AN APPROACH FOR AUTOMATED VERIFICATION OF WEB APPLICATIONS USING MODEL CHECKING AND REPLAYING THE SCENARIOS OF COUNTEREXAMPLES

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS
OF
THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

YUDUM PAÇİN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
THE DEPARTMENT OF INFORMATION SYSTEMS

SEPTEMBER 2015

AN APPROACH FOR AUTOMATED VERIFICATION OF WEB APPLICATIONS USING MODEL CHECKING AND REPLAYING THE SCENARIOS OF COUNTEREXAMPLES

Submitted by **YUDUM PAÇİN** in partial fulfillment of the requirements for the degree of **Master of Science in Information Systems, Middle East Technical University** by,

Prof. Dr. Nazife Baykal
Director, **Graduate School of Informatics** ____________________

Prof. Dr. Yasemin Yardımcı
Head of Department, **Information Systems** ____________________

Assoc. Prof. Dr. Aysu Betin Can
Supervisor, **Information Systems** ____________________

Examining Committee Members:

Assoc. Prof. Dr. Altan Koçyiğit
Information Systems, METU ____________________

Assoc. Prof. Dr. Aysu Betin Can
Information Systems, METU ____________________

Assoc. Prof. Dr. Banu Günel Kılıç
Information Systems, METU ____________________

Assoc. Prof. Dr. Halit Oğuztüzün
Computer Engineering, METU ____________________

Assoc. Prof. Dr. Vahid Garousi Yusifoğlu
Software Engineering, Haccettepe University ____________________

Date: 03/09/2015

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name	**:**	**Yudum Paçin**
Signature	**:**	**_______________**

ABSTRACT

AN APPROACH FOR AUTOMATED VERIFICATION OF WEB APPLICATIONS USING MODEL CHECKING AND REPLAYING THE SCENARIOS OF COUNTEREXAMPLES

Paçin, Yudum

M.S., Department of Information Systems

Supervisor: Assoc. Prof. Dr. Aysu Betin Can

September 2015, 137 pages

The increase in the use of web applications in various domains, raised the importance of the methodologies for verification of web applications. We propose a framework for the verification of web applications with respect to access control, link consistency and reachability properties using model checking. In this approach, users define the properties by explanatory guidance of user interface. The execution traces that lead to a property violation is translated to a script that automates the replaying of the counterexample scenarios on a web browser. This facility enables the user to observe incorrect behaviors of the web application with respect to specified properties so that the user is released from the tedious task of understanding and interpreting the counterexamples generated by the model checker. In addition, to automate this verification

process, we need to automate the model extraction of a web application to be given to the model checker as an input. To this purpose, we use two dynamic web application crawlers and automatically transform their models to an intermediate web model we have developed. This intermediate web model both enables model extraction tool independence and gives the user to edit the model manually to increase precision of verification process. In order to evaluate the tool we developed for this purpose, we conducted a user study and the participants reported our tool to be useful for detecting and visualizing errors. We also evaluated the effectiveness on real web applications and observed that the tool can reveal real faults.

Keywords: Model checking, Web software verification, Counterexample animation

ÖZ

WEB UYGULAMALARININ MODEL DENETLEME KULLANILARAK OTOMATİK DOĞRULANMASI VE KARŞI ÖRNEK SENARYOLARININ OYNATILMASI İÇİN BİR YAKLAŞIM

Paçin, Yudum

Yüksek Lisans, Bilişim Sistemleri Bölümü

Tez Yöneticisi: Doç. Dr. Aysu Betin Can

Eylül 2015, 137 sayfa

Web uygulamalarının kullanım alanının gittikçe artması, doğrulama yöntemlerinin önemini de arttırmaktadır. Bu çalışmada, model denetleme kullanılarak web uygulamalarının erişim kontrolü, bağlantı tutarlılığı ve erişilebilirlik özelliklerinin doğrulanması için bir yöntem öneriyoruz. Bu yaklaşımda, kullanıcılara sorgulanacak özellikleri ara yüz yardımıyla tanımlama imkanı verilmektedir. Sorgulanan özelliklerin ihlaline neden olan yürütme adımları, model denetleyicinin ürettiği karşı örneklerin web tarayıcısı üzerinde oynatılmasını sağlayan betiklere çevrilmektedir. Bu sayede, kullanıcıların web uygulamasında bulunan hatalı davranışları gözlemlemelerine olanak verilmekte, kullanıcıya model denetleme aracının ürettiği anlaşılması zor ve oldukça detaylı olan karşı örneklerin çözümlenmesinde yardımcı olunmaktadır. Buna ek olarak, doğrulama sürecini

otomatikleştirmek için, model denetleme aracına girdi olarak belirlenen modelin web uygulamasından elde edilmesinin de otomatikleştirilmesi gerekmektedir. Biz bu amaçla, var olan iki dinamik web arama robotunu kullanmakta ve çıkan modelleri otomatik olarak kendi geliştirdiğimiz ara web modeline çevirmekteyiz. Bu ara web modeli, hem model çıkarma aracından bağımsızlık sağlamakta hem de doğrulama sürecinin kesinliğini arttırmak için kullanıcıya çıkan modeli manuel biçimde düzenleme imkanı sunmaktadır. Ayrıca, bu amaçla geliştirdiğimiz aracın değerlendirilmesi kapsamında bir kullanıcı çalışması yürütülmüştür. Katılımcılar, hata tespiti ve hataların görselleştirilmesini yararlı bulduklarını bildirmişlerdir. Ayrıca, aracın hata tespitindeki etkililiği gerçek web uygulamalarıyla değerlendirilmiş ve aracın hataları tespit edebildiği gözlemlenmiştir.

