# PROBING ANALYSIS OF CLOSED COMPONENTS

by

Marc Fisher II

A DISSERTATION

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfillment of Requirements

For the Degree of Doctor of Philosophy

Major: Computer Science

Under the Supervision of Professors Gregg Rothermel and Sebastian Elbaum

Lincoln, Nebraska

August, 2008

# PROBING ANALYSIS OF CLOSED COMPONENTS

Marc Fisher II, Ph.D.

University of Nebraska, 2008

Advisors: Gregg Rothermel and Sebastian Elbaum

Software is increasingly being built through the composition of exisiting components, such as web services, where access to the source code, and in some cases even the executable code, is unavailable. The users of these components must rely on the existing documentation of the component to understand how the component should be used. However, this documentation is often imprecise or incomplete (when generated automatically via static analysis) or does not match the current version of the service (when maintained manually). Additionally, users of closed components have no control over the delivered quality of the component, and currently rely of the developers' claims or ad-hoc testing to assess the quality of the component. In this work, we present probing analysis, a new analysis technique designed specifically to address these issues for closed components.

Probing analysis is a black-box dynamic analysis technique that probes a component by generating inputs for the component, executing the component on those inputs, and analyzing the inputs and outputs to infer properties that describe some aspect of the behavior of the component. We present a formal definition of probing analysis that can be used to guide the development of probing analysis techniques. We then develop a probing analysis methodology for web applications and web services called WebAppSleuth.

We have implemented versions of WebAppSleuth for web applications and for web services. We apply these versions to six production web applications and two

commercial web services. WebAppSleuth was able to accurately and quickly infer properties for the six web application and found anomalous behavior in four of the web applications. Additionally WebAppSleuth was able to find several ways that the two web services could be improved.

These results suggest that probing analysis is a useful new technique for analyzing the behavior of software components and the provided definition of probing analysis provides a framework for the development of new probing analysis techniques.

# ACKNOWLEDGEMENTS

I would like to thank my advisors, Gregg Rothermel and Sebastian Elbaum, who have guided my work on this dissertation and my education in general. They have consistently provided advice and support without which I would have not completed this work.

I would also like to thank Matt Dwyer and David Olson for serving on my committee and providing insightful commentary and questions that have improved this work.

I am also grateful to various members of the ESQuaReD research group who have provided assistance throughout my work on this. In particular, Myra Cohen who provided the covering array generation tool that was used in Chapter 4, Kalyan-Ram Chilakamarri who started the work on characterizing web applications and worked with me during the early portions of this work, and Joseph Ruthruff, Madeline Diep, and other members of the group who have served as sounding boards for various aspects of this work.

Finally, I would like to thank my family, especially my children, Joel and Elizabeth, who have patiently supported me as I have achieved my dream of getting my doctorate.

This work was supported in part by NSF CAREER Award 0347518, the EUSES Consortium through NSF-ITR 0325273 and NSF-ITR 0324861, and ARO DURIP award W911NF-04-1-0104.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Chapter 1

# Introduction



## 1.1 Motivation

As software becomes more complex, it is increasingly common for applications to be built by composing existing applications or components. On the desktop, many common applications are frequently used as components in other applications. For example, Microsoft's Office products all support being used as components in other applications developed in .NET.

The growth of the internet has also spurred the development of web applications and services that can be used as components. For example, a wide variety of businesses that provide services to other businesses, such as shipping companies and credit card processing organizations, provide a web service for accessing those services. Further, some businesses that target consumers, such as Amazon or eBay, also provide access to their products or services via web services. By doing so, these companies allow outside parties to utilize or resell their services or products. According to the companies that provide these services, web services are a significant part of their business. For example, eBay claims that third party applications account for over

25% their listings [21] and Amazon claims that over 200,000 developers, start-ups, and Fortune 1000 companies use their technologies [2].

