AutoDBT: A Framework for Automatic Testing of Web Database Applications

By

Lihua Ran

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

December 2004

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of Lihua Ran find it satisfactory and recommend that it be accepted.

Chair

Acknowledgment

I would like to express my gratitude to all the people helping me in making the thesis possible. First, many thanks to my advisor, Dr. Dyreson, for his brilliant insights and endless patience. It's been a pleasure working with him. Secondly, thanks to Dr. Andrews for bringing out this topic that leading me to explore this new exciting research area, and also thanks to Dr. Medidi for being in my committee and giving me many kind suggestions. Thanks to my husband, my parents and parents-in-law for their love and support. Thanks to my friend GaoYan Xie for many good helpful discussions and thanks to his wife Yun Jiang for her kind help with my little baby. Finally, thanks to my lovely little daughter Joy for all the joy and kisses she has given me.

AutoDBT: A Framework for Automatic Testing of Web Database Applications

Abstract

by Lihua Ran, M.S.
Washington State University
December 2004

Chair: Curtis Dyreson

The complex functionalities and high demands of software quality make manual testing of a web application ineffective. Automatic software testing methods can help to determine if an web application is working correctly, but existing methods are unable to test whether such an application interacts correctly with a back-end database. This paper elaborates an approach, called the Automatic Database Tester (AutoDBT), that extends the functional or black-box testing of a web database application to include database updates. AutoDBT takes as input a model of the application and a set of testing criteria. The model consists of a state transition diagram showing how a user navigates among pages in the application, and a data specification which captures how data flows in the application and how the database is updated. AutoDBT uses the model along with the test criteria to generate test cases for functional testing of the application. AutoDBT can also generate an oracle to validate whether a back-end database is updated correctly during a test. This paper first reports on the design and architecture of AutoDBT, then the generality and the implementation plan of the AutoDBT have been discussed.

Contents

Acknowledgment iii

Abstract iv

List of Figures ix

1 Introduction **1**

2 Motivating Example **6**

3 Framework of Problem **9**

3.1 Formal Description of Testing Web Application without Database 9

3.2 Formal Description of Testing Web Database Application . . . 12

3.2.1 Dirty Test Suite Approach 17

3.2.2 Clean Test Case Approach 18

3.2.3 Tradeoff between these two Approaches 18

3.3 One Possible Solution of Testing
Web Database Application 19

4 The AutoDBT Framework 23

4.1 Overview 23

4.2 DEFSM Modeling 24

4.2.1 State Transition Diagram 25

4.2.2 Data Specification 27

4.3 Test Sequence Generator 36

4.4 Dynamic Data Specification Generator 37

4.4.1 Update List 38

4.4.2 Data Inheritance Graph Generation 40

4.4.3 Dynamic Input Specification Generation 41

4.5 Summary 44

5 The Testing Process 45

5.1 Test Sequence Scheduler 46

5.2 Guard Generation and Evaluation 48

5.3 Step Mode Execution and Test Result Evaluation 50

5.3.1 On-demand Oracle Generator 50

6 Analysis of the Test Report 54

7 Generality Analysis 57

7.1 Basic Cases of Database Update 58

7.2 Generality Argument 61

8 Implementation Architecture 65

8.1 Overview 65

8.1.1 Test Sequence Generator 65

8.2 Dynamic Data Specification Generator 66

8.3 Test Sequence Scheduler 68

8.4 Guard Generator 69

8.5 Oracle Generator 72

8.6 Step Mode Execution 73

8.7 Test Result Evaluation 73

8.8 Test Controller 73

9 Related Work 74

10 Conclusions and Future Work 78

List of Figures

1.1 A web application 2

2.1 The online car rental system (OCRS) 8

3.1 The correspondence of pages between the specification and implementation of page sequence $P_0, \ldots, P_m$ 12

3.2 The correspondence of AD states between the specification and implementation of page sequence $P_0, \ldots, P_m$ 14

4.1 AutoDBT's framework 25

4.2 OCRS's state transition diagram 26

5.1 Framework of the testing process 47

7.1 Implementation Framework 64

Chapter 1

Introduction

Web applications are increasing in importance as consumers use the web for a wide range of daily activities such as online shopping and banking. Though web applications differ widely in their functionality and in the technologies used in their implementation, at heart, these applications share the architecture depicted in Figure 1.1. The figure shows that a web application interfaces with users through a set of (HTML) forms. The forms collect input data that is processed by the application. Typically, a web application is supported by a back-end *application database* (AD), which is updated in response to user actions.

Testing is critically important in ensuring that a web application has been implemented correctly. Several methods have been developed for testing web

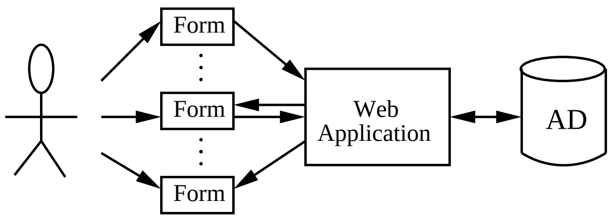


Figure 1.1: A web application

applications (see Section 9), but current testing methods do not adequately incorporate testing of the application database. Surprisingly, relatively little attention has been given to developing systematic techniques for assuring the correctness of the interactions between a database management system (DBMS) and a web application program, although substantial effort has been devoted to ensuring that the algorithms and data structures implemented in a DBMS work efficiently and protect the integrity of the data [21, 22]. Given the critical role that web applications play in e-commerce, there is a need for new approaches to assess their quality.

This paper presents a framework for the functional or black-box testing of web database applications called AutoDBT. Functional testing evaluates the correctness of an application by judging whether the application passes or fails selected test cases. The judge is called an *oracle* since it incorporates a prediction of how the application should have performed had it been implemented correctly. Previously, we developed an approach to perform functional testing of web applications [1], we then extended our previous work to include the interaction of the application with a database [2]. Additional problems that emerge from this interaction are listed below.

- Evolving database state - An in-memory application can start a test from a known, clean state by restarting the application for each test. But when a database is involved, the state of the database might be different at the start of each test since database modifications accumulate over time as individual tests update the database. AutoDBT automatically generates a *guard* query for each test case. The guard determines whether the test can be performed given the current state of the database.

- Modeling of database updates - As the application executes, it will update the database. AutoDBT allows a software tester to specify, at a high-level, the database updates associated with page transitions.

- Correctly choosing test data - Each test case is self-contained in the sense that it contains all of the test data that is input to the set of forms during evaluation of the test. A test of an in-memory application can manufacture the test data as needed since the application does not store data between tests. But when testing a web database application, the test data must be carefully selected. Some tests are based on data that must be present in a database (e.g., a test to modify an existing car reservation needs that reservation to be in the database prior to the test) while other tests require that the same data be absent (e.g., a test to create a car reservation demands that the reservation be new to the database). AutoDBT distinguishes between data that should be drawn from the application database, and data that is not present in the application database, and should instead be chosen from a database of synthesized test data, called the *synthetic database* (SD).

- Correctness of the final database state - An oracle reports on the correctness of a software test. An in-memory web application passes a test if it ends up in the expected final state (i.e., at the expected page). For a web database application, however, the expected final state also has to include the correct state of the database. AutoDBT automatically generates an oracle that can, in part, determine if an application correctly updates its database.

In this paper, we extend our previous work [2] to include the formal framework of the problem, the analysis of the test report, the generality of the approach and the implementation plan of the AutoDBT.

The next section introduces an example that is used to illustrate the AutoDBT framework. A formal description of testing the web application is given in Section 3. An overview of the framework is given in Section 4. The section also describes how to model the page navigation, how to specify the input data in each transition, and how the database is updated. Section 5 outlines the test case generator. The generator is used to generate test cases for testing of the application. Section 6 discusses how to analyze the test report. Section 7 discusses the generality of the AutoDBT. Section 8 gives the implementation plan of the AutoDBT. Section 9 compares our contribution to related work, both in the area of database technology and in the area of testing of web applications. Finally, the paper concludes and presents a few ideas for future work.

Chapter 2

Motivating Example

This section presents a fragment of a web application that will be used to illustrate the ideas in this paper. The example application is an online car rental system (OCRS). On the main page shown in Figure 2.1(a), there are buttons to enable a customer to make a new reservation, to view a current reservation, and to modify or cancel an existing reservation. For instance, consider the third button, "Modify Reservation." When this button is activated, the customer is led to a new page as shown in Figure 2.1(b). On the modify reservation page there are three text input boxes: First Name, Last Name and Confirmation #. After entering the required data and activating the "Continue" button, the desired car reservation, consisting of a Pickup Date, a Return Date, and a Car Type, is shown in Figure 2.1(c). This page allows the car reservation data to be updated. After activating the “Confirmation” button, the updated reservation information is displayed in Figure 2.1(d). The modification process can be terminated at any point by activating the “Cancel” or “Back to Main Page” button, which transfers the customer to the main page (Figure 2.1(a)). The Cancel Reservation branch can be explained in a similar fashion. When the “Cancel Reservation” button is activated, the customer is led to a new page as shown in Figure 2.1(e). On this page, there is a single text input box: Confirmation #. After entering the required data and activating the “Cancel Reservation” button, the Cancellation Notification page is reached as shown in Figure 2.1(f). If the “Cancel Another Reservation” button is activated, the customer is led back to the page of Figure 2.1(e). The cancellation process can be terminated at any step by activating the “Back to Main Page” button, which transfers the customer to the main page (Figure 2.1(a)).

One common feature of web database applications such as OCRS is the frequent interaction between customers and an application database. An important component to testing this type of application is verifying whether the database has been correctly updated in response to the customers’ inputs. For instance, in the previous example, a customer provides required information through the GUI in Figure 2.1(b) and changes some data in Figure 2.1(c). After clicking the “Confirmation” button, the database should be updated.

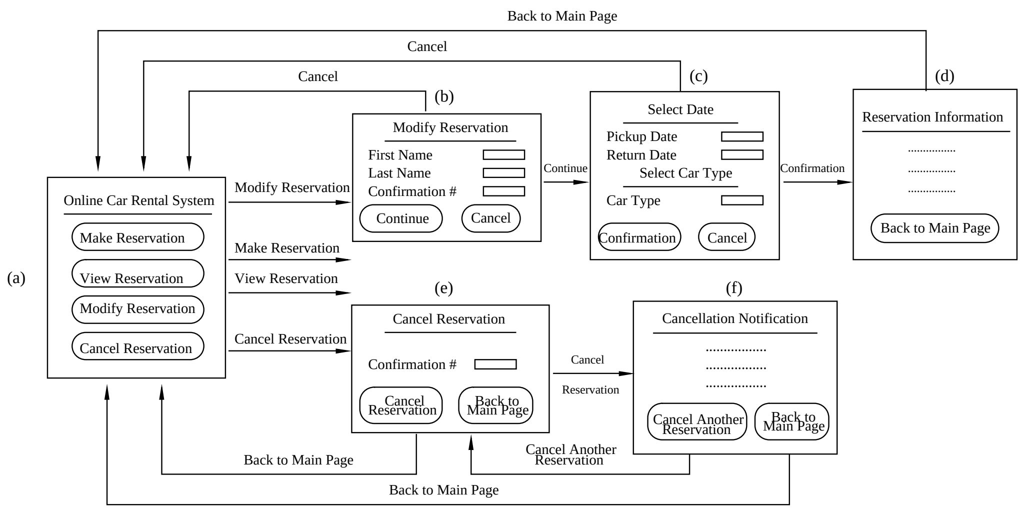


Figure 2.1: The online car rental system (OCRS)

Therefore, the application needs to be thoroughly tested to determine whether the database has been successfully updated.