Anahtar Kelimeler: Model denetleme, Web yazılımı doğrulama, Karşı örnek oynatımı

To my family,

Yusuf, Nuran & Doğukan Paçin

ACKNOWLEDGEMENTS

I would like to thank my advisor Assoc. Prof. Aysu Betin Can for making this experience possible for me. I thank her for her great patience and the time that she spent with me for discussing the research and suggesting new ideas. I also would like to thank for her kind and quick responses, which helped me in every stage of this work.

I also want to thank my friends, Murathan Kurfalı and Ece Kamer Takmaz for their friendship and valuable comments, which light my way most of the time.

I would like to thank also Ezgi Arslan for her great support in my first year at the graduate school, which gave me the strength to continue to my education.

I would like to thank my family for supporting me to pursue my education since I was a little child. My father, Yusuf Paçin, my mother Nuran Paçin and my brother Doğukan Paçin provided me with their unconditional support and love which become the motivation of my thesis.

TABLE OF CONTENTS

ABSTRACT iv

ÖZ vi

ACKNOWLEDGEMENTS ix

TABLE OF CONTENTS x

LIST OF TABLES xiii

LIST OF FIGURES xiv

LIST OF ABBREVIATIONS AND ACRONYMS xvi

CHAPTER

1. INTRODUCTION 1

2. BACKGROUND 5

2.1 Model Checking 6

2.2 Web Crawlers Used for Modelling 8

2.2.1 Crawljax 9

2.2.2 Micro-Crawler 10

2.3 Counterexamples 11

3. LITERATURE REVIEW 15

3.1 Verification of Web Applications with Model Checkers 15

3.1.1 Classification of the Related Studies 19

3.1.1.1 Categorization Criteria 19

3.2 Interpreting Counterexamples 23

3.3 Testing Web Applications with Model Checkers 24
4. METHODOLOGY 27
4.1 Running example 28
4.2 Modelling Web Application as Intermediate Web Model 30
4.3 Generation of Model in NuSMV 32
4.4 Properties 37
4.4.1 Navigation Errors 37
4.4.2 Access Control Errors 37
4.4.3 CTL Formulas 38
4.5 Converting Counterexamples to Executable Error Scenarios 40
5. MCWebSoft 43
5.1 System Overview 43
5.2 Implementation Details 44
5.2.1 Decomposition Description and Description of Classes 45
5.2.1.1 Data Package 45
5.2.1.2 Application Package 49
5.2.1.3 Presentation Package 51
5.2.2 Dependency Description 53
5.2.3 Third Party Tools and Packages Used 56
5.2.3.1 NuSMV 2.5.4 56
5.2.3.2 Selenium 56
5.2.3.3 com.google.gson 57
5.2.3.4 org.junit 57
6. EVALUATION 59
6.1 User Study 59

6.2 Error Detection Evaluation ... 61
6.2.1 Navigation Errors ... 61
6.2.2 Access Control Errors ... 66
6.3 Performance Evaluation ... 70
6.3.1 Performance of navigation verification ... 70
6.3.1.1 Results of the Size Measurement ... 70
6.3.1.2 Results of Performance Evaluation ... 74
6.3.2 Performance of Access Control Verification ... 76
7. DISCUSSION AND CONCLUSION ... 79
7.1 Summary ... 79
7.2 Contributions ... 80
7.3 Limitations ... 81
7.3.1 Completeness and Soundness ... 81
7.3.2 Threads to Validity of the Evaluation ... 82
7.4 Future Work ... 82
REFERENCES ... 85
APPENDICES
A. INTERMEDIATE WEB MODEL OF RUNNING EXAMPLE ... 93
B. NuSMV MODEL OF RUNNING EXAMPLE ... 101
C. OUTPUT OF NuSMV FOR RUNNING EXAMPLE ... 105
D. A JUNIT CLASS TO REPLAY FOR RUNNING EXAMPLE ... 109
E. CLASS DIAGRAM OF MCWebSoft ... 111
F. THE QUESTIONNAIRE ... 113
G. NAVIGATIONAL VERIFICATION RESULTS ... 115

LIST OF TABLES

TABLES

1. Categorization of MCWebSoft 20
2. Categorization of the related studies 21
3. Navigation and access control properties in CTL 38
4. Items of the user study evaluation and responses of all participants 60
5. Perceived usefulness and perceived ease of use results 61
6. Subject web applications used in verification of navigation properties 62
7: Results of verification of navigation properties 64
8. Number of real faults and false alarms generated before and after model refinement 66
9. Results of verification of access control properties 68
10. The webstates detected in Version 1 68
11. The webstates detected in Version 2 69
12. The webstates detected in Version 3 69
13. Results of size measurements of subject web applications in Table 6 73
14. Time measurement results of subject web applications in Table 6 74
15. Results of performance evaluation of the Restaurant Script web site 77