Components are often developed by someone other than the person using them, and generally the source code and, in the case of web-based applications and services, the executable code is not available for analysis. We call these components where the user has limited or no access to the source or executable code *closed components*.

When discussing closed components, we differentiate between two roles: the *user* and the *developer*. The developer of a closed component is the person or organization that actually implements the closed component. A user of a closed component is a person or organization that wishes to integrate the functionality provided by the closed component into their own application.

Since a user of a closed component has limited access to the code for that component, they must rely on the documentation for that component. However, documentation often has shortcomings. Documentation for closed components, when available, comes in two primary formats, informal textual documentation written manually by the developer of the component, and a formal interface description such as in a web service description language (WSDL) file. Since textual documentation is manually maintained, the maintainer is often able to include semantic information about the component that is not available from automated analysis, such as temporal dependencies between operations. However, since the textual documentation is generally maintained separately from the actual component, it is not uncommon for it to be out of sync with the actual component. Additionally, since it is informal it is often inconsistent in the level of detail it provides and is not amenable to use by automated tools that analyze or test components.

Formal interface descriptions, such as WSDL files, are frequently automatically generated from the component itself, and therefore tend to stay synchronized with

the component. Also, the formal nature of these descriptions are amenable to (and often intended for) use by automated tools. However, the type of information that can be included in these formal specifications is often limited (e.g., most of these formats do not include any form of temporal specifications), and in many cases, since they are automatically generated via static analysis, they are not precise, for example marking required parameters as optional or using more general types than what the component will actually accept or return.

In addition to having to rely on documentation that may not be present or may be incomplete, users of closed components do not have control over the delivered quality of the component. Therefore a methodology that can analyze the component, and find anomalies in the behavior of the component, can help users determine if the component is suitable for the task at hand.

In summary individuals working with closed component need a methodology to address the following issues:

- Textual documentation for closed components is hard to maintain and keep synchronized with the component. However, it is currently the primary source of certain types of semantic information such as temporal relationships.

- Formal documentation generated through static analysis, such as WSDL files, is often imprecise, incomplete, and provides no mechanisms for encoding certain types of semantic information.

- Users of closed components have no control over the delivered quality of the component, and currently rely on the developers' claims or ad-hoc testing to assess the quality of the component.

## 1.2 Approach

To address these problems in understanding and assessing closed components, as well as find anomalies in and suggest improvements for closed components, we have developed a new form of analysis technique, *probing analysis*. Probing analysis is a black-box dynamic analysis technique that generates large numbers of inputs for a closed component, executes the component with those inputs, and analyzes the results to generate inferences which can serve as a partial model of the behavior of the component. This process can be useful for a variety of applications such as automated anomaly detection, generation of suggestions for improving the component, and providing additional documentation about the behavior of the component, and is suitable for use both by developers and users of closed components.