Chapter 3

Framework of Problem

In this section, we give a formal description of the problem of testing web database application and a formal model of our solution.

3.1 Formal Description of Testing Web Application without Database

First let's abstractly consider a web application. A web application consists of a sequence of *transitions*. A transition starts from a source web page, upon an input, arrives at a destination page. The pages and the input can be simply seen as strings and thus the transitions of a web application can be abstractly considered as the following function T:

$$T(P_{src}, I) = P_{dst}\text{such that} \begin{cases} T_1(P_{src}, I), & \text{if } C_1(P_{src}, I) = true \\ \vdots & \vdots \\ T_n(P_{src}, I), & \text{if } C_n(P_{src}, I) = true \end{cases} \tag{3.1}$$

in which P_{src}, I, and P_{dst} are the source page, the input, and the destination page, respectively, and I is a sequence of *input variables* $V_0, \ldots, V_j$, each of which represents an input parameter of the input form. Each T_i $(1 \leq i \leq n)$ defines the page transition of transition i if the corresponding condition C_i is true. We assume that the transition is deterministic, hence exactly one C_i is true for a given input I. A web application can be modeled as the transitive closure, W, of the transition function, T, as follows.

$$W(P_{src}, I) = T(\ldots T(\ldots T(T(P_{src}, I_1), I_2) \ldots I_i) \ldots I_k) = P_{dst} \tag{3.2}$$

In specification-based black box testing, there are two functions: W and W'; W represents the specification function, while W' represents the implementation function. The goal of the testing is to test, starting from same web page P_{src} and given input I, whether the implementation function, W', produces the same destination web page, P_{dst}, as the function W. In testing, I is called a *test case*, while P_{dst} is the corresponding *oracle*.

Before testing begins, a test coverage criterion is given that describes the range of the testing based on the specification, i.e., which page sequence of the web application should be tested. For instance, according to a given test coverage criterion, the testing target is the page sequence $P_0, \ldots, P_m$. Based on the definition of the transition function T, we can determine a transition sequence, for instance, the transition sequence corresponding to the page sequence $P_0, \ldots, P_m$ is $T_0, \ldots, T_{m-1}$. In order to drive a test along the specified transition sequence, each $I_i (0 \leq i \leq m-1)$ of the input sequence I needs to be carefully designed to ensure the corresponding condition C_i true, so that each transition T_i can be exercised. Figure 3.1 shows the correspondence of pages between the specification and implementation functions for a page sequence. The testing starts at the same page, P_0. After applying an input sequence $I_0, \ldots, I_{i-1}$, the implementation function W' transfers the web application to a page P'_i. Upon the input I_i, which is designed to make the condition C_i true, function T_i is triggered and exercised, then the destination page P_{i+1} should be reached by the function T_i. The testing process is designed to check whether P'_{i+1} produced by the implementation function T'_i is the same as the page predicated by the specification P_{i+1}.

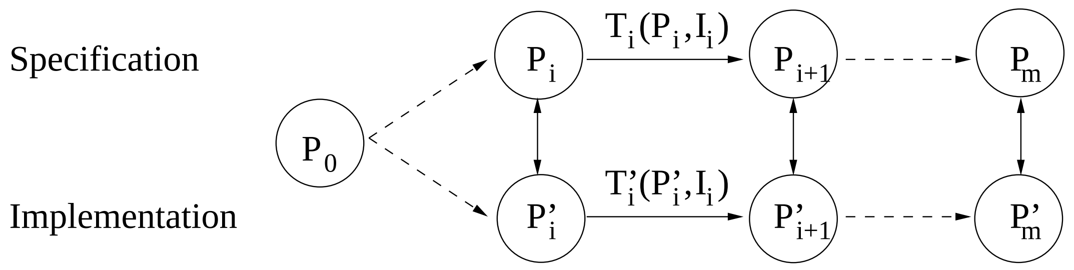


Figure 3.1: The correspondence of pages between the specification and implementation of page sequence $P_0, \ldots, P_m$

3.2 Formal Description of Testing Web Database Application

Most web applications have a back-end application database (AD) behind the web interface. The testing of web database applications is more complicated because the tests must include whether the database is in a correct state after a sequence of updates. The transitions of a web database application can be modeled as the following function U:

$$U((P_{src}, AD_{src}), I) = (P_{dst}, AD_{dst}) = \begin{cases} U_1(P_{src}, AD_{src}, I) & \text{if } C_1(P_{src}, AD_{src}, I) \\ \vdots & \vdots \\ U_n(P_{src}, AD_{src}, I) & \text{if } C_n(P_{src}, AD_{src}, I) \end{cases} \tag{3.3}$$

in which AD_{src} and AD_{dst} are the source and destination AD state respectively, and each U_i $(1 \leq i \leq n)$ defines both the page navigation and database update of transition i. U_i can be further defined as follows.

$$U_i((P_{src}, AD_{src}), I) = (Navigate_i((P_{src}, AD_{src}), I), Update_i(AD_{src}, I)) \quad (3.4)$$

in which $Navigate_i$ defines page navigation, while $Update_i$ defines how the database should be updated. A web database application can be represented as the transitive closure, WD, of the transition function, U, as follows:

$$WD((P_{src}, AD_{src}), I) = U(\ldots U(\ldots U(U((P_{src}, AD_{src}), I_1), I_2) \ldots I_i) \ldots I_k) = (P_{dst}, AD_{dst})$$

A web database application can be tested, whether the implementation function, WD', produces the same destination web page and database state, (P'_{dst}, AD'_{dst}), as the oracle, (P_{dst}, AD_{dst}), as the specification function WD, when started from the same page and database state.

In this paper, we only focus on testing the database state, i.e., whether each database update function $Update'_i$ $(1 \leq i \leq n)$ of the implementation function U'_i performs like the specification function $Update_i$, therefore, the following assumption is made: the P'_{dst} generated by $Navigate'_i$ is always the same as P_{dst} specified by $Navigate_i$, that is, $Navigate_i$ has been implemented correctly. Additionally, since the web interface is visible to the tester, we further assume that the source web page P_{src} can always be reached after each test.

Figure 3.2 shows the correspondence of database states between the specification and implementation of a page sequence $P_0, \ldots, P_m$. It is similar to Figure 3.1. The first row of Figure 3.2 shows the sequence of AD states $(AD_0, \ldots, AD_m)$ produced by specification *WD*, while the second row shows the corresponding AD' sequence $(AD'_0, \ldots, AD'_m)$ of the implementation. The testing starts at the same state, AD_0. Each state AD_{i+1} $(0 \leq i \leq m-1)$ is computed by applying the function Update_i on (AD_i, I_i). A test oracle tests whether the states AD_{i+1} and AD'_{i+1} are the same, for each pair of states.

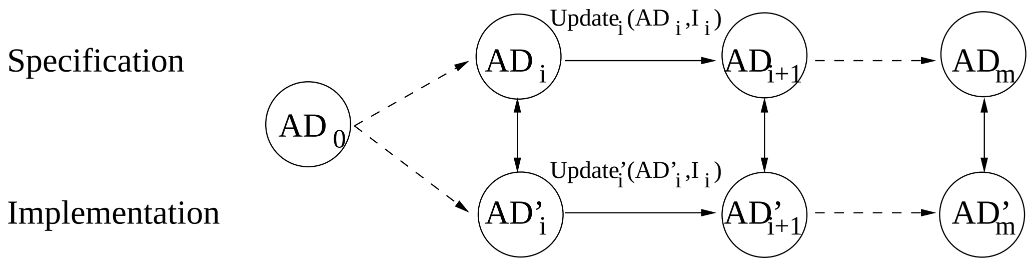


Figure 3.2: The correspondence of AD states between the specification and implementation of page sequence $P_0, \ldots, P_m$

In black box testing, the database state is invisible to the tester when running a test. Thus it's not possible to directly compare the state of AD'_i with the oracle AD_i. Furthermore, it is costly to instantiate AD_i. But partial testing is feasible. In partial testing, instead of trying to guess a precise instantiation of the AD'_i, we can use one or a sequence of queries to detect whether AD'_i

satisfies some correctness properties, which we call a *partial oracles.* A partial oracle can be specified by the tester based on which properties are meant to test. For instance, in this paper, we focus on testing whether the database update operations are correctly performed. Accordingly, each partial oracle should be specified in a way that reflects the result of an update.

The input sequence needs to be carefully designed in order to drive the testing process in a direction specified by a given test coverage criterion. However, the sources of the input for web database application are AD when existing data is required, and also new data which is different than AD. We use the synthetic database (SD) to store the new data. The SD and AD are disjoint, that is they have different tuples, but exactly the same schema. Additionally, since the input data of one transition often depends on the data of a previous transition. If we generate the input data for transitions separately, the input data for a previous transition might or might not produce a "good" AD for the generation of the next input data. For instance, in the OCRS example, if we separate the First Name, Last Name and Confirmation # of Modify Reservation path into different transitions, we need to make sure that the First Name and Last Name generated for the previous transition results in an associated Confirmation # for a later transition. Therefore, in order to maintain the data dependency among the input data, we need to generate the entire input sequence I as a whole. Moreover, since along the transition sequence, the database sources (AD and SD) might be updated, the generation of each input data $I_i = (V_i^0, \ldots, V_i^i)$ can be seen as a function G_i of a particular AD and SD version, and the generation of the entire input sequence $I_0, \ldots, I_n$ for a particular page sequence should be a function of following form.

$$I_0, \ldots, I_n \text{such that} \begin{cases} I_0 = (V_0^0, \ldots, V_0^a) = G_0(V_0^1, \ldots, V_0^a, AD_0, SD_0) \\ \vdots \\ I_n = (V_n^0, \ldots, V_n^z) = G_n(V_n^1, \ldots, V_n^e, AD_n, SD_n) \end{cases} \tag{3.5}$$

where each $G_i(0 \leq i \leq n)$ defines how the input data of I_i should be generated. A data dependency is maintained by sharing the same variable name, i.e., V_i^2, in I_i is inherited by I_j. Each AD_i can be derived by applying the function $Update_{i-1}$ on (AD_{i-1}, I_{i-1}), while each SD_i can be updated by simply deleting whatever data has been used as an input in the previous transitions.

Another issue that makes the black box testing of the database state different than the traditional black box testing is that it's not easy to bring the database state back to the initial state AD_0 after a test, or even worse, the initial state is also invisible in the sense that the tester doesn't know exactly what data is in there and whether there is enough data to perform a certain test. Traditional black box testing always assumes that the testing process starts in a known initial state, so that all the tests are independent and will be always valid if a bunch of tests are generated based on the initial state in advance. However, in practice, keeping a copy of the initial database state for each test is too expensive, moreover it's not easy to trace back to the initial state after a test. Therefore, new approaches are needed to solve this problem. In this paper, we propose two approaches which complement each other in the generation of the test data.

3.2.1 Dirty Test Suite Approach

In this approach, the entire test suite consists of many test cases generated in advance. Each test case is generated based on the initial state (AD_0). If we kept a copy of the AD_0 for running each test case, the test suite would be valid at any time. However, it's too expensive to do so in practice. A practical solution is continuing to run later test cases on the database, i.e., AD_i left over from the previous tests. But the later test cases which are generated based on the AD_0 might not be valid any more based on the AD_i since the AD_0 might have been updated by the previous tests, so that the AD_i is no longer the same as the AD_0. In order to ensure a valid test case for each test, before running a test case, a *filter query* is evaluated based on the current state of the database to check whether this particular preset test case is still valid. If the query succeeds, the test case is run based on the current database, otherwise, the test case is queued for later evaluation.