LIST OF FIGURES

FIGURES

1. Overview of the approach 2

2. A high-level representation of the subjects covered in the study 5

3. A sample Kripke structure, K. The states marked with x are the states where the atomic property x holds [21] 11

4. Counterexample of AF ¬x in K. The counterexample shows that there is a loop where x is always true which violates the property specifying x will eventually be false in all paths [21] .. 12

5. A sample counterexample generated with NuSMV model checker 12

6. Workflow of the approach 28

7. A representative model of the running example 29

8. The intermediate web model of the running example (Excerpt) 31

9. Kripke structure representation of the running example 34

10. The NuSMV model of the running example (Excerpt) 35

11. A counterexample generated by NuSMV for the running example 41

12. Auto generated JUnit code from the counterexample in Figure 11 (Excerpt) 42

13. A high-level overview of the system 43

14. The detailed representation of the verification processes depicted in Figure 13 44

15. Package structure of MCWebSoft 45

16. WebState Class 45

17. Element Class.. 46
18. Edge Class.. 46
19. MicroCrawlerIntFormat Class 47
20. CrawljaxIntFormat Class 47
21: IntFormat Interface 48
22. Trace Class.. 48
23. CreateIntFormat Class 49
24. IntFormatToNuSMV Class.. 50
25. NuSMVExecuter Class 50
26. CounterExampleToScript Class... 51
27. GUI Class.. 52
28. Class diagram of the Data Package.. 53
29. Class diagram of the Application Package 54
30. Class diagram of the Presentation Package.. 55
31. Relations between classes of data and application packages 55
32. Relations between classes of application and presentation packages 56
33. Technology Acceptance Model (TAM) (Adapted from [59]) 59
34. Verification results with respect to Micro-Crawler's output 64
35. Verification results with respect to Crawljax's output.. 65
36. The model of the Restaurant Script web site 67
37. Code fragment causing access control errors.. 67
38. Boxplot of file size of subject web applications 71
39: Boxplot of average DOM size of subject web applications.................................... 71
40. Boxplot of average number of edges in intermediate web models subject web applications 72
41. Boxplot of average number of webstates in intermediate web models subject web applications 73
42. Boxplot of average time spend in verification of subject web applications........................... 74
43. Scatterplot of average number of edges and average time..................................... 75
44. Scatterplot of average number of webstates and average time 76

LIST OF ABBREVIATIONS AND ACRONYMS

ADM	: Ariande Development Method
CSS	: Cascading Style Sheets
CFG	: Control Flow Graph
CTL	: Computation Tree logic
DOM	: Document Object Model
EE	: Enterprise Edition
FSM	: Finite State Machine
GUI	: Graphical User Interface
HTML	: HyperText Markup Language
HTTP	: HyperText Transfer Protocol
Id	: Identity
IDE	: Integrated Development Approach
IIS	: Internet Information Server
KB	: KiloByte
LTL	: Linear Temporal logic
MB	: MegaByte
MDA	: Model Driven Approach
ms	: MiliSeconds
OBDD	: Ordered Binary Decision Diagram
ORD	: Object Relation Diagram
SMV	: Symbolic Model Verifier

UML	: Unified Modeling Language
UWE	: UML-based Web Engineering
URL	: Uniform Resource Locator
WAG	: Web Application Graph
WebML	: Web Modeling Language
XMI	: XML Metadata Interchange
XML	: eXtensible Markup Language

CHAPTER 1

INTRODUCTION

Web applications have been increasingly common and effective in our daily lives, where they are heavily used in various types of domains as commerce, education, health and government. In addition, with the rapid evolving technologies in web development, web applications have become capable of more significant transactions, and as a result, more complicated and mission critical. Therefore, an effective way to assure the correctness of these applications emerges as an important necessity.

Verification of software is used to determine whether the developed software meets its destined specifications [1]. Verification procedures increase the reliability of applications by identifying the undesirable functions before they cause bigger problems. Model checking is a formal, model-based verification technique. Using this method provides several benefits. One advantage of using model checking is that it performs automated exhaustive search on state space of the system under test. Users are only responsible for providing a high-level finite state model of a system and properties to be verified for model checking. Second, model checkers produce an error trace, called counterexample, for each violated property. This feature helps developers to localize errors and to fix the verified system according to scenarios in counterexamples.

Despite these advantages, model checking is a challenging technique for developers who are not familiar with formal methods. Defining the properties and understanding the counterexamples are some of the complex tasks for non-expert users. Also, in order to perform model checking, the system should be modelled as a finite state machine (FSM). However, modelling web applications is not an easy task because of the complex and rich structures they have.

In this thesis, we aim to introduce users a way of applying this powerful verification technique, model checking, to web applications without formal methods background and help with the challenging tasks of using model checking discussed above. A tool-chain, MCWebSoft, is implemented for this aim.