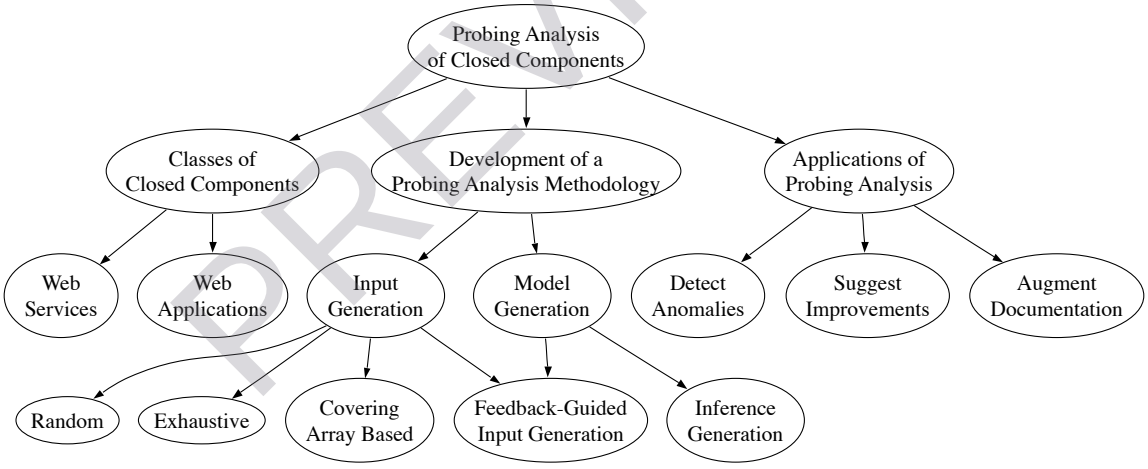


Figure 1.1: Overview of work

Figure 1.1 provides an overview of how we approach the problems of developing probing analysis and applying it to the analysis of closed components. We decompose *probing analysis of closed components* along three different tracks, the different *classes of closed components*, the *development of a probing analysis methodology*, and the different *applications of probing analysis*. In the next three sections we identify areas

of interest within each of these tracks. Section 1.3 describes how we combine the different aspects of these three tracks.

### 1.2.1 Classes of Closed Components

As stated earlier, probing analysis is primarily useful in contexts where the person applying the analysis has limited or no access to the source code, or possibly even the executable code of the component being analyzed. Therefore we have determined that components provided as *web services* and *web applications* are of particular interest and have focused on these classes of components. These components are usually hosted on servers owned by the developers of the components, and the users of the components only have access to them through a network interface that allows them to submit requests and receive responses.

Web services are generally provided by companies to provide access to or allow the resale of services offered by that company. An example of this is in the shipping industry. All the major shipping companies (e.g. FedEx, UPS, USPS) provide web services that allow online retailers to receive quotes on the cost of shipping. These services have been designed to conform to standards that allow them to be readily integrated into applications being built by others, and usually have both informal documentation and a formal WSDL definition.

Web applications use technologies similar to those used by web services, but provide an HTML user interface and are intended to be used directly by the end user. Generally these applications are not intended for use as components in other applications, but often they provide features that are not readily available elsewhere so they are used as components in other applications. One common example of this is the use of existing web applications in web mash-ups, web applications that combine information from two or more other web applications. Since web applications are

not intended to be reused, they generally do not include any documentation, and therefore it is up to the user to reverse engineer the interface to the application by examining the HTML.

### 1.2.2 Development of a Probing Analysis Methodology

The overall probing analysis process can be broken down into two major steps: *input generation* and *model generation*. In order for probing analysis to be effective, it needs to be able to generate a set of inputs (sequences of method or function calls or of HTTP requests) that exercise the component being studied in interesting ways. Toward that end we present various methods of generating inputs, including random and exhaustive techniques and some combinatorial selection techniques.

For model generation, we have built on the work being done in automatic generation of inferences, such as the work by Ernst, et al. on Daikon [26, 35, 56]. Daikon works by instrumenting a component being analyzed, and executing that component on some set of test cases. It then uses the data collected by instrumentation to statistically infer likely invariants in the behavior of the component, for example that some variable always has one of a limited set of values. Since we are unable to directly instrument closed components, we are limited to making inferences that use only provided inputs and the actual output. Therefore we have developed suitable *inference generation* algorithms that use only the inputs and outputs. This set of inferences can then be used as a partial model of the component.

Finally, there is a significant opportunity to feed information derived from the output back into the input generation step. A simple example of this is when generating sequences of calls to a web service. Each call to the web service returns a set of values that can then be used in subsequent calls to the same service. More complex cases could include generating queries targeted toward refining or disproving hypoth-

esized inferences. Thus, we have developed several *feedback-guided input generation* techniques.

### 1.2.3 Applications of Probing Analysis

Section 1.1 listed a number of issues with using closed components, including difficulty assessing the quality of closed components and incomplete or missing documentation. Probing analysis can address each of these issues.