3.2.2 Clean Test Case Approach

Another approach, which is investigated in more detail in this paper, is to generate valid test cases on the fly (during the testing process). This approach is different from the dirty test suite approach in two ways. First, all the test cases are generated on the fly during the testing process rather than having the entire test suite generated in advance. Second, since the test cases are always generated based on the most current version of the AD, all the test cases generated are valid during the testing process.

3.2.3 Tradeoff between these two Approaches

The dirty suite approach is more efficient when there is no database updates, or the database updates only influence a narrow range of the data in the AD. It's efficient in the sense that the whole test suite is generated in advance, and there is no overhead on generating the tests at run time (though filter queries must still be evaluated). Moreover, the entire test suite can be reused to retest the same application. However, if the database has been influenced too much by the previous tests and thus the filter query has very low percentage of success, this approach won't work well, and in the worst case, only the first test case is valid.

Compared with the dirty suite approach, the clean test case approach is less efficient but is guaranteed to make progress. Test cases are generated during the testing process, so the testing process might take longer than the dirty suite approach. However, every test case generated in this approach is valid, so the approach works no matter how of the database has been updated. In this paper, we design a framework of the clean test case approach.

Based on the tradeoff of the two approaches, we can use both to complement each other in testing one web application by using the dirty test suite approach to test those page sequences involving few database updates, while applying the second approach to test those with more intensive database updates.

3.3 One Possible Solution of Testing Web Database Application

For a web application function W, the output (P_{dst}) of one relation might be a source page (P_{src}) of another relation. We have found that FSMs are a suitable tool for characterizing the transitive behaviors of the web application function. Finite state machines (FSM) provide a convenient way to model software behavior in a way that avoids issues associated with the implementation. Several methods for deriving tests from FSMs have also been proposed [8, 9, 19]. Theoretically, web applications can be completely modeled with FSMs, however, even simple web pages can suffer from the state space explosion problem. There can be a large variety of possible inputs to text fields, a large number of options on some web pages, and choices as to the order in which information can be entered. Factors such as these mean that a finite state machine can become prohibitively large, even for a single page. Thus, an FSM-based testing method can only be used if techniques are found to generate FSMs that are descriptive enough to yield effective tests yet small enough to be practically useful.

The technique in [1], FSMWeb, addresses the state explosion problem with a hierarchical collection of aggregated FSMs. The bottom level FSMs are formed from web pages and parts of web pages called logical web pages, and the top level FSM represents the entire web application. Application level tests are formed by combining test sequences from lower-level FSMs.

Our approach extends FSMWeb to include testing of the application database. We introduce a Database Extended FSM (DEFSM) to model the behavior of a web database application as follows.

Definition 1 (DEFSM) *The DEFSM D is a six-tuple* $D\langle S, s_0, \mathrm{ADs}, \mathrm{AD}_0, V, T\rangle$

where,

- S *is a finite set of web states which correspond to the logical web pages.*

- s_0 *is the initial web state or logical web page.*

- ADs *is a finite set of the associated back-end application database (*AD*) states. The schemas of all elements of the* ADs *are the same:* AD *(*$R_1, \ldots, R_n$*), where* $R_1, \ldots, R_n$ *are relations in the* AD*. The schema of relation* R_i *is the form:* $R_i(A_1, \ldots, A_m)$ *where* A_j*,* $1 \leq j \leq m$ *denotes the* j^{th} *attribute of relation* R_i*. However the state (instantiation) of each element of the* ADs *is unique.*

- AD_0 *is the initial* AD *state.*

- V *is a set of variables which denote all the possible web page input parameters.*

- T *is a set of state transitions, each element* t *is a quadruple* $\langle S_{src}, S_{dst}, P(V_t), \text{Update}_t \rangle$*.*

  - S_{src}*,* $S_{dst} \in S$*, they are the source and destination web states of transition* t *respectively;*

  - $V_t = (V_t^0, \ldots, V_t^j)$ *(*$V_t^i \in V, 0 \leq i \leq j$*) is the list of input variables of transition* t*. If some variable* V_t^i *has had a value, then the value is* *inherited. $P(V_t)$ is a predicate expressed in terms of V_t that defines the evaluation of the variables of V_t. In our approach, $P(V_t)$ is defined as a Prolog rule in Section 4.2.2.*

– Update_t *consists of a sequence of database update operations including insertions and deletions. It's defined as Prolog rules in Section 4.2.2.*

At state S_{src}, for an instantiation of V_t, if $P(V_t)$ evaluates true, the web state changes to S_{dst}, while the database state is transferred from AD_{src} *to* AD_{dst} *(*AD_{src}, $\text{AD}_{dst} \in \text{AD}_s$*) in response to the* Update_t*. If there is no update,* $\text{AD}_{src} = \text{AD}_{dst}$*.*

∎

Given a specification DEFSM, D, and a test coverage criterion (which dictates the range of the testing), a set of test sequences is generated, each of which is a sequence of transitions under test, $TS = (t_1, \ldots, t_n)$ $(t_i \in T, 1 \leq i \leq n)$.

Definition 2 (Test Case) *A test case is an instantiation of the set of input variables ($V_{t_1} \cup \ldots \cup V_{t_n}$) of a test sequence ($t_1, \ldots, t_n$).*

∎

Chapter 4

The AutoDBT Framework

This section describes the AutoDBT framework for testing a web database application. We present an overview of the architecture first. Then, each component in the architecture is described.

4.1 Overview

AutoDBT is a framework for testing web database applications. The framework is depicted in Figure 4.1. AutoDBT has three main steps. The first step is to specify the expected behavior of the application as a DEFSM. In this step, a modeler develops a DEFSM specification for the web application. As shown inside the top-most component of Figure 4.1, the DEFSM consists of two parts: a *state transition diagram* and a *data specification.* The state transition diagram is a directed graph that models the user navigation between forms in the interface (see Figure 4.2). Each edge in the graph is labelled with a unique transition number. The data specification articulates input constraints and database updates associated with each transition in the state transition diagram. In the second step the Test Sequence Generator automatically generates a set of *test sequences*. A test sequence traces a path in the DEFSM. The *test coverage criteria* dictate the range of test sequences that are generated. Meanwhile, a Dynamic Data Specification Generator automatically generates a *dynamic data specification* based on the data specification given in the first step. The dynamic data specification captures how the application database is updated during evaluation of a test. The third step performs the testing process which takes as input the dynamic data specification, the generated test sequences, data sources and test data selection criteria, and generates a report about the test result. The testing process is described in detail in Section 5. The rest of this section illustrates the framework in detail using the OCRS example.

4.2 DEFSM Modeling

The first step of using AutoDBT is to model the page navigation with a DEFSM. Part of the DEFSM is a state transition diagram that captures the

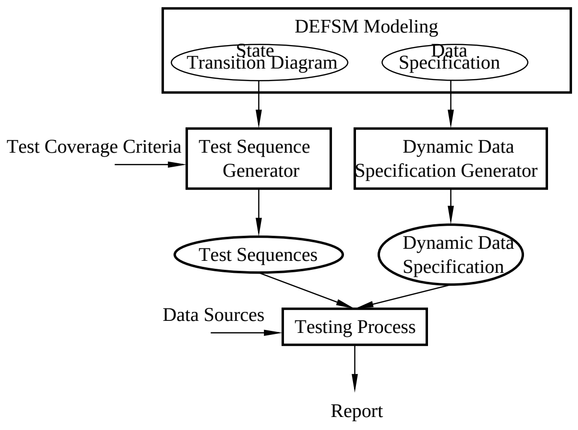


Figure 4.1: AutoDBT's framework

expected behavior of a given web application at an abstract level. We assume that the modeler who builds the DEFSM has a thorough knowledge about the requirements and behavior of the application.

4.2.1 State Transition Diagram

The state transition diagram is a directed graph, in which each node represents a (logical) web page and each edge represents a possible page transition (button or hyperlink). For the OCRS example, the diagram is given in Figure 4.2. In this paper we focus only on the Modify Reservation and Cancel Reservation paths, so in Figure 4.2, only those paths are drawn in detail. According to this diagram, the web application begins in the Main state. There are four paths that emanate from the Main state. Through transition 2, the

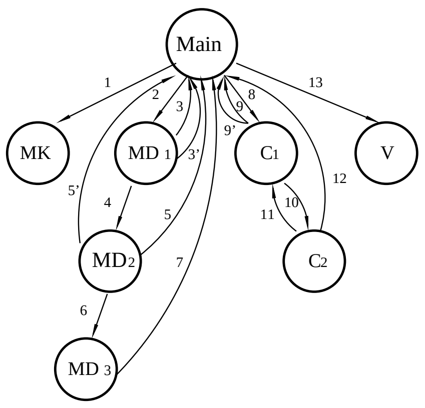


Figure 4.2: OCRS's state transition diagram

application is transferred first to state (MD_1) of the Modify Reservation path, then through transition 4 to state MD_2, and finally through transition 6 to state MD_3. At each of these three states (MD_1 through MD_3), the application can be transferred back to the original Main state by transition $3, 3', 5, 5'$, or 7, respectively. The transitions of other functional paths can be similarly explained. OCRS is a relatively simple application so a single state transition diagram suffices. For complex applications, we allow the transition diagrams to be "nested" [3]. That is, simple FSMs can be nested as states in other FSMs as described in detail elsewhere [1].

4.2.2 Data Specification

A modeler also has to specify how data flows in the application, especially between the application and its database. So associated with each transition of the DEFSM, the modeler gives constraints on the input (the *input specification*) and sketches the correctness criteria for the output. Since the output correctness criteria are based on how the database should be updated, the expected updates are modeled as well (the *update specification*). From among the many potential approaches to giving data specifications, AutoDBT adopts a declarative approach, in which database updates and input constraints are expressed in *Prolog*. We chose Prolog because it offers a well-defined, declarative semantics for expressing database queries. We use Prolog rather than Datalog because we generally need to evaluate our queries using a top-down, tuple-at-a-time evaluation technique, i.e., using Prolog's evaluation model. In the rest of this section, we illustrate how the input and output are specified in Prolog through the OCRS example.

Web applications have many different kinds of input widgets, such as drop down menus. For simplicity, this paper focuses only on text boxes and buttons.

The input specification consists of two related parts. The first part is to specify the source of the test data. For a web database application, the input data can be drawn from two sources: the application database (AD) and the synthetic database (SD). The SD contains data that is not in the AD. For instance, in a test of the page in Figure 2.1(b), the customer's last name, first name and confirmation number should be drawn from the AD since the data should already exist in the application database. However, in a test of Figure 2.1(c) some "new" data, which is not resident in the AD, is needed when the customer changes an existing reservation. The new data is chosen from the SD.

We assume that the schemas of the both data sources are known to the modeler. If the modeler doesn't know the schema of the AD, a mapping between the AD and AutoDBT's AD has to be provided before the testing process starts. We further assume that the schema of the SD is the same as that of the AD. Later on, we'll show how this assumption can be relaxed.

The second part of the input specification captures the data flow on transitions in the DEFSM.