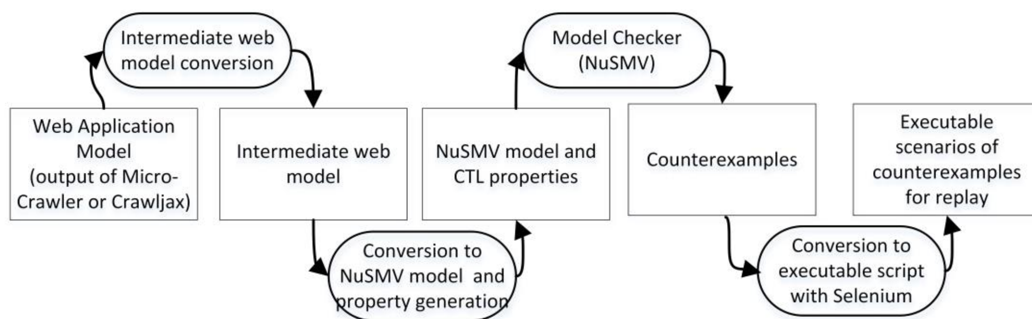

Figure 1: Overview of the approach

The workflow of MCWebSoft is depicted in Figure 1. In our approach, for defining model of web applications, the tool is made compatible with models generated from web crawlers. We have used two web crawlers, Crawljax [4] and Micro-Crawler, subsequent version of WebMole [5]. This model is then converted to an XML-based model, which is called intermediate web model, in order to prevent being fully dependent on the web crawler outputs and enabling user modifications to the model extracted from web crawlers.

NuSMV [6] is embedded in our approach as the model checker. Therefore the intermediate web model is converted to a NuSMV model. Complexities of defining properties to be verified are eliminated by means of a user interface that helps to generate temporal logic properties required for verifying navigation and access control mechanism of web applications. Finally, counterexamples produced by the model checker is interpreted as replaying error scripts based on Selenium's WebDriver to enable users to visualize the detected errors.

With MCWebSoft, we also aim to find navigation and access control errors of web applications with model checking. Navigation errors, within the scope of this thesis, consist of faults preventing reachability between web pages. Reachability is one of the key aspects of web applications, since navigation between web pages is established by the communication in client server system which forms the general architecture of the web applications [2]. Also, reachability between all states of a web application is an indicator of a good design [3]. In some web applications, reachability of particular states require authorization of users. For restricting the access of private services or contents, web applications employ access control mechanisms. Errors arising from any gap in these mechanisms can cause unauthorized users to access private contents or services, which constitute another type of errors aimed to be detected in this study called access control errors.

We conducted a user study to evaluate the ease of use and usefulness of MCWebSoft. The participants had no experience in formal methods. These non-experienced users succeeded to define their own properties and checked web application with a model checker for the first time. The participants reported our tool to be useful for detecting and visualizing errors.

In addition, we evaluated our tool with 30 real web applications to investigate the capability of detecting navigational errors, and found real reachability errors. However, there were false alarms due to inaccuracies in the models extracted by crawlers. We refined the web models using only the observations when scenarios are replayed and were able to eliminate all false alarms. Detecting

the access control errors were evaluated using fault injection on an open source web application. We injected access faults by removing the delegations with authorization policy at several locations in the source code. This resulted in accessing the private states without the authorization page. MCWebSoft succeeded to detect the access control errors in each faulty version.

The remainder of this document is organized as follows: In Chapter 2, background information of the thesis, which includes descriptions of model checking, web crawlers used for modelling, counterexamples, is given. In Chapter 3, the literature survey on model checking usage on verification of web applications, testing web applications and previous studies about interpretation of counterexamples are presented. In Chapter 4, MCWebSoft is explained with its methodology. Whole methodology is explained step by step with a running example and property definitions are introduced in detail. In Chapter 5, implementation details and the system overview of MCWebSoft are given. Information about the design and third party software packages, tools used for MCWebSoft are presented in this chapter. In Chapter 6, the evaluation of MCWebSoft which is comprised of user study, error detection evaluation and performance evaluation is presented. In Chapter 7, conclusion and discussion of the study is given with the summary, contributions, limitations and future work subsections.

CHAPTER 2

BACKGROUND

This study combines different methods to allow inexperienced users to apply model checking for verification of web applications. In this chapter, descriptions of methods, tools and contexts constituting the framework of the study are given.