Probing analysis executes the component with a wide range of inputs. Constructing a test suite with oracles for such a large number of test cases, and manually inspecting the results of executing that test suite is difficult and time-consuming. Probing analysis eases this burden by automatically generating inputs and using a simple, partial oracle to generate a small set of inferred properties that are amenable to inspection. This inspection can help users or developers *identify anomalies* in the behavior of the component that would normally have been hidden by the volume of testing, which can lead to the identification of quality problems in the component.

Additionally, in some domains there are patterns of anomalies that can be automatically identified, e.g., web services often have incorrectly specified minimum numbers of occurrences for elements. When these anomalies are automatically identified, it is often possible to directly *suggest improvements* to the component to correct them.

Finally, the properties themselves can be used to *augment existing documentation*. Like the formal interface description, the properties can easily be kept in sync with the actual implementation by reapplying the analysis to changed versions of the component. In addition, they also can include more complex specifications, such as temporal relationships between operations, that are usually found only in the informal documentation.

## 1.3 Organization of Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 provides necessary background on web applications and web services. It also includes discussion of early work on user-session based testing of web applications that helped to motivate the development of probing analysis. Chapter 3 includes a more complete definition of probing analysis and describes the core of WebAppSleuth, our implementation of probing analysis for web applications and services. Chapters 4 and 5 describe how WebAppSleuth was modified for web applications and web services, respectively, and include empirical evaluations on live, production systems. Chapter 6 describes related work on test input generation, dynamic property inference, black-box analysis, and web dependability. Chapter 7 provides overall conclusions for this work and suggests areas for further study.

# Chapter 2

# Background



As mentioned in Chapter 1, we have chosen to focus on web applications and services. Throughout the remainder of this dissertation we use the following definitions of web applications and web services:

**Definition: Web Applications** are applications that are hosted on a web server and are accessed by end-users using a standard web browser such as FireFox or Internet Explorer.

**Definition: Web Services** are applications that are hosted on a web server and are designed to be used as components in other applications that communicate with the service via standardized technologies and an API described by, for example, a web service description language (WSDL) document.

The following sections provide necessary technical background on web applications and web services. Section 2.3 describes early work we did on user-session-based testing of web applications that provided some of the impetus for this work.

## 2.1 Web Applications

Navigating through the WWW can be perceived as submitting a sequence of requests to and rendering the responses from a multitude of servers. Browsers assemble such requests as the user clicks on links. Servers generate responses to address those requests, the responses are channelled through the web to the client, and then they are processed by the browser.

