logic, for example, the data refered to by the direct object `question` is contained within the data referred to by the direct object `quiz`, and dependence relationships exist between resources containing these direct objects (ex. deleting a quiz also deletes a question).

I expect that the dependencies found with this methodology could help improve code coverage if the estimated dependence relationships were upheld in a generated test suite; however, several of the dependencies were over- or underestimates. For example, if one resource was incorrectly assumed to belong to a set of dependent resources, this methodology would overestimate that the incorrectly grouped resource had a dependence relationship with all of the other resources in that grouping. In Logic, the resource `Logic/documentation/studentView.jsp` was incorrectly associated with several other resources, including `Logic/Admin/CreateStudent`; however, the former resource is independent of the latter, since it simply refers to documentation that shows what a student user of the application would see rather than to an access of the `student` direct object. In CPM, the dependence between the resources containing the direct objects `demo` and `sched` were underestimated—these words were used as synonyms in the application but seen as independent direct objects by this methodology.

Since no direct objects were found for Book and since it was automatically generated, it is not surprising that there were also no dependencies discovered for Book. Similarly, only one dependence relation was found in Masplas between the two resources that contain the only direct object found for the application. Masplas is essentially a write-only application, so this is not surprising.

This methodology correctly identifies several dependencies. Although additional questions must be answered to improve the accuracy of this approach, I expect that generating test suites that correctly uphold the dependencies that this approach already finds would more realistically represent user behavior and improve code coverage.

5.3 Feasibility Study Conclusions

The feasibility study confirmed my intuitions that a text-based approach is promising. I have shown that it is possible to manually identify action words that indicate accesses or changes to a shared application state and classify them as write, read, edit, or kill. I found that action words are universal across applications of varying sizes from a wide range of domains and that direct objects are application- or domain-specific.

Although this approach can only estimate dependencies where the resource names reflect what the application does in terms of the data store, it does find dependencies between resources that are commonly used by actual users. Since each of the scripts I wrote to find dependencies executes in less than one second, my proposed methodology quickly estimates dependencies and significantly reduces the dependency results for a tester to check. This feasibility study also points to several ways that the text-based approach to estimating dependencies could be configured to improve results, which could be explored in future work.

Chapter 6

AUGMENTING THE TEST CASE GENERATION PROCESS TO UPHOLD DEPENDENCIES

Dependency estimations can be used to generate test cases that uphold test case dependencies and, therefore, achieve better code coverage. I propose augmenting the test case generation process developed by Sprenkle et al. [16] to uphold dependencies. The process, shown in Figure 6.1, takes collected user sessions as input, builds navigation and data models, and generates a suite of new test cases for the web application that represent users. From a set of user sessions and a navigation model specification, the intra-session navigation analyzer uses the navigation model and abstract test case criteria to produce a set of abstract test cases. Each abstract test case represents the navigation of a single user. That is, an abstract test case is a series of partially specified requests that include the resource and parameter names but not parameter values, like those shown in Figure 6.2. A data model is constructed through analysis of the original set of recorded user sessions. The abstract test cases and the data model are input to the test case generator to produce a set of test cases—the test suite.

In the proposed augmented test case generation process, a set of dependency estimates is input to the test case generator along with the abstract test cases and the data model, as shown in Figure 6.3. Instead of using only the data model to fill in parameter values in the abstract test suite, the dependence relationships would be leveraged as shown in Figure 6.2. Since no reads, edits, or kills to a specific piece of data can occur until it has been written, we use the data model to fill in the parameter value and anticipate a dependency violation if a read, edit, or kill request occurs before a write request. We can use the data model to determine the parameter values for write requests and keep track of the parameter values for direct objects that have been

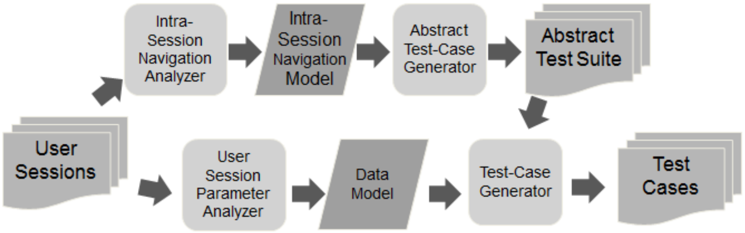

User Sessions
Intra-Session Navigation Analyzer
Intra-Session Navigation Model
Abstract Test-Case Generator
Abstract Test Suite
User Session Parameter Analyzer
Data Model
Test-Case Generator
Test Cases

Figure 6.1: Sprenkle et al.'s test case generation process [16].