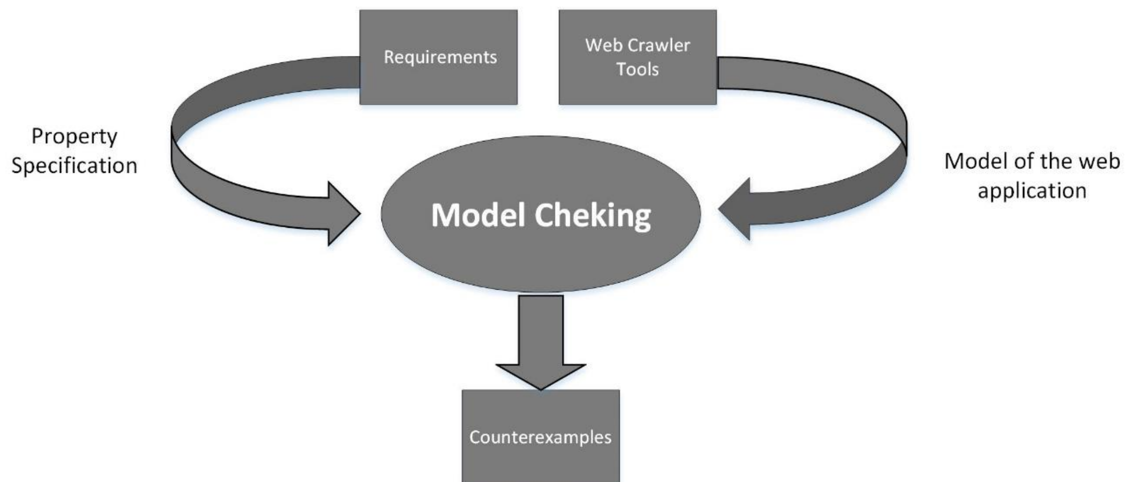

Figure 2: A high-level representation of the subjects covered in the study

In Figure 2, a general view of the subjects included in our proposed model are given to better explain the background information. In Section 2.1, model checking, the method used for verification, is described. As a model based method, model checking requires a model of the system under test. The system models are obtained from two web crawlers in our approach. These web crawlers and the structure of the models they produce are presented in Section 2.2. Property specification is made to check whether the model of web application satisfies certain requirements. In our approach, properties are defined to find navigation and access control errors, but this section is not included in this chapter, since the concept of property definition is more suitable for Chapter 4 in which the methodology of the study is explained (see Section 4.4). Finally, in this study, we used output of model checker to replay errors of web applications. In the output file of model checker, we used counterexamples for replaying the error traces. Therefore, counterexamples are defined in Section 2.3, in more detail.

2.1 Model Checking

Model Checking is an automated formal verification technique for evaluating functional properties of system models [7]. Model checking examines all system space to verify a system model against properties which are the specifications that define what the system should or should not do. A system model describes how the system behaves. Model checking performs state space exploration on the system model to find whether the model fulfills the given property. If the property holds for the system model, the execution terminates with a message indicating that the property is true. Otherwise, violated property is displayed with a counterexample showing the sequences of states leading to property violation [7].

In the rest of this section, formal and detailed descriptions of the prerequisites for model checking technique are given. First prerequisite is the model of the system. The model expresses the behavior of the system in a FSM which consists of a set of states and a set of transitions. States give current values of variables. Transitions describe how the system evolves.

The model specified for model checking method should be formal; the formalism employed to describe model and properties used for model checking is called Kripke structure [8].

Definition: Kripke structure is defined as a 4-tuple $K = (S, S_0, T, L)$ where,

- S is a finite set of states
- $S_0 \subseteq S$ is an initial state set,
- $T \subseteq S \times S$, is a left-total transition relation; for each $s \in S$ there is a $s' \in S$ such that $(s, s') \in T$,
- $L: S \rightarrow 2^{AP}$, is a labelling function that maps each state to a set of atomic representation that hold in this state, where AP is a set of atomic propositions.

In a Kripke structure, all possible behaviors of the system are expressed with a sequence of states, called path. Definition of a path, according to the Kripke structure definition above, is as follows:

Definition: A path in the Kripke structure $K = (S, S_0, T, L)$ is an infinite sequence $s_0, s_1, s_2, ..$ in S such that $\forall i \geq 0, (s_i, s_{(i+1)}) \in T$.

Second prerequisite for model checking is the properties. The properties specify the desired features of the model. They are defined as formulas of temporal logic. Temporal logic describes the sequences of transitions between states [9]. A temporal logic formula's truth is not static, but determined by a point of time inside the model. The time point specification is made implicitly with the keywords in temporal logic operators, as eventually, never and globally [10].

The most commonly preferred temporal logics are Computation Tree Logic (CTL) and Linear Temporal Logic (LTL) [8]. We used CTL in this study for specifying the properties to be checked in web applications. In CTL, time is represented as branching from present time to the future, whereas in LTL, time is considered as linear.

A CTL formula consists of atomic propositions, logical operators and temporal operators. A temporal operator in CTL consists of two parts: a path quantifier (A and E) and a temporal operator (F, G, U, and X). The meaning of temporal operators, F, G, U, and X are 'in a Future state', 'always (Globally)', 'Until' and 'neXt state,' respectively. These operators are preceded by a path quantifier. The path quantifier, A means 'along All paths' and E means 'along at least (there Exists) one path' [10].

The syntax of CTL is given in Backus Naur form as,
$\phi ::= \bot \mid \top \mid a \mid (\neg\phi) \mid (\phi_1 \wedge \phi_2) \mid (\phi_1 \vee \phi_2) \mid (\phi_1 \rightarrow \phi_2) \mid AX\phi \mid EX\phi \mid AF\phi \mid EF\phi \mid AG\phi \mid EG\phi \mid A[\phi_1 U \phi_2] \mid E[\phi_1 U \phi_2]$, where a ranges over a set of atomic formulas and ϕ, ϕ_1 and ϕ_2 are state formulas.