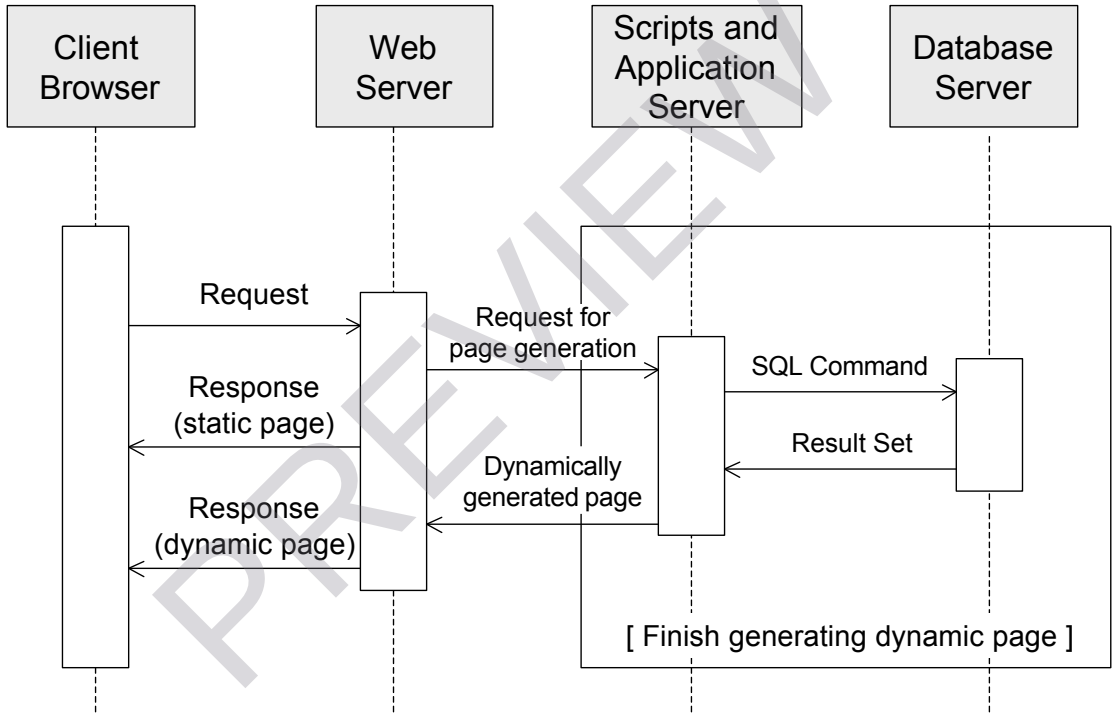




Figure 2.1: Sequence diagram of a typical web application request

Figure 2.1 provides a sequence diagram for a typical request to a web server. The user initiates a request in the web browser, typically by clicking on a link or clicking on the submit button for the form. Users can provide data through forms consisting of input fields (e.g., radio buttons, text fields) that can be manipulated by a visitor (e.g, click on a radio button, enter text in a field) to tailor a request. These input fields can be thought of as variables. Some of the variables have predefined sets of

potential values (e.g., radio-buttons, list-boxes), while others are set by the user (e.g., text fields).

After the user clicks the link or the submit button, the browser is responsible for assembling a request from the data the user has provided. Any form fields are sent as request parameters known as name-value pairs (input fields' names and their values). This request is then passed to the web server via the HTTP protocol.

The web server is responsible for either returning the resource named in the request (if a request is for a static page) or for dispatching the request to a script or an application server that knows how to handle that particular request. For most non-trivial applications the application server interacts with a database server to get the necessary information for constructing the response. The application server then generates a page, which is passed to the web server, which returns the page to the browser where it is rendered for the user to view.

Of primary importance to our work is the interaction between the browser and the web server. Probing analysis treats everything in the sequence diagram to the right of the web server as a black-box and interacts with this through requests whose structure is defined by links and forms found in pages returned by the server.



## 2.2 Web Services

Interactions with a web service are similar to interactions with a web application. The primary differences are that a request is not initiated by a user in a web browser, but is instead initiated by some application that makes use of the service, and the response is not an HTML page to be rendered for the user, but is some data, usually in an XML document, that is processed by the requesting application. The interface for a web service is generally described in a WSDL document.

Figure 2.2 is a partial WSDL file describing an example bookstore web service that conforms to the WSDL 1.1 standard [14]. This service allows client applications to search for books, login with an existing username and password, add books to a shopping cart and change the quantities of items already in the cart.

A WSDL file consists of six kinds of definitions: types, messages, port types, bindings, ports, and services. Together the type (lines 2-95) and message definitions (omitted) describe the overall structure of request and response messages. The port type definitions (lines 120-140) describe a set of abstract operations and which messages are used as inputs and outputs of each operation, while the bindings (omitted) define the concrete protocols for the abstract operations of port types. A port definition (omitted) specifies the address for a particular binding and a service (omitted) is used to aggregate a set of related ports.