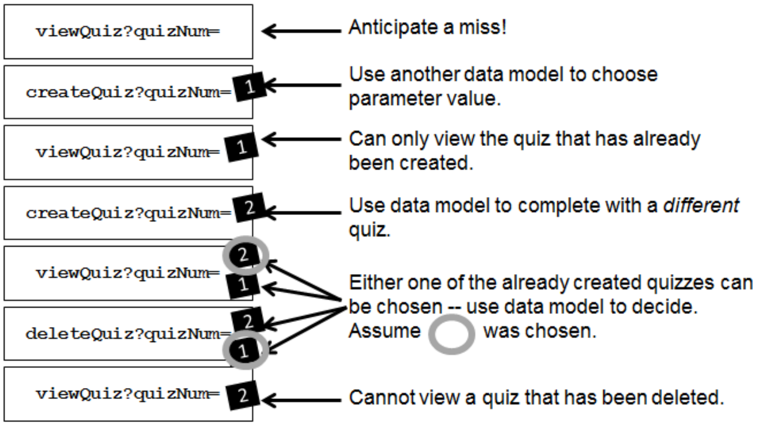

viewQuiz?quizNum=
Anticipate a miss!
createQuiz?quizNum= 1
Use another data model to choose parameter value.
viewQuiz?quizNum= 1
Can only view the quiz that has already been created.
createQuiz?quizNum= 2
Use data model to complete with a *different* quiz.
viewQuiz?quizNum= 2 1
deleteQuiz?quizNum= 2 1
Either one of the already created quizzes can be chosen -- use data model to decide. Assume ○ was chosen.
viewQuiz?quizNum= 2
Cannot view a quiz that has been deleted.

Figure 6.2: Example of augmentation process.

written. Subsequent reads, edits, or kills should only fill in parameter values that have been used in previous write requests. We also keep track of parameter values for direct objects that have been killed. Once a parameter value has been used in a kill, it should not be used in subsequent reads or edits. In any case where there is no "live" parameter value, the data model or when there is no known dependence relationship for a request, the data model is used to fill in the parameter value.

Since generating test suites has a low cost terms of time and space [16], a large number of test suites can be generated at a low cost. Knowledge of the anticipated dependency violations within a test suite could be used to choose test suites with the fewest anticipated violations, which are likely to achieve better code coverage.

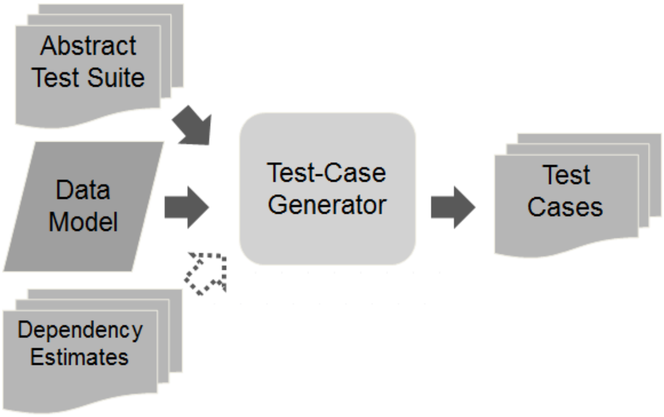
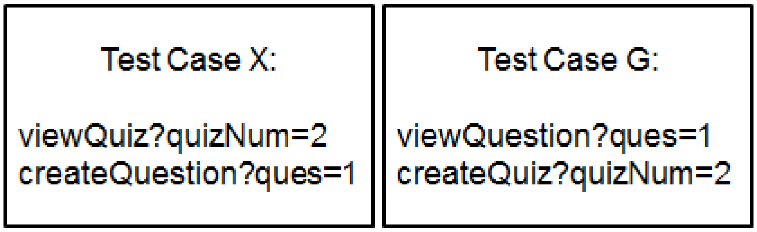


Figure 6.3: Proposed update to the test case generation process.


Test Case X:

viewQuiz?quizNum=2
createQuestion?ques=1

Test Case G:

viewQuestion?ques=1
createQuiz?quizNum=2

Figure 6.4: Example of test cases with no valid ordering that upholds all dependencies.

Although this proposed test case generation process augmentation is for Sprenkle et al.'s test case generation process, I believe that it could be applied to other automated test case generation processes.

An alternative approach to generating test cases that uphold the estimated dependencies is reordering the generated test cases. This process should be analogous to def-use in compilers, but it is complicated because a valid ordering is not always possible because a test case may include multiple requests with contradicting dependence relationships. For example, in Figure 6.4, the quiz cannot be viewed until it is created, but the question also cannot be viewed before it is created. Determining the best possible ordering (i.e., the ordering with the fewest dependency violations) is difficult.

Chapter 7

CONTRIBUTIONS AND FUTURE WORK

In this chapter, I reflect on the contributions of my work presented in this thesis and propose directions for future work.

7.1 Contributions

The contributions of my work presented in this thesis are:

1. **Identification of the inter-test-case dependency problem**—that dependencies between test cases in automatically generated test suites are not represented by current test case generation processes and are often violated, which significantly limits the code coverage of generated test suites.

2. **A text-based methodology for estimating dependencies** that leverages the intuition that web application requests contain embedded context since they were written by human developers. Action words within a resource can indicate a write, read, edit, or kill to the shared application state and direct objects within a resource indicate which portion of the application state is affected. My key insight is that dependency estimates can be made based on these action words and direct objects.