Definition Let $s \in S$ be a state of a Kripke structure $K = (S, S_0, T, L)$ and ϕ be a state formula. According to these, $(K, s| = \phi)$ means ϕ holds at state s in Kripke structure K. $(K, s\neg| = \phi)$ means ϕ does not hold at state s in Kripke structure K. With these variables and explanations, satisfaction of CTL formulas are explained as follows:

$K, s| = \phi_1 \vee \phi_2$ iff $(K, s| = \phi_1) \vee (K, s| = \phi_2)$
$K, s| = \phi_1 \wedge \phi_2$ iff $(K, s| = \phi_1) \wedge (K, s| = \phi_2)$
$K, s_0| = AX\ \phi$ iff for every path $s_0, s_1, s_2, ..$ in K: $K, s_1| = \phi$
$K, s_0| = AF\ \phi$ iff for every path $s_0, s_1, s_2, ..$ in K: $\exists i \in \mathbb{N}:\ K, s_i| = \phi$
$K, s_0| = AG\ \phi$ iff for every path $s_0, s_1, s_2, ..$ in K : $\forall i \in \mathbb{N}:\ K, s_i| = \phi$
$K, s_0| = A\ (\phi_1 U \phi_2)$ iff for every path $s_0, s_1, s_2, ..$ in K: $\exists i \in \mathbb{N}^+: (K, s_i| = \phi_1) \wedge \forall j \in \mathbb{N}: j < i$: $(K, s_j| = \phi_2)$
$K, s_0| = EX\ \phi$ iff there exist a path $s_0, s_1, s_2, ..$ in K: $K, s_1| = \phi$
$K, s_0| = EF\ \phi$ iff there exist a path $s_0, s_1, s_2, ..$ in K: $\exists i \in \mathbb{N}:\ K, s_i| = \phi$
$K, s_0| = EG\ \phi$ iff there exist a path $s_0, s_1, s_2, ..$ in K: $\forall i \in \mathbb{N}:\ K, s_i| = \phi$
$K, s_0| = E\ (\phi_1 U \phi_2)$ iff there exist a path $s_0, s_1, s_2, ..$ in K: $\exists i \in \mathbb{N}^+: (K, s_i| = \phi_1) \wedge \forall j \in \mathbb{N}: j < i$: $(K, s_j| = \phi_2)$

In order to illustrate CTL formulas, we give some of the properties used in this thesis (see Section 4.4). The formulas are defined for the Kripke structure in which *webstate* is a variable of a state

and '*index*', '*state1*' are values of the *webstate* variable. Let $(webstate = index)$ and $(webstate = state1)$ be two atomic properties. According to these assumptions, a CTL formula $EF(webstate = index)$ expresses that it is possible to eventually reach a state where $(webstate = index)$ is true (i.e., the index is reachable).

Another property is $AG\big(webstate = index \rightarrow AF\,(webstate = state1)\big)$. This formula specifies that whenever the system encounters a state where $(webstate = index)$ is true, it will eventually reach a state where $(webstate = state1)$ is true.

LTL is the other commonly preferred temporal logic used for model checking. LTL differs from CTL, in terms of the logic of modelling the time. LTL models the time as sequence of states, called computation path. Also, LTL formulas are evaluated over paths, unlike CTL which is evaluated over states. LTL formulas implicitly check formulas for all paths. Hence, existence of a path cannot be examined in LTL [10]. Therefore, path quantifiers, A and E do not exist in LTL.

There are also several, different model checking algorithms. The first algorithm of model checking was the explicit state model checking [8]. In explicit state model checking, one reachable state is enumerated at a single time, but this can lead to state space explosion problem. That is, the size of a state space of a system can grow exponentially with the increase in the number of its processes and variables. McMillan et al. introduced symbolic model checking to solve this problem by considering large set of states at a single step [11]. The idea behind the symbolic model checking is to represent states and the transitions relation as Boolean logic formulas. In symbolic model checking Ordered Binary Decision Diagrams (OBDDs) are used for manipulations of Boolean formulas [9]. There are many model checkers supporting symbolic model checking (e.g. SPIN, SAL, SMV, and NuSMV). In this approach we used NuSMV [6], a symbolic model checker, to verify the web application models against CTL properties.

2.2 Web Crawlers Used for Modelling

A web crawler, which is also called web spider, ant or automatic indexer, is a web robot that systematically browse World Wide Web. The first web crawler, the Wanderer, was developed in 1993 for measuring the size of web [12]. In 1994, the web crawler, called the WebCrawler, was built for indexing and navigating the web [13, 14]. In this study, we used web crawlers to generate navigational maps of web applications.

For indexing web applications, traditional web crawlers follow the links in web pages successively and differentiates each web page by their URLs. However, with 2.0, traditional web crawlers have become insufficient. Web 2.0 brought the technologies that increase the interactivity of users (e.g., AJAX and JavaScript) to use. In web applications developed with these technologies, firing an event does not necessarily change the URL, but changes the Document Object Model (DOM) of the web pages.