Definition 3 (Input data flow specification) *The input data flow specification for transition i is either a button name or a Prolog rule of the following form followed by a button name:*

$$\text{input}_i(V_{i^1}, \ldots, V_{i^n}) \text{ :- } \text{Predicate}_1, \ldots, \text{Predicate}_m.$$

where

- input_i *is a predicate with a list of variables* $(V_{i^1}, \ldots, V_{i^n})$ *denoting all of the required input parameters of transition i; and*

- $\text{Predicate}_1, \ldots, \text{Predicate}_m$ *is a list of predicates of the following form:*

$$\text{Database\_Relation}(A_1, \ldots, A_k);$$

or

$$\text{input}_j(V_{j^1}, \ldots, V_{j^k}).$$

where

  - Database $\in \{AD,\ SD\}$;
  - Relation $\in \{R_1,\ \ldots,\ R_n\}$;
  - $A_i \in \{constant,\ variable,\ \text{'\_'}\}$ *(1* $\leq i \leq$ *k), where constant* $\in$ *domain of the* i^{th} *column; and*
  - *the rule is safe which means all variables in the head appear in some predicate in the body.*

∎

To help explain the input data flow specification, consider the OCRS example. Table 4.1 shows the specification for transitions of the Modify Reservation path and the Cancel Reservation path in Figure 4.2. According to the

Transition	**Input Data Flow Specification**
2	Button(Modify Reservation).
3	$input_3$(*Fn*, *Ln*, *C#*) :- AD_Customer(*Fn*, *Ln*, *Cid*), AD_Reserves(*Cid*, *C#*). Button(Cancel).
$3'$	$input_{3'}$(*Fn*, *Ln*, *C#*) :- SD_Customer(*Fn*, *Ln*, *Cid*), SD_Reserves(*Cid*, *C#*). Button(Cancel).
4	$input_4$(*Fn*, *Ln*, *C#*) :- AD_Customer(*Fn*, *Ln*, *Cid*), AD_Reserves(*Cid*, *C#*). Button(Continue).
5	$input_5$(*Pd*, *Rd*, *Ct*) :- AD_Reservation(_, *Pd*, *Rd*, *Ct*). Button(Cancel).
$5'$	$input_{5'}$(*Pd*, *Rd*, *Ct*) :- SD_Reservation(_, *Pd*, *Rd*, *Ct*). Button(Cancel).
6	$input_6$(*Pd*, *Rd*, *Ct*) :- SD_Reservation(_, *Pd*, *Rd*, *Ct*). Button(Confirmation).
7	Button(Back to Main Page).
8	Button(Cancel Reservation).
9	$input_9$(*C#*) :- AD_Reservation(*C#*, _, _, _). Button(Back to Main Page).
$9'$	$input_{9'}$(*C#*) :- SD_Reservation(*C#*, _, _, _). Button(Back to Main Page).
10	$input_{10}$(*C#*) :- AD_Reservation(*C#*, _, _, _). Button(Back to Main Page).
11	Button(Cancel Another Reservation).
12	Button(Back to Main Page).

Table 4.1: Input data flow specification for the Modify Reservation and Cancel Reservation paths

specification, on transition 2, a "Modify Reservation" button is required. On transition 4, the customer's first and last names, and the confirmation number of an existing reservation are required before a "Continue" button is activated. The DEFSM modeler uses *Fn*, *Ln*, *C#* to denote the required input. The *Fn* and *Ln* are chosen from the Customer relation while *C#* comes from the Reserves relation of the AD. The meaning of the input data flow specification for the other transitions can be similarly explained.

For some applications the schema of the SD can be simplified. The relations of the AD provide input data and also participate all the functionality of the application. But the relations of the SD are only useful for generating the input data. Relations that are not used in the SD can be removed from the SD's schema. For instance, the Available_Car_Type relation name does not appear in Table 4.1, which means that this relation is not useful in defining the input data. Therefore it's safe to delete it from SD's schema.

Definition 4 (Simplified SD schema) *The simplified SD schema is a subset of the original SD schema:*

$$SD(R'_1, \ldots, R'_k);$$

where $\{R'_1, \ldots, R'_k\} \subseteq \{R_1, \ldots, R_n\}$*, and each* R'_i*,* $1 \leq i \leq k$*, appears at least once in the body of input specification.* ∎

As for the OCRS example, the Available_Car_Type relation schema is deleted from the SD schema and Table 4.2 shows the simplified schema diagram of the SD.

Before the testing process commences, the AD and SD need to be populated. Initially the SD is empty, while the AD might already contain some data (if the web application has been running prior to testing). Since the testing process will change values in the AD, the AD should be copied prior to

Customer		
FirstName	LastName	CustomerID

Reserves	
CustomerID	Confirmation#

Reservation			
Confirmation#	PickupDate	ReturnDate	CarType

Table 4.2: Simplified schema diagram of the SD

testing and the copy used for testing. The SD will need to be populated with *synthetic data*, that is, data generated strictly for testing purposes. Gray et al. present several techniques to populate a database with synthetic data [10]. In particular they show how to quickly generate, in parallel, a large database that obeys certain statistical properties among the records generated. Using their techniques we can populate the SD (and the AD if needed) with synthetic data. As for the OCRS example, the populated AD and SD are shown in Table 4.3 and Table 4.4, respectively. However, it's not clear, in [10], how referential integrity are maintained. Theoretically, it shouldn't be a difficult problem to solve. Thus it's reasonable to make the following assumption: there is a perfect tool to populate the AD and SD with synthetic data which satisfies all the integrity constraints.

In addition to the input specification, the modeler also needs to specify how the database should be updated. This specification is used to evaluate whether the application correctly updates the AD. There are three kinds of

Customer		
FirstName	LastName	CustomerID
john	smith	c0001
mike	green	c0002
rick	reed	c0003
kate	brown	c0004

Reserves	
CustomerID	Confirmation#
c0001	0001
c0002	0002
c0003	0003
c0004	0004
c0004	0005

Reservation			
Confirmation#	PickupDate	ReturnDate	CarType
0001	10/01/03	10/03/03	economy
0002	10/02/03	10/05/03	compact
0003	10/15/10	10/23/03	full size
0004	11/03/03	11/30/03	minivan
0005	11/03/03	11/30/03	full size

Available_Car_Type	
CarType	CarNumber
economy	50
compact	40
full size	60
minivan	45
luxury	20
convertible	30

Table 4.3: An example application database (AD)

updates: insertion, deletion, and modification. We treat a modification as a deletion followed by an insertion. To model deletions from AD_*Relation*, we add a relation, delete_AD_*Relation*, that is used to store tuples that should be deleted. For insertions, we introduce insert_AD_*Relation* which buffers tuples that should be inserted. The schema of each relation is the same as that of AD_*Relation*.

Customer		
FirstName	LastName	CustomerID
franklin	bond	c0006
alicia	wong	c0007

Reserves	
CustomerID	Confirmation#
c0006	0006
c0007	0007

Reservation			
Confirmation#	PickupDate	ReturnDate	CarType
0006	12/04/03	12/10/03	luxury
0007	12/06/03	12/08/03	convertible

Table 4.4: An example synthetic database (SD)

In the update specification, the modeler gives a specification of what should be deleted or inserted during an update.

Definition 5 (Update specification) *The update specification is one or more Prolog rules of the following two forms.*

*1) delete_AD_*Relation*($A_1, \ldots, A_n$) :-*

$$\text{Predicate}_1, \ldots, \text{Predicate}_m.$$

*2) insert_AD_*Relation*($A_1, \ldots, A_n$) :-*

$$\text{Predicate}_1, \ldots, \text{Predicate}_m.$$

The form of each Predicate$_i$ *is given in Definition 3.* ∎

Table 4.5 shows the update specification for transitions of the Modify Reservation and Cancel Reservation paths. There is no update associated with

Transition	Update Specification
6	delete_AD_Reservation($C\#$, Pd, Rd, Ct) :- $input_4$(_, _, $C\#$), AD_Reservation($C\#$, Pd, Rd, Ct). insert_AD_Reservation($C\#$, Pd, Rd, Ct) :- $input_4$(_, _, $C\#$), $input_6$(Pd, Rd, Ct).
10	delete_AD_Reservation($C\#$, Pd, Rd, Ct) :- $input_{10}$($C\#$), AD_Reservation($C\#$, Pd, Rd, Ct). delete_AD_Reserves(Cid, $C\#$) :- $input_{10}$($C\#$), AD_Reserves(Cid, $C\#$).

Table 4.5: The update specification for the Modify and Cancel Reservation paths

most of the transitions so only a few of the transitions have update specifications. Transition 6 modifies the AD while transition 10 involves a deletion. The modification is modeled as a deletion followed by an insertion, so two rules are associated with transition 6. Transition 10 deletes a tuple from the AD_Reservation relation. In order to maintain referential integrity, the corresponding tuple which has the same $C\#$ in the AD_Reserves relation has to be deleted as well.

Finally, we should note that the modeler has to be careful when developing an update specification to associate updates only with transitions that reflect transaction commit points. In our example the database is updated right after each update transition, but in general the update could be delayed. Many web database applications are designed to support concurrent users. For instance, in the OCRS, many users can simultaneously make car reservations, and the reservation process might extend over several transitions. An application often packages the work done by a sequence of forms into a single *transaction*. A transaction is a logical unit of database processing that includes one or more database access operations, and it executes on an all-or-none basis. The effect of the transaction won't be permanently recorded in the database until the transaction commits. Since the database state won't be updated until the transaction commits (which might span over multiple transitions), oracles should be evaluated until the commit point. Although the oracles can be generated automatically based on the update specification, the modeler needs to account for transactions and should only associate updates with transitions in which a transaction commits.

4.3 Test Sequence Generator

Based on the state transition diagram, for any given test coverage criteria, we can automatically generate a *test sequence*. A test sequence is a sequence of transitions. It describes which transitions need to be tested and in what order. Common kinds of coverage criteria include testing combinations of transitions (switch cover) [8], testing most likely paths [23], and random walks [17]. Since the state transition diagram is a directed graph, well-known graph theory algorithms can be applied to efficiently generate the test sequences automatically. One algorithm that can be employed solves the New York Street Sweeper Problem [4] by generating a minimal length tour (which is a closed tour covering each link) for a directed graph [5]. This algorithm can be applied to the testing problem for generating a minimal length test sequence which starts and ends at an initial state and covers each transition of the testing target. For instance, after applying this algorithm to the Modify Reservation path, the following test sequence is generated.

$$2 \rightarrow 3 \rightarrow 2 \rightarrow 3' \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 2 \rightarrow 4 \rightarrow 5' \rightarrow 2 \rightarrow 4 \rightarrow 6 \rightarrow 7$$

4.4 Dynamic Data Specification Generator

A test sequence is a sequence of transitions while a *test case* is an instantiation of the input parameters for those transitions, i.e., it contains all of the data necessary to perform the test. Since the state of the database is dynamic, the test case generation process needs to model each database version. The input specification stipulates which database is used in choosing the test data, while the update specification models the expected dynamic behavior in response to the input data. In this section we describe how AutoDBT combines the input specification and the update specification to produce a *dynamic data specification* that captures this dynamic behavior. Before illustrating how to generate the dynamic data specification, we need to first to introduce an important data structure which will be used to capture the dynamic behaviors.

4.4.1 Update List

In our approach, the dynamic behavior is captured by a Prolog data structure: *List*. Each version of the AD is associated with one **update list** consisting of a buffer of deleted tuples and a buffer of inserted tuples. The update list for the application database, AUL, is represented as follows: [$ADel$, $AIns$] where $ADel$ is the delete buffer and $AIns$ is the insert buffer. The SD is simpler since data is only deleted but never inserted. One tuple is deleted from the SD whenever it's used (so that subsequent tests can't reselect the data, thereby ensuring that the data in the AD and SD are disjoint). The update list for SD (SUL) is a buffer of deleted tuples. Initially the AD/SD buffers are empty (denoted as [[],[]]/[]). For the AD, after each update transaction, the buffers are managed to capture the effects of a modification as follows. When a tuple is deleted, the tuple is inserted into the $ADel$ buffer, unless it appears in the $AIns$ buffer (as a result of a previous insertion in the evaluation of the test case) in which case it is simply removed from $AIns$ buffer. Insertion is similar. If the update transaction contains more than one transition, then the update list won't be changed until the transaction commits. Based on the update specification given by the modeler, the updated AD versions can always be derived from the initial version and the associated list, and each updated relation can be derived by applying the following Prolog rules.