3. **The design, implementation, and analysis the results of a feasibility study** that shows the promise of a text-based approach to estimating dependencies. I showed that action words are universal across a variety of applications, and that a relatively small set of action words is contained in a large number of the resources of the applications that I studied. Direct objects are application- or domain-specific, and can be found automatically given a set of resources and action words. The direct objects found for the applications I studied matched our expectations of the applications' vocabulary relatively well. Finally, this study showed that correct dependencies can be discovered with this approach.

4. **An augmented test case generation process** to produce test suites that uphold inter-test-case dependencies or detect anticipated dependency violations.

7.2 Future Work

Future work in estimating inter-test-case dependencies includes:

1. **Methodology improvements**, such as identifying and classifying a more exhaustive set of action words by examining more web applications, exploring configuration options to allow testers to more accurately estimate dependencies in the context of a specific application,

2. **Implementation of the alternative approach to estimating dependencies**, proposed in Chapter 4, which involves instrumenting application code and monitoring the database to determine which requests result in a write, read, edit, or kill to the database and what portion of the database is accessed or changed,

3. **Evaluating** the accuracy of dependency estimates from the text-based approach and the text-based approach compared to manually identified dependencies, and

4. **Applying dependencies** to the data or navigation model and to test suite reduction. Evaluate test suites generated using the augmented test case generation process in terms of code coverage relative to test suites generated without knowledge of inter-test-case dependencies.

BIBLIOGRAPHY

[1] Manar H. Alalfi, James R. Cordy, and Thomas R. Dean. Modelling methods for web application verification and testing: state of the art. *Software Testing, Verification, and Reliability*, 19(4):265–296, 2009.

[2] Anneliese A. Andrews, Jeff Offutt, Curtis Dyreson, Christopher J. Mallery, Kshamta Jerath, and Roger Alexander. Scalability issues with using FSMs to test web applications. *Information and Software Technology*, 2009.

[3] Shay Artzi, Adam Kieżun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D. Ernst. Finding bugs in dynamic web applications. In *Int'l Symp. on Software Testing and Analysis*, July 2008.

[4] Cobertura. http://cobertura.sourceforge.net/, 2012.

[5] G Di Lucca, A. Fasolino, F. Faralli, and U.D. Carlini. Testing web applications. In *International Conference on Software Maintenance*, 2002.

[6] DSpace Federation. http://www.dspace.org/, 2012.

[7] Eric Enslen, Emily Hill, Lori Pollock, and K Vijay-Shanker. Mining source code to automatically split identifiers for software analysis. In *6th IEEE Working Conference on Mining Software Repositories (MSR)*, May 2009.

[8] Open source web applications with source code. http://www.gotocode.com, 2003.

[9] Emily Hill, Zachary P. Fry, Haley Boyd, Giriprasad Sridhara, Yana Novikova, Lori Pollock, and K. Vijay-Shanker. Amap: Automatically mining abbreviation expansions in programs to enhance software maintenance tools. In *MSR 2008: 5th Working Conference on Mining Software Repositories*, May 2008.

[10] Chaitanya Kallepalli and Jeff Tian. Measuring and modeling usage and reliability for statistical web testing. *IEEE Transactions on Software Engineering*, 27(11):1023–1036, 2001.

[11] Chien-Hung Liu, Kung D. C., Pei Hsia, and Chih-Tung Hsu. Structural testing of web applications. In *International Symposium on Software Reliability Engineering (ISSRE)*, 2000.

[12] Filippo Ricca and Paolo Tonella. Analysis and testing of web applications. In *Int'l Conf. on Software Engineering (ICSE)*, 2001.

[13] Jessica Sant, Amie Souter, and Lloyd Greenwald. An exploration of statistical models of automated test case generation. In *International Workshop on Dynamic Analysis (WODA)*, May 2005.

[14] Sara Sprenkle, Camille Cobb, and Lori Pollock. Leveraging user-privilege classification to customize usage-based statistical models of web applications. In *International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Apr 2012.

[15] Sara Sprenkle, Emily Gibson, Sreedevi Sampath, and Lori Pollock. A case study of automatically creating test suites from web application field data. In *Workshop on Testing, Analysis, and Verification of Web Services and Applications*, 2006.

[16] Sara Sprenkle, Lori Pollock, and Lucy Simko. A study of usage-based navigation models and generated abstract test cases for web applications. In *International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Mar 2011.

[17] Paolo Tonella and Filippo Ricca. Statistical testing of web applications. *Journal of Software Maintenance and Evolution*, 16(1-2):103–127, 2004.

[18] Wenhua Wang, Yu Lei, Sreedevi Sampath, Raghu Kacker, Rick Kuhn, and James Lawrence. A combinatorial approach to building navigation graphs for dynamic web applications. In *International Conference on Software Maintenance (ICSM)*, 2009.

[19] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. Dynamic test input generation for web applications. In *Int'l Symp. on Software Testing and Analysis*, 2008.