DOM is a standard for accessing HTML and XML documents. DOM defines how the elements in the HTML documents are related with each other. In DOM, a document consists of nodes and has a tree structure, which is called DOM tree. All nodes in the document can be accessed with the DOM tree [15]. JavaScript and AJAX code can manipulate the HTML pages through the DOM

Crawljax and Micro-Crawler are two open source web crawler tools which are developed especially for AJAX-based web applications. Crawljax and Micro-Crawler automatically apply dynamic analysis, which is a type of analysis performed by execution of web applications. During the analysis, web crawlers explore and record the states of web applications by recursive crawl algorithms. Dynamically generated contents, which depend on previous actions or requests, can be crawled by these web crawlers thanks to their DOM structure based algorithms. Therefore, Crawljax and Micro-Crawler were selected to extract a FSM of web applications.

The models inferred by Crawljax and Micro-Crawler are used for model checking first in this study. However, using the models resulted from Crawljax and Micro-Crawler for the analysis of web applications was suggested before. For Crawljax, Mesbah et al., recommend using Crawljax for analyzing dynamic web applications [4]. Le Breton et al. state that the map resulted from WebMole, earlier version of Micro-Crawler, can be used for verification purposes [5]. Also, Micro-Crawler has a component to test web applications that user can write oracles and perform runtime testing on the map resulted from crawling. Crawljax does not have a testing component like Micro-Crawler, but it has been used for testing AJAX based web applications in earlier studies. [16, 17, 18, 19].

Using web crawler tools for extracting models of web application saves time of developers and give more time to think about other necessities of model checking such as defining properties. Also, such automation eliminates the system modelling burden that discourages the use of model checking. During model extraction, the number of states can blow up quite fast. This is a well-known problem in model checking domain which is called state space explosion. Micro-Crawler and Crawljax are both able to limit the number of states explored; Micro-Crawler offers a state restriction option; users can define a stopping condition to limit the size of states to be crawled. As for Crawljax, users can determine the maximum depth level and maximum number of states for crawling. Also users can limit the maximum crawling time, ignore external links or they can ignore the links matching the predefined expressions to limit the size of exploration with Crawljax.

Although, using web crawlers for model-based analysis is acceptable, there are several questions about adequacy of the generated models. Deursen et al. examined Crawljax for its challenges in the usage of web analysis and state that to increase the suitability of the resulted model and the real web application, further abstractions on state machine of Crawljax such as presenting the super states, hyper edges for displaying the recursive behaviors in web applications are needed [14]. Demarty et al., developed a technique for existing web crawlers which apply an abstraction by redefining the links in application symbolically. The abstraction technique they developed allows to reduce the size of resulted state machines [20]. As for our study, in order to compensate the inadequacies of web crawlers, we add an option for user to modify the model extracted from web crawlers.

In following subsections Crawljax and Micro-Crawler tools are explained in more detail.

2.2.1 Crawljax

Crawljax is a browser-based, open source web crawler which browses the pages. Crawljax fires the user events generating various DOM trees of web application recursively and infers a static representation of web application called as State Flow Graph (SFG). SFG is a directed graph, $G =$

$< r, V, E, L >$ where, r, is the initial state of application, V indicates the set of states and symbolizes the vertices of the graph, E is the set of directed edges between vertices in graph and represents a relation between two states if and only if there exists a clickable element connecting them, L is a labeling function which assigns a label from set of event types and DOM element properties to each edge. Although, labeling function assumes the number of event types as more than one, only the click event type can be recognized by Crawljax 3.6 which is the latest version of Crawljax available.

The algorithm of Crawljax simply detects clickable elements on each state beginning from the root state recursively and fires click events. If the DOM structure of the new state is different from previous found ones, according to difference algorithm of Crawljax, the state and the relevant edge and elements are saved. This process is continued until no clickable element is remained. When the crawling process is finished for one state, Crawljax starts the backtracking procedure. Within the backtracking, the first unexplored previous state is searched, and if there is such a state, it is crawled. Before firing the clickable event, Crawljax compares the attributes of element with the visited elements' attributes. Although this feature helps crawler to avoid looping, it causes deficiencies in SFG.

Crawljax identifies clickable elements according to the tag attributes of the element, but it compares clickable elements of whole web application rather than comparing elements on one state of web application. This situation leads to unrecognized clickable elements, since Crawljax does not add the same element twice in SFG. Assume that the tag of link to index page *<a href=index.html id="home">* is located in different pages of web application. Although, the link works properly for reaching the index page, Crawljax do not recognize them unless the values for id attributes are different in each tag of web pages. Therefore, the homepage links located in all pages should have a different id to be recognized by Crawljax.

Crawljax gives the opportunity to crawl web pages that require authorization of users by enabling form filling with particular data inputs. Hence, for verifying access of private pages, web pages that require authorization, model of web applications should be extracted from Crawljax tool, instead of Micro-Crawler.