AD_$Relation(A_1, \ldots, A_k, AUL)$:-

  AD_$Relation(A_1, \ldots, A_k)$,

  notdeleted($Relation(A_1, \ldots, A_k), AUL$).

AD_$Relation(A_1, \ldots, A_k, AUL)$:-

  inserted($Relation(A_1, \ldots, A_k), AUL$).

notdeleted(*Fact*, AUL) :-

  $AUL = [ADel, AIns]$,

  $\neg$ member(*Fact*, $ADel$).

inserted(*Fact*, AUL) :-

  $AUL = [ADel, AIns]$,

  member(*Fact*, $ADel$).

The first rule says that a particular version of an AD_*Relation* with an associated update list (AUL) can be derived by including all the tuples that existed in the initial version and have yet to be deleted. Those facts can be checked by the third rule that screens out the tuples that are members of the *ADel* list. The second rule adds all of the tuples that have been inserted in AD_*Relation*, or have been re-inserted after being deleted. The fourth rule that checks whether a particular tuple is a member of the *AIns* list.

Similarly, the *Relation* of updated SD versions can be derived by applying the following rule.

$$\begin{aligned}
&\text{SD\_}\mathit{Relation}(A_1, \ldots, A_k,\ \mathit{SUL}) \text{ :-} \\
&\qquad\qquad \text{SD\_}\mathit{Relation}(A_1, \ldots, A_k), \\
&\qquad\qquad \neg\ \text{member}(\mathit{Relation}(A_1, \ldots, A_k), \mathit{SUL}).
\end{aligned}$$

The dynamic data specification generator takes in both the input and update specification, and generates a dynamic data specification which consists of two components: a *dynamic input specification* (DIS) and a *data inheritance graph* (DIG). The DIS expresses both how the input data should be generated, and also how new versions of the AD and SD are generated as the database is updated. The DIG captures how data flows from one transition into another. In the following two sections, we outline how the two components are generated.

4.4.2 Data Inheritance Graph Generation

As a test navigates from page to page, input for a form in one transition may have to appear in an update performed in a much later transition or as a defining atom for a later input. For instance, the update rule associated with transition 6 in Table 4.5 uses a confirmation number, $C\#$, from the input_4 fact produced in the evaluation of the input rule associated with transition 4 in Table 4.1. The DIG is a collection of these dependencies. The graph can be automatically generated by analyzing the input and update Specifications. In the OCRS example, the graph consists of nodes for transitions 4 and 6 and one edge from 4 to 6, labelled with *C#*.

4.4.3 Dynamic Input Specification Generation

The DIS specifies how new versions of the database are created in response to user inputs and database updates. The DIS is generated by merging the input and update specifications. One rule is added to the DIS for each rule in the input specification. The DIS rule is constructed in two steps as follows.

1) **Generate the head** The head of the DIS rule is derived from the head of the input rule by modifying it to include the current AD version (denoted as AD_{in}) and the next version of the AD (denoted as AD_{out}). For the SD, SD_{in} and SD_{out} are used. In addition, for every variable inherited by this transition in the DIG, add the variable to the head. For example, the head of the input rule for transition 6 is $input_6(Pd, Rd, Ct)$ and the variable *C#* is inherited by transition 6, so the corresponding head of the DIS rule for transition 6 should be $input_6(Pd, Rd, Ct, C\#, AD_{in}, AD_{out}, SD_{in}, SD_{out})$.

2) **Generate the body** If the transition has no associated update specification, then the input version of each database is the same as the output version. So add AD_{in} (SD_{in}) to each predicate prefixed by AD (SD). Also

add one more atom: $AD_{in} = AD_{out}$. But if a transition updates the AD, then there are four steps to creating the body.

1. Append the body of the update rule to the body of the input rule.

2. Remove *input* predicates from the body. For example, in transition 6, $input_4(\_, \_, C\#)$ and $input_6(Pd, Rd, Ct)$ are removed from the body.

3. Add a variable AD_{in} (SD_{in}) to each body predicate prefixed by AD (SD).

4. Append a AD *update predicate* of the following form to the body depending on whether the head of the update rule is delete_AD_$Relation(x_1, \ldots, x_n)$, or insert_AD_$Relation(x_1, \ldots, x_n)$.

   - $delete\_AD(Relation(x_1, \ldots, x_n), AD_{in}, AD_{out})$
   - $insert\_AD(Relation(x_1, \ldots, x_n), AD_{in}, AD_{out})$

   If the SD prefixes any body predicate of the input rule, such as SD_$Relation(x_1, \ldots, x_n)$, then SD *update predicate* of the following form

   - $delete\_SD(Relation(x_1, \ldots, x_n), SD_{in}, SD_{out})$

   is appended to the body.

   The database update predicates are defined by the following Prolog rules, respectively.

```
delete_AD(Fact, $AUL_{in}$, $AUL_{out}$) :-
        $AUL_{in}$ = [ADel, AIns],
        append(ADel, [Fact], ADel'),
        delete(AIns, Fact, AIns'),
        $AUL_{out}$ = [ADel', AIns'].
insert_AD(Fact, $AUL_{in}$, $AUL_{out}$) :-
        $AUL_{in}$ = [ADel, AIns],
        append(AIns, [Fact], AIns'),
        $AUL_{out}$ = [ADel, AIns'].
delete_SD(Fact, $SUL_{in}$, $SUL_{out}$) :-
        append($SUL_{in}$, [Fact], $SUL_{out}$).
```

The first three steps are straightforward. The final step requires modifying a rule based on a simple pattern matching technique on the head of an update rule. The final step is complicated by the fact that for efficiency, we use two buffers to store only the tuples that are deleted and inserted in each version rather than the entire version. The **delete buffer** consists of a list of deleted tuples and the **insert buffer** consists of a list of inserted tuples. So the final step is to include a predicate that inserts or deletes a tuple from the specified buffer, thereby generating the next version of the database.

As an example, the DIS rule for transition 6 is given below.

$input_6(Pd, Rd, Ct, \boldsymbol{C\#}, AD_{in}, AD_{out}, SD_{in}, SD_{out})$:-

SD_Reservation(_, Pd, Rd, Ct, SD_{in}),

AD_Reservation($C\#$, Pd', Rd', Ct', AD_{in}),

$delete\_AD$(Reservation($C\#$, Pd', Rd', Ct'), AD_{in}, AD_t),

$insert\_AD$(Reservation($C\#$, Pd, Rd, Ct), AD_t, AD_{out}),

$delete\_SD$(Reservation(_, Pd, Rd, Ct), SD_{in}, SD_{out}).

4.5 Summary

The first step to using AutoDBT is to construct a model of the web application to be tested. The model consists of an DEFSM and a data specification. The data specification is a high-level description of the data flow in the application. AutoDBT automatically generates a dynamic data specification, which is a low-level, precise description of the data flow. The second step to using AutoDBT is to decide on test coverage criteria. The criteria are input to the Test Sequence Generator to generate a list of test sequences. Each test sequence is a list of DEFSM transitions. The next step is to generate and run individual test cases as described in detail in the next section.

Chapter 5

The Testing Process

Figure 5.1 diagrams the testing process. The testing process starts with a *test sequence scheduler*. The scheduler schedules all of the test sequences, forming a queue of test sequences. Next, a test sequence is chosen from the queue and a *guard* is generated. The guard checks whether a test sequence can generate a *test case* given the current AD and SD states. A test case is an instantiation of the input parameters for an entire test sequence. If the guard fails, then the current AD and SD states can't build a test case for the entire test sequence, and the test sequence is placed at the end of the queue. Possibly, a future database state will be conducive to generating the test case. If the guard succeeds the test case is generated, as well as oracles to determine whether the test succeeds or fails. The test case is subsequently evaluated on the web application. During evaluation of the test case, an oracle is consulted after each transaction. If the oracle fails then the database was updated incorrectly, and the testing process aborts with a failure message. Finally, some test sequences involve more than one test case, so if more tests are needed for this particular test sequence, then the guard will be re-evaluated to generate more test cases. The process completes when the queue of test sequences becomes empty or the guard fails for every test sequence in the queue. The following sections elaborate upon each component in the testing process.

5.1 Test Sequence Scheduler

In testing a web database application, the testing process is influenced by cumulative effect of previous tests. The testing process of one test sequence can modify the database state arbitrarily, consequently each test starts with the database left over from the previous test. Since the database state is not visible during the testing process, there is no way we can trace back to the original state after running a test. Keeping a copy of the original database state for each test sequence is not a practical solution either since it's too expensive to maintain multiple copies of a database.

In this paper we propose an approach that mitigates the impact of prior tests. In our approach, a *test sequence scheduler* delays test sequences that

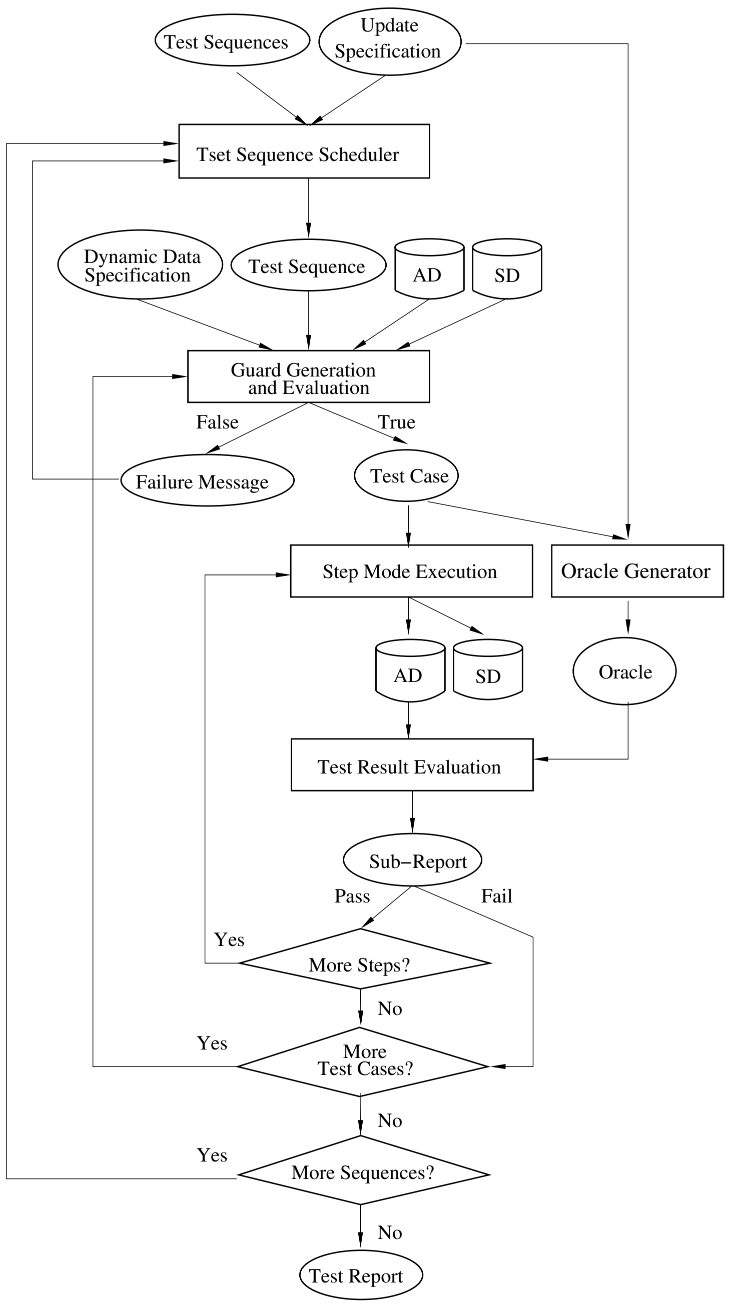


Figure 5.1: Framework of the testing process

delete data. More precisely, based on the update specification, all the test sequences are classified into four groups and scheduled in the following order: read-only, insertion-only, mixed insertion and deletion, and deletion-only. Within a group the test sequence order is random. For example, in OCRS, suppose that we need to test all four of the paths. One test sequence will be generated for each path. The four generated test sequences are classified as follows.