Although all of these definitions are important when describing a web service, in this work we focus on type definitions. The type definitions in a WSDL file tend to make up the majority of the document. For example, the WSDL file describing Amazon's E-Commerce service is 3,244 lines long, with a type definition that is 2,850 lines long,[1] while the WSDL description for eBay's Trading web service is 90,450 lines long with a type definition that is 87,725 lines long.[2]

In addition to constituting the largest portion of most WSDL documents, type definitions are often very complex, with highly nested structures and complex relationships between different parts of the definition. Although the WSDL specification does not specify a single standard notation for defining the types in the document, XML schemas have become the de facto standard. An XML schema defines a set of XML elements and describes the types of the XML elements using simple and

[1] `http://webservices.amazon.com/AWSECommerceService/2007-07-16/AWSECommerceService.wsdl`

[2] `http://developer.ebay.com/webservices/515/eBaySvc.wsdl`

```
<definitions name="bookstore">
  <types>
    <schema>
      <simpleType name="Category">
        <restriction base="string">
          <enumeration value="Databases" />
          <enumeration value="Web Design" />
          <enumeration value="Programming" />
        </restriction>
      </simpleType>
      <element name="StartSessionResponse">
        <complexType>
          <all>
            <element name="sessionId" type="string" />
          </all>
        </complexType>
      </element>
      <element name="SearchRequest">
        <complexType>
          <all>
            <element name="sessionId" type="string" />
            <element name="category" type="Category" minOccurs="0" />
            <element name="name" type="string" minOccurs="0" />
            <element name="priceMin" type="decimal" minOccurs="0" />
            <element name="priceMax" type="decimal" minOccurs="0" />
          </all>
        </complexType>
      </element>
      <element name="SearchResponse">
        <complexType>
          <all>
            <element name="bookDetail" minOccurs="0" maxOccurs="unbounded">
              <complexType>
                <all>
                  <element name="itemId" type="integer" />
                  <element name="name" type="string" />
                  <element name="price" type="decimal" />
                  <element name="category" type="Category" />
                </all>
              </complexType>
            </element>
          </all>
        </complexType>
      </element>
      <element name="LoginRequest">
        <complexType>
          <all>
            <element name="sessionId" type="string" />
            <element name="login" type="string" />
            <element name="password" type="string" />
          </all>
        </complexType>
      </element>
      <element name="LoginResponse'>
        <complexType>
          <all>
            <element name="success" type="boolean" />
          </all>
        </complexType>
      </element>
```



Figure 2.2: Bookstore web service description file

```
      <element name="AddToCartRequest">
        <complexType>
          <all>
            <element name="sessionId" type="string" />
            <element name="itemId" type="integer" />
            <element name="quantity" type="integer" />
          </all>
        </complexType>
      </element>
      <element name="UpdateQtyInCartRequest">
        <complexType>
          <all>
            <element name="sessionId" type="string" />
            <element name="orderId" type="integer" />
            <element name="quantity" type="integer" />
          </all>
        </complexType>
      </element>
      <element name="CartResponse">
        <complexType>
          <all>
            <element name="cartItem" minOccurs="0" maxOccurs="unbounded">
              <complexType>
                <all>
                  <element name="orderId" type="integer" />
                  <element name="itemId" type="integer" />
                  <element name="quantity" type="integer" />
                </all>
              </complexType>
            </element>
          </all>
        </complexType>
      </element>
    </schema>
  </types>
            (Lines 96-119 omitted)
  <portType name="BookstoreAPI">
    <operation name="StartSession">
      <output message="StartSessionResponse" />
    </operation>
    <operation name="Search">
      <input message="SearchRequest" />
      <output message="SearchResponse" />
    </operation>
    <operation name="Login">
      <input message="LoginRequest" />
      <output message="LoginResponse" />
    </operation>
    <operation name="AddToCart">
      <input message="AddToCartRequest" />
      <output message="CartResponse" />
    </operation>
    <operation name="UpdateQtyInCart">
      <input message="UpdateQtyInCartRequest" />
      <output message="CartResponse" />
    </operation>
  </portType>
            (Lines 141-193 omitted)
</definitions>
```

Figure 2.2 (continued)