2.2.2 Micro-Crawler

Micro-Crawler is a subsequent version of WebMole which is a browser-based, open source web crawler and web testing tool. The model inferred by Micro-Crawler is called Web State Machine (WSM). WSM is a directed graph, $G =< V, E, \delta, v_0 >$, where V, vertices of graph, represents the DOM trees in the application, E, edges of graph, represents the DOM paths, δ is a function, $\delta: V \times E \to V$, describing that clicking an element in the path, $e \in E$, in the DOM tree $v \in V$ results in the DOM tree $v' \in V$. Lastly, v_0 is the initial DOM tree (i.e., the homepage of the web application).

The algorithm of Micro-Crawler explores web application from initial state and always moves forward. After a page is explored, Micro-Crawler never goes back to these state, instead, it restarts exploration from the initial state of web application. Despite of its strong algorithm, Micro-Crawler often fails to terminate while crawling web applications, because the heuristic of Micro-

Crawler misses abstract the pages which are dynamically created using a content from a data store (e.g., item description pages created for each item in a store).

2.3 Counterexamples

In mathematics, counterexamples are used for proving the negation of a proposition (i.e., proof by contradiction). For example, disclaim of a property, which is claimed to hold for each element of a set, can be disproved by a single element in the set which does not fulfill the property. Model checkers generate counterexamples in the same manner, to give a proof of the violated property. Counterexamples of model checkers can be in tree-like (i.e., branching) or in linear path (i.e., non-branching) format. The counterexamples in linear path format are in the form of sequences of states from initial state to the state where the property is violated. Linear path structured counterexamples are generated with Linear Time Logic (LTL). In Computation Tree Logic (CTL), counterexamples are generated in the form of linear path with the help of special algorithms [8].

Now, we give an example counterexample in linear path for CTL from [21][1].

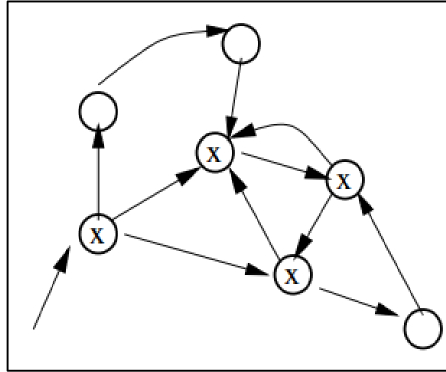

Figure 3: A sample Kripke structure, K. The states marked with x are the states where the atomic property x holds [21]

[1] In [21], the linear counterexample for $AF\ \neg x$ is given for ACTL, which is a fragment of CTL. In ACTL, only A is allowed as a path quantifier [22]. The counterexample of the Kripke structure, K against the property, $AF\ \neg x$ is also valid for CTL.

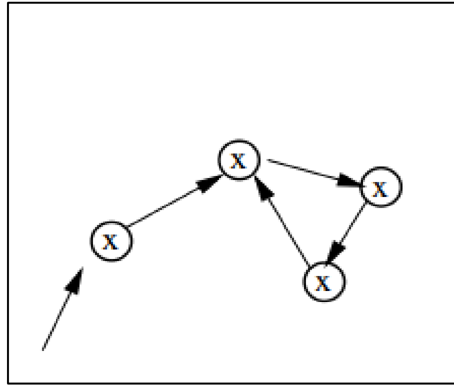

Figure 4: Counterexample of $AF\ \neg x$ in K. The counterexample shows that there is a loop where x is always true which violates the property specifying x will eventually be false in all paths [21]

In Figure 3, a sample Kripke structure, K is shown with an atomic proposition x. In Figure 4, a counterexample of the property $AF\ \neg x$ is depicted on the Kripke structure. $AF\ \neg x$ indicates that from every state it is always possible to reach a state where x is false. However, in K there is a path where negation of x is not reachable (i.e., there is a cycle where x is always true). This path constitutes of the linear counterexample of the property.

In Figure 5, a sample of a linear counterexample, which is generated with NuSMV model checker, is depicted. The sample counterexample in Figure 5, is generated against the CTL property, $AF\ \neg(x)$ where x is a Boolean variable, which can have two states as true or false.

```
-- specification AF !x  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-- Loop starts here
-> State: 1.1 <-
  x = TRUE
  y = FALSE
-> State: 1.2 <-
```

Figure 5: A sample counterexample generated with NuSMV model checker

In order to understand the counterexample, we must first know the variables, transactions that compromise the verified model. For example, by looking at the counterexample in Figure 5, a developer who has limited or no information about the model may not understand the cause of violation of the property.

Counterexamples of this study refer to linear path formatted counterexamples described with CTL, but in our property specifications, a situation about counterexamples poses a problem for our study. This problem is that only one number of counterexample is generated for each violated property. For instance, a counterexample of CTL property$AG(EF(x))$, which means from any state it is possible to reach x, can reveal only one path where x is not reachable. However, there can be other paths from which x is not reachable. Therefore, we defined CTL properties so that each path causing the violation can be captured (See Section 4.4.3 for details).

CHAPTER 3

LITERATURE REVIEW

3.1 Verification of Web Applications with Model Checkers

As it was mentioned in earlier sections, model checkers verify hardware and software systems on high level abstractions of them by checking all possible interactions. Today, model checking method is successfully applied to software applications. Zing [23], SLAM [24] and Cmc [25] are examples of model