1. read-only: View Reservation
2. insertion-only: Make Reservation
3. mixed insertion and deletion: Modify Reservation
4. deletion-only: Cancel Reservation

Then the four test sequences should be scheduled in the order listed above.

5.2 Guard Generation and Evaluation

A guard is a query to determine whether a test sequence can be instantiated to produce a test case. A guard is automatically constructed for a test sequence by concatenating the head of each DIS rule corresponding to a transition in the sequence. In the following, we explain how to form a guard through an example.

Consider the following test sequence for Modify Reservation path.

2→3→2→3'→2 →4→5→2→4→5' →2→4→6→7

Ignoring the button inputs, the guard for this given test sequence is given below.

?- $input_3(Fn_1, Ln_1, C\#_1,\ AD_{in},\ AD_1),$

$input_{3'}(Fn_2, Ln_2, C\#_2,\ SD_{in},\ SD_1),$

$input_4(Fn_3, Ln_3, C\#_3, AD_1, AD_2),$

$input_5(Pd_1, Rd_1, Ct_1, AD_2, AD_3),$

$input_4(Fn_4, Ln_4, C\#_4, AD_3, AD_4),$

$input_{5'}(Pd_2, Rd_2, Ct_2, SD_1, SD_2),$

$input_4(Fn_5, Ln_5, C\#_5, AD_4, AD_5),$

$input_6(Pd_3, Rd_3, Ct_3,\ C\#_5,\ AD_5, AD_6, SD_2, SD_3).$

The following two points are important in generating the guard.

1. Variable Renaming - Variables should be renamed to be unique in the guard, even though they are the same in the DIS. The exception is variables that are inherited from a previous transition (the inheritance is part of the DEFSM modeling). If a later transition inherits a variable

from an earlier transition then the same variable is used. For example, $C\#_5$ is passed into transition 6 from transition 4.

2. Database versions - The database is (potentially) modified in each transition, so the output version of a database in one transition, e.g., AD_2, is passed as input into the next transition.

To generate a test case, the guard is evaluated. In a successful evaluation, values will be bound to the variables. The binding produces a test case. An unsuccessful evaluation implies that the initial AD/SD is not in a good state, so the test sequence is put back to the queue.

5.3 Step Mode Execution and Test Result Evaluation

The generated test case will be subsequently evaluated on the web application after each transaction commits, the updated AD is checked by an oracle. But first, the oracles have to be generated.

5.3.1 On-demand Oracle Generator

For black box testing, the state of the AD is not visible to the tester. Instead, AutoDBT generates an oracle to determine whether the current state of the

AD satisfies a correctness property, for instance, whether a deletion described in the update specification has actually been done. An oracle is a set of rules, which are evaluated to determine success of a test case. AutoDBT generates the oracle from the update specification.

Definition 6 (Oracle) *An oracle is a set of Prolog rules of the following two forms.*

*1) insert_*Relation*_Failed :-*

$\quad\quad$ *insert_AD_*Relation*($A_1, \ldots, A_k$),*

$\quad\quad$ ¬AD_Relation($A_1, \ldots, A_k$).

*2) delete_*Relation*_Failed :-*

$\quad\quad$ *delete_AD_*Relation*($A_1, \ldots, A_k$),*

$\quad\quad$ AD_Relation($A_1, \ldots, A_k$).

∎

The first rule defines a failed insertion operation. Tuples that should have been inserted into AD_*Relation* haven't been inserted correctly. In other words, some tuples stored in the insert_AD_*Relation* can not be found in AD_*Relation.* The second rule defines a failed deletion operation. Tuples that should have been deleted from AD_*Relation* haven't been deleted correctly which means that some tuples stored in the delete_AD_*Relation* can still be found in AD_*Relation*. Table 5.1 shows the oracles generated from the update specification in Table 4.5.

Before the test data for a transaction is executed on the web application, the associated update specification (if there is any) is evaluated on the most current AD/SD version, then all the tuples that should be inserted or deleted will be stored in the insert_AD_*Relation* or delete_AD_*Relation*, respectively. Unlike the other Prolog queries, it is best to evaluate the update specification using a bottom-up strategy, like Datalog. The reason is that more than one tuple could be inserted or deleted, and bottom-up evaluation will capture all of the insertions or deletions. Before the evaluation begins, the update specification is adjusted by unifying the variables inherited from the previous or current transitions with the values generated in the test case. The facts generated on test case generated are used to evaluate the update specification.

Transition	**Oracles**
6	delete_Reservation_Failed :- delete_AD_Reservation(*C#*, *Pd*, *Rd*, *Ct*), AD_Reservation(*C#*, *Pd*, *Rd*, *Ct*). insert_Reservation_Failed :- insert_AD_Reservation(*C#*, *Pd*, *Rd*, *Ct*), ¬ AD_Reservation(*C#*, *Pd*, *Rd*, *Ct*).
10	delete_Reservation_Failed :- delete_AD_Reservation(*C#*, *Pd*, *Rd*, *Ct*), AD_Reservation(*C#*, *Pd*, *Rd*, *Ct*). delete_Reserves_Failed :- delete_AD_Reserves(*Cid*, *C#*), AD_Reserves(*Cid*, *C#*).

Table 5.1: The oracles for the Modify and Cancel Reservation paths

Oracles are evaluated during a test as follows.

Definition 7 (Oracle evaluation) *An oracle is evaluated by executing one of the following Prolog queries.*

- *?- insert*_Relation_*Failed.*
- *?- delete*_Relation_*Failed.*

If an oracle succeeds, then the corresponding update operation failed. ∎

Table 5.2 shows the queries to evaluate the oracles defined in Table 5.1. The oracle is always evaluated using the current version of the AD/SD. If an oracle fails, then the testing process for this test case aborts. Otherwise, testing will continue. The following test case generation process will continue using the

Transition	**Oracle Evaluation**
6	?- delete_Reservation_Failed. ?- insert_Reservation_Failed.
10	?- delete_Reservation_Failed. ?- delete_Reserves_Failed.

Table 5.2: Evaluating the oracles for the Modify and Cancel Reservation paths

AD/SD state left over by the previous processes, but with a new pair of empty AUL and SUL.

Chapter 6

Analysis of the Test Report

After the testing process is finished, the test report is analyzed to find out whether the application is correctly implemented, or more precisely, whether the testing process has revealed any defects in the implementation. Several conditions, which can be classified into the following three categories, can lead to a failed evaluation of an oracle.

1. Modeling Error

   In this category, the modeler didn't model the transaction correctly, giving either a wrong input specification or a wrong update specification, leading to a buggy oracle, which can cause (1) correct results to be evaluated as a failure (spurious failure) or (2) coincidental failure when both the test result and oracle are buggy. In this paper, we assume that the modeler can do a perfect job on modeling the application, therefore, both the input and output specifications are given correctly, and the oracles generated based on the update specification are always trusted. Based on this assumption, this category can be eliminated.

2. Buggy Implementation of this Particular Transaction

   When a failure is encountered, the first hunch to the tester is that the implementation of this particular transaction is buggy. So the tester will first narrow down the related code of this transaction and check where the bug lies. If no bug is revealed, or after fixing the bugs and rerunning the testing process on the whole test sequence, the same failure still exists, then there is very large probability that the failure is caused by a buggy implementation of the previous transaction which is explained in the rest category.

3. Buggy Implementation of the Previous Transactions

   If the second condition fails, the cause of the failure might lie in the previous transactions. The buggy implementation of the previous transaction might distort the AD in an unexpected way. Due to the limitation of the partial oracle which only tests whether the expected updates are correctly performed rather than fully test the entire database, some bugs hidden in the previous transaction are invisible when evaluating the specified partial oracle, but cause a spurious failure of a later transaction by making the preset input data for the later transaction no longer valid. Therefore, the later transaction can not be even triggered. Currently, we can only be certain that the bugs are hidden in some transactions prior to the failure one, but can not pinpoint exactly which transaction it is if there are more than one. We hope in the future, we can narrow the range of finding a bug by introducing new techniques.

Chapter 7

Generality Analysis

By modeling the input and update specifications for the OCRS example, we demonstrate that the modeler can model the single tuple update easily, in which case only one tuple of a relation is inserted, deleted, or modified at one transition, and the AutoDBT can process the specification successfully. However, for a real web database application, the database updates might be much more complicated than what we have shown. So whether the modeler can model every input and update required for a web database application remains suspicious unless the generality of the approach has been discussed.

In the following subsections, we first explore the modeling steps of all the basic cases compounding the complicated one, then an argument is made to show that every other case can be modeled as the combination of the basic cases.

7.1 Basic Cases of Database Update

In this section, we illustrate, from a modeler's point of view, the steps of modeling the most basic cases of database updates for the transition i. For simplicity, we assume the input for the transition i has only one parameter X and the update influences only one relation R which has only one column.

- Single Tuple Insertion Modeling

  In this case, a input X retrieved from the SD/AD is used as an input of a function (func(X, Y)), then the result of the function, Y, is inserted into the AD. The function can be as simple as $X = Y$, or it can be any complicated computation.

  - Input Specification Modeling

    * Source database: AD/SD
    * Source relation: R
    * Input specification:

$$\mathit{input}_i(X) \text{ :- SD\_R}(X).$$

or

$$\mathit{input}_i(X) \text{ :- AD\_R}(X).$$

- Update Specification Modeling
    - Update database: AD (Since we only model the updates of the AD, this step can be omitted and we won't show it for the following cases.)
    - Update relation: R
    - Update Specification: insert_AD_R(Y) :- $input_i(X), func(X, Y)$.

- Single Tuple Deletion Modeling

  In this case, a single tuple is deleted from the relation R of AD.

  - Input Specification Modeling
    - Source database: AD
    - Source relation: R
    - Input specification: $input_i(X)$:- AD_R(X).
  - Update Specification Modeling
    - Update relation: R
    - Update specification: delete_AD_R(X) :- $input_i(X)$.

  Note that more than one tuple could be deleted.

- Single Tuple Modification Modeling

The modification can be modeled as a deletion followed by an insertion. In this case, a tuple specified by one input Y from previous transition j is deleted from the relation R of AD, then a new tuple specified by the input of the transition i is inserted into the same relation R.

- Input Specification Modeling
  - Source database: AD/SD
  - Source relation: R
  - Input specification:

$$\mathit{input}_i(X) \text{ :- } \mathit{input}_j(Y),\ \text{AD\_R}(Y),\ \text{func}(X, Y).$$

or

$$\mathit{input}_i(X) \text{ :- } \text{SD\_R}(Y),\ \text{func}(X, Y).$$

- Update Specification Modeling
  - Update relation: R
  - Update specification:

$$\text{delete\_AD\_R}(Y) \text{ :- } \mathit{input}_j(Y).$$

$$\text{insert\_AD\_R}(X) \text{ :- } \mathit{input}_i(X).$$

7.2 Generality Argument

All the complicated cases can be modeled as a combination of the basic cases. In the following, we illustrate a few complicated cases.

- Multiple Tuples Insertion Modeling

  In this case, the transition i has more than one input parameters, i.e., $V_i^1, \ldots, V_i^n (1 \leq n)$. Each $V_i (1 \leq i \leq n)$ retrieved from the SD/AD is used as an input of a function $(func_i(V_i, Y_i))$, then the result of the function, Y_i, is inserted into the AD. In the worst case, each V_i is inserted into a unique relation R_i.

  - Input Specification Modeling
    - Source database: AD/SD
    - Source relation: $R_1, \ldots, R_n$
    - Input specification:

      $$input_i(V_1, \ldots, V_n) \text{ :- SD\_}R_1(V_1), \ldots, \text{SD\_}R_n(V_n).$$

      or

      $$input_i(V_1, \ldots, V_n) \text{ :- AD\_}R_1(V_1), \ldots, \text{AD\_}R_n(V_n).$$

  - Update Specification Modeling
    - Update database: AD (Since we only model the updates of the

AD, this step can be omitted and we won't show it for the following cases.)

  * Update relation: R
  * Update Specification:

$$\text{insert\_AD\_}R_1(Y_1) \text{ :- } input_i(V_1, \_, \ldots, \_), func_1(V_1, Y_1).$$
$$\vdots$$
$$\text{insert\_AD\_}R_n(Y_n) \text{ :- } input_i(\_, \ldots, V_n), func_n(V_n, Y_n).$$

- Multiple Tuples Deletion Modeling

  In this case, multiple tuples which have the same attribute value specified by the input of transition i are deleted from the relation R of AD. The input and update specification are the same as the ones of single tuple deletion

- Multiple Tuples Modification Modeling

  We use an example to illustrate this case. For instance, in the Reserves relation of OCRS example, increase the Confirmation# by 1 to those tuples with a CustomerID = the input of transition i.

  – Input Specification Modeling

    * Source database: AD

* Source relation: Reserves
* Input specification:

  $input_i(CID)$:- AD_Reserves(CID, _).

- Update Specification Modeling
  * Update relation: Reserves
  * Update specification:

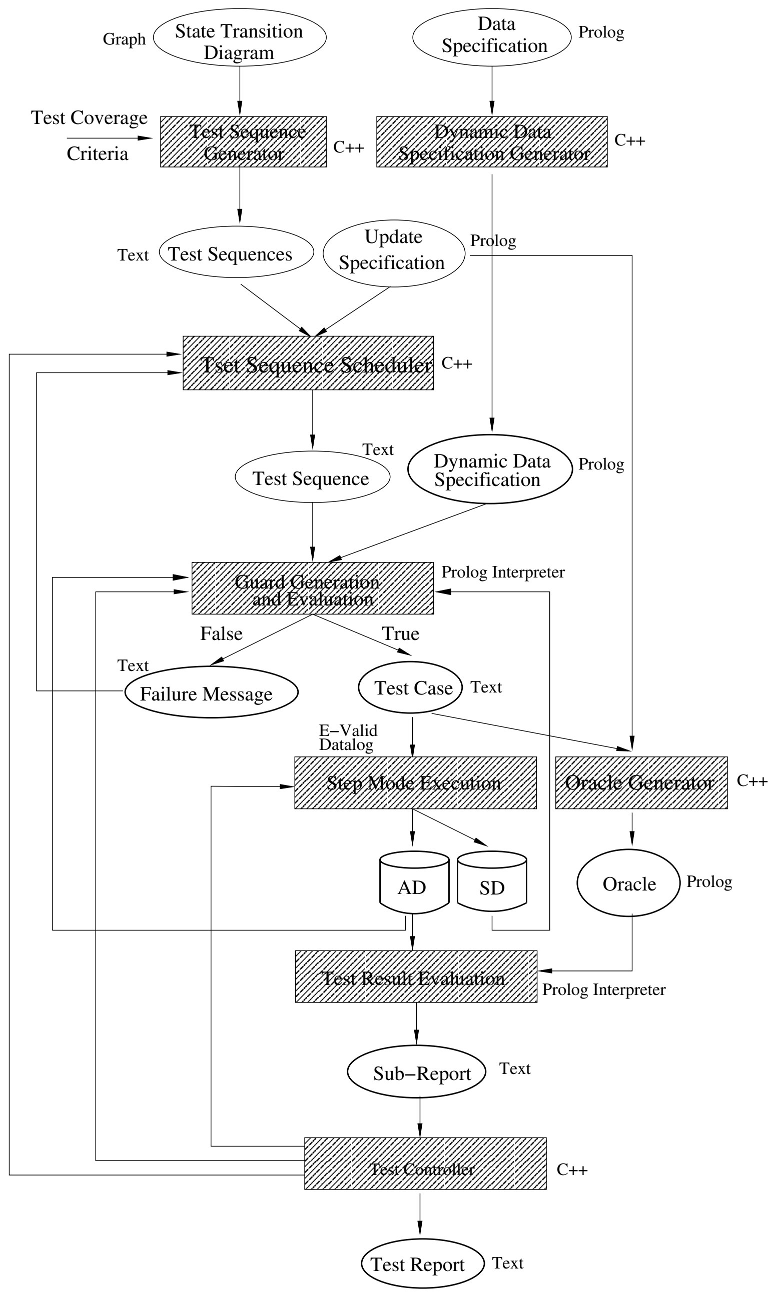
    delete_AD_Reserves(CID, $C\#$) :-

    $input_i(CID)$, AD_Reserves(CID, $C\#$).

    insert_AD_Reserves(CID, $C\#$) :-

    $input_i(CID)$, AD_Reserves(CID, $C\#'$), $C\# = C\#' + 1$.

  When $CID = c0004$, two tuples are modified.



Note: the note besides each component is the language or tools the component will use and the input/output type.

Figure 7.1: Implementation Framework

Chapter 8

Implementation Architecture

8.1 Overview

Combining the Figure 4.1 and Figure 5.1, Figure 7.1 is created, in which each shaded rectangular component is a tool that needs to implemented. In the following subsections, we sketch out the implementation plan for each component.

8.1.1 Test Sequence Generator

The inputs of the test sequence generator are a directed graph which is the FSM of the web application and a test coverage criterion, while the output is a set of test sequences. Since the FSM is a directed graph, some well-known graph theory algorithms can be applied to achieve the given test coverage criterion. For instance, as for the OCRS, if we'd like to test each branch separately, the graph is first separated into four components. In each component, if we'd like to execute every possible action in an efficient manner, an algorithm which solves the New York Street Sweeper Problem can be applied.

8.2 Dynamic Data Specification Generator

The dynamic data specification generator includes two parts: the data inheritance graph (DIG) generator and the dynamic input specification (DIS) generator.

1. DIG Generator

   The inputs of the DIG generator are the input and update specifications, while the output is a directed graph represented as an adjacency matrix. We set $a_{ij} = k, (1 \leq k)$ if the variable represented by k is inherited from transition j to transition i. Since the inputs are Prolog rules, so a parser is needed to parse the specifications, and it can be generated by using a lexical analyzer generator *lex* and a parser generator *yacc*. Whenever the parser finds a pattern $input_i (1 \leq i \leq n)$ in the body predicates of the input or update rule of the transition j, an integer k representing the variables associated with that predicate is set in a_{ij} of the adjacency matrix.

2. DIS Generator

   The inputs of the DIS generator are the output of DIG, input and output specifications, while the output is the Dynamic Input Specification (DIS). Each rule of DIS can be generated in two steps as follows.

   (a) Head Generator

      The head of each rule is generated by modifying the head of each input specification based on the input and update specification, and the DIG.

      Algorithm 8.1

```
For each $i^{\text{th}}(0 \leq i \leq n)$ rule in the input sepcification
  If AD (SD) prefixes any body predicate of the input
      or update rule, add $AD_{in}, AD_{out}$ $(SD_{in}, SD_{out})$
      to the head of the rule.
  For each $a_{ij}(0 \leq j \leq n)$ in DIG
    If $a_{ij} \neq 0$, add the variable name represented by
      the $a_{ij}$ into the head of the rule.
```

   (b) Body Generator

The body of the DIS is generated by modifying the body of the original input specification by, first, as a parameter, adding $AD_{in}(SD_{in})$ into each corresponding body predicate prefixed by AD (SD), then appending the body of the update rule and the database update predicates to the body of the input rule.

8.3 Test Sequence Scheduler

The input of the test sequence scheduler is a set of test sequences waiting to be tested and the update specification, while the output is a queue of the ordered test sequences. It can be implemented in the following way.

First, we assign two variables *Deletion* and *Insertion* to each test sequence, which are initiated as 0. For each test sequence, a parser, generated by using lex and yacc, is used to parse the head predicates of the update specification of each transition, whenever a *delete* is encountered, the variable *Deletion* is increased by 1. Similarly, whenever an *insert* is encountered, the variable *Insertion* is increased by 1.

The test sequences are classified into four groups based on the final values of the variables *Deletion* and *Insertion* in the following way.

1. Read Only: $Deletion = 0$ and $Insertion = 0$;

2. Insertion Only: $Deletion = 0$ and $Insertion \geq 0$;

3. Mixed Insertion and Deletion: $Deletion \geq 0$ and $Insertion \geq 0$;

4. Deletion Only: $Deletion \geq 0$ and $Insertion = 0$

The four groups are ordered in the above order and the order of the test sequences in one group is random. Each test sequence is assigned a sequence number *SN* and the ordered *SN* can be maintained in a linked list.

8.4 Guard Generator

The inputs of the Guard Generator are the first test sequence of the test sequences queue formed by the test sequence scheduler, DIS and DIG, and the AD and SD, while the output is a test case if the guard evaluates to true, or a failure message otherwise.

The guard is generated by the following four functions.

1. Head_Concatenator: Concatenating all the head predicates of the DIS associated with the entire test sequence. (Input: the DIS, test sequence. Output: a string)

In this step, a parser is used to parse all the head predicates associated with the test sequence from the DIS. A string is formed after concatenating all the associated head predicates which are separated by ",".

2. Variable_Renamer: Renaming all the variables of each predicate. (Input: the string generated by Head_Concatenator. Output: a table.)

   A Name_Generator, which generates a unique string each time invoked, is invoked whenever a variable, except the $AD_{in}, AD_{out}, SD_{in}, SD_{out}$, is encountered in parsing the string generated by Head_Concatenator, and the variable is replaced by the new name generated. A table (T) is maintained to store the correspondence between transition number, the original variables and its new name. The following function is invoked to ensure the data inheritance between transitions.

3. Data_Inheritance: If a variable (e.g. A) inherits a value from a previous variable (e.g. B), then rename variable A as B. (Input: the string generated by Head_Concatenator, the table T generated by Variable_Renamer and DIG. Output: a string.)

   Check DIG to find any data inheritance between transitions. If there is any, e.g. $a_{ij} = k$, which means a variable represented by k e.g. $C\#$ is inherited from transition j to i, then check the tuples associated with transition j and i in table T and change the new name of the $C\#$ in the transition i to the new name of the $C\#$ in the transition j. A new string is generated by changing the variable in the input string to the updated new name. The string is returned.

4. Pass_DBVersion: Passing the database version from one transition into the next. (Input: the string generated by Data_Inheritance. Output: the guard)

   Algorithm 8.2

```
$i = 0$;
$j = 0$;
Parsing the input string;
For each $t^{\text{th}}$ $(0 \leq t \leq n)$ predicate
    If $AD_{in}$ is matched
        Rename $AD_{in}$ to $AD_i$;
        i++;
        Rename $AD_{out}$ to $AD_i$;
    If $SD_{in}$ is matched
        Rename $SD_{in}$ to $SD_j$;
        j++;
```

```
            Rename $SD_{out}$ to $SD_j$;
        t++;
    return the new string.
```

After the guard is generated, it is evaluated based on the most current AD and SD with a pair of empty AD_0/SD_0.

8.5 Oracle Generator

The input of the Oracle Generator is the update specification, while the output is the oracle rules.

Algorithm 8.3

```
Parsing the update specification;
For each update rule
    If the head = insert_AD_$Relation(A_1, \ldots, A_k)$
        set oracle as
        insert_$Relation$_Failed :-
            insert_AD_$Relation(A_1, \ldots, A_k)$,
            $\neg$AD_$Relation(A_1, \ldots, A_k)$.
    If the head = delete_AD_$Relation(A_1, \ldots, A_k)$
```

set oracle as

delete_*Relation*_Failed :-

  delete_AD_*Relation*$(A_1, \ldots, A_k)$,

  AD_*Relation*$(A_1, \ldots, A_k)$.

8.6 Step Mode Execution

We adapt a capture/playback tool: E-Valid to execute the test case.

8.7 Test Result Evaluation

After a transaction commits, the corresponding oracle is evaluated on the current version of AD in a Prolog environment.

8.8 Test Controller

Test controller is implemented to control the testing process. The testing process continues at the Step Mode Execution if more steps need to be executed, at the Guard Generation and Evaluation if more test cases are needed, at the Test Sequence Scheduler if more sequences are waiting to be tested.

Chapter 9

Related Work

Much of the literature on testing web applications is in the commercial sector and tests non-functional aspects of the software. An extensive listing of existing web test support tools is on a web site maintained by Hower [11]. The list includes link checking tools, HTML validators, capture/playback tools, security test tools, and load and performance stress tools. These are all static validation and measurement tools, none of which support functional testing or black box testing.

Kung, Liu, and Hsia [13, 14] developed a test generation method based on multiple models of the applications under test. The models include Object Relation Diagrams, Object State Diagrams, a Script Cluster Diagram, and a Page Navigation Diagram. This model assumes that the source is available, whereas our research does not. Also, this paper uses an enhanced Finite State Machine that includes representation of test constraints and does not need multiple types of diagrams. Unlike Kung et al., we also represent the FSM via logical, rather than physical web pages and solve potential state space explosion problems through partitioning and a different approach towards input description on the edges of the FSM.

Lee and Offutt [16] describe a system that generates test cases using a form of mutation analysis. It focuses on validating the reliability of data interactions among web-based software components. Specifically, it considers XML based component interactions. This approach tests web software component interactions, whereas our current research is focused on the web application level.

Ricca and Tonella [20] proposed a UML model of web application for high level abstraction. The model is based entirely on static HTML links and does not incorporate any dynamic aspects of the software. Any web application can be seen as an instance of the UML model. The model is supported by a tool that creates a static graph based on HTML links and another that creates tests comprised of sequences of URLs. Although the paper claims that the tools can "guarantee that all paths in a web site" are covered, assumptions about data inputs and lack of information about dynamically created links clearly limit the paths that are covered.

Yang et al. [25, 26] present an architecture for test tools that is directed towards testing web applications. The architecture consists of five subsystems including test development, test measurement, test execution, test failure analysis and test management. From the paper, it is not clear whether the test architecture includes new tools or whether it is meant to incorporate existing tools. The FSM modeling-based tool proposed in [1] satisfies the test development and test measurement portion of Yang et al.'s test architecture.

Jia and Liu [12] proposes an approach for formally describing tests for web applications using XML. A prototype tool, WebTest, was also developed. Their XML approach could be combined with the test criteria proposed in [18] by expressing the tests in XML.

Benedikt, Freire and Godefroid [6] presented VeriWeb, a dynamic navigation testing tool for web applications. VeriWeb explores sequences of links in web applications by nondeterministically exploring action sequences, starting from a given URL. Excessively long sequences of links are limited by pruning paths in a form of path coverage. VeriWeb creates data for form fields by choosing from a set of name-value pairs that are initialized by the tester. VeriWeb is the most similar work to the ideas presented in [1]. The primary difference is in the graphs that are used and the technique applied to reduce their size. VeriWeb's testing is based on graphs where nodes are web pages and edges are explicit HTML links, and the size of the graphs is controlled by a pruning process. We rely on DEFSM models of the web application and use aggregation abstraction of the DEFSMs to control the size.

None of the above papers deal with application databases. Database testing research has focused primarily on techniques to automatically populate a database with synthetic data for testing purposes [7, 10]. These approaches are complementary to our research in modeling and testing the interaction of a web application with a database. Database benchmarks like TPC are popular for assessing the performance of DBMSs [22, 21]. However, DBMS benchmarks aren't useful for functional testing of database applications, though they could play a role in the future if we extend AutoDBT to performance testing.

Finally, Prolog has been a popular language and tool used in AI for efficient search. However, to our best knowledge, the use of Prolog in test case generation in this paper is novel.

Chapter 10

Conclusions and Future Work

There is a need for strategies to automate testing of web database applications since relying on manual testing is ineffective for many such applications. AutoDBT extends the functional testing of a web application to include database updates. To use AutoDBT, a modeler develops a model of the application. The model consists of a state transition diagram that shows how users navigate from page to page in the application, and a Data Specification that describes how data is input and the database is updated. Once the model is complete, a tester decides on test coverage criteria. AutoDBT uses the model and the coverage criteria to generate test cases. Each test case is a self-contained test of the web application. AutoDBT selects test case data from either the application database or the synthesized database as needed. AutoDBT also generates a guard for each test. The guard checks whether the current state of the database is conducive to generating a test case, since previous tests may have modified the database to an extent that renders the test case generation impossible. Finally, AutoDBT also generates an oracle for each test. The oracle checks whether a test case has correctly updated the application database.

The main contributions of this paper include a design of AutoDBT and the identification of testing as an important, open issue for web database applications. Much remains to be done. We plan to complete the implementation of AutoDBT. We have yet to build tools for many of the components in the framework. We designed AutoDBT for functional testing, but we'd like to investigate other kinds of testing of web database applications, such as performance testing. Another vein of future work is to improve the feedback on failed tests. A test fails because an oracle detects a problem in the application database. Ideally the oracle would also provide a hint on how to fix the problem; currently no such hints are provided. Finally, we'd like to improve the guards. A guard checks whether a test case can be attempted, but it would be better if the guard populated the database with the data that is needed to perform the test.

Bibliography

[1] A. A. Andrews, J. Offutt, R. T. Alexander. Testing Web Applications By Modeling with FSMs. Submitted to *Software and System Modeling*, Springer-Verlag.

[2] Lihua Ran, Curtis E. Dyreson, Anneliese Andrews. AutoDBT: A framework for Automatic Testing of Web Database Applications. To appear in International Conference on Web Information Systems Engineering (WISE), 2004.

[3] B. Beizer. Black-Box Testing. Wiley, 1995.

[4] E. Beltrami. Models for Public Systems Analysis, 1977.

[5] L. Bodin, A. Tucker. Model for Municipal Street Sweeping Operations. In *Modules in Applied Mathematics Vol. 3: Discrete and System Models*, 1983.

[6] M. Benedikt, J. Freire, P. Godefroid. VeriWeb: Automatically Testing Dynamic Web Sites. In *Proceedings International WWW Conference(11)*, Honolulu, Hawaii, USA. 2002

[7] D. Chays, S. Dan, P. G. Frankl, F. I. Vokolos, E. J. Weyuker. A Framework for Testing Database Applications. In *Proc. of the International Symposium on Software Testing and Analysis*, pp. 147-157, 2000.

[8] T. S. Chow. Testing Software Design Modeled by Finite-State Machines. In *IEEE Transactions on Software Engineering*, SE-4(3), pp. 178-187, May 1978.

[9] S. Fujiwara, G. Bochmann, F. Khendek, M. Amalou, and A. Ghedasmi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591-603, June 1991.

[10] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, P.J. Weinberger. Quickly Generating Billion Record Synthetic Databases. In *Proc. ACM SIGMOD*, 1994

[11] R. Hower. Web site test tools and site management tools. Software QA and Testing Resource Center, 2002. `www.softwareqatest.com/qatweb1.html` (accessed November 2003).

[12] X. Jia, H. Liu. Rigorous and Automatic Testing of Web Application. In *Proceedings of the 6th IASTED International Conference on Software Engineering and Applications (SEA 2002)*, Cambridge, MA, USA. pp. 280-285, November 2002

[13] D. Kung, C. H. Liu, and P. Hsia. A model-based approach for testing Web applications. In *Proc. of Twelfth International Conference on Software Engineering and Knowledge Engineering*, Chicago, IL, July 2000.

[14] D. Kung, C. H. Liu, and P. Hsia. An object-oriented Web test model for testing Web applications. In *Proc. of IEEE 24th Annual International Computer Software and Applications Conference (COMP-SAC2000)*, pp. 537-542, Taipei, Taiwan, October 2000.

[15] D. Lee and M. Yannakakis, Principles and Methods of Testing Finite State Machines, a survey. In *Proceedings of the IEEE*, vol. 84, 8, pp. 1090-1123, 1996.

[16] S. C. Lee and J. Offutt. Generating test cases for XML-based Web component interactions using mutation analysis. In *Proceedings of the 12th International Symposium on Software Reliability Engineering*, pp. 200-209, Hong Kong China, November 2001. IEEE Computer Society Press.

[17] N. Nyman. GUI Application Testing with Dumb Monkeys. In *Proceedings of STAR West*, 1998.

[18] J. Offutt and A. Abdurazik. Generating tests from UML specifications. In *Proceedings of the Second IEEE International Conference on the Unified Modeling Language (UML99)*, Lecture Notes in Computer Science Volume 1723. pp. 416-429, Fort Collins, CO, October 1999.

[19] J. Ofutt, S. Liu, A. Abdurazik, and P. Ammann. Generating test data from state-based specifications. *The Journal of Software Testing, Verification, and Reliability*, 12(1):25-53, March 2003.

[20] F. Ricca, P. Tonella. Analysis and Testing of Web Applications. In *23rd International Conference on Software Engineering (ICSE)*, Toronto, Ontario, Canada. pp. 25-34, 2001

[21] D. Slutz. Massive stochastic testing of SQL, In *Proceedings of the Twenty-Fourth International Conference on Very-Large Databases*, Morgan Kaufmann, pp. 618-622, 1998

[22] Transaction Processing Performance Council. *TPC-Benchmark C*. 1998

[23] J. Whittaker, M. Thomason. A Markov Chain Model for Statistical Software Testing. In *IEEE Transactions on Software Engineering, Vol. 20*, pp. 812-824, 1992.

[24] R. Elmasri, S. B. Navathe. Fundamentals of Database Systems. Third Edition. Addison-Wesley 2000.

[25] J. Yang, J. Huang, F. Wang, and W. Chu. An object-oriented architecture supporting Web application testing. In *First Asian-Pacific Conference on Quality Software (APAQS '99)*, pp. 122-129, Japan, December 1999.

[26] J. Yang, J. Huang, F. Wang, and W. Chu. Constructing an object-oriented architecture for Web application testing. *Journal of Information Science and Engineering*, 18(1):59-84, January 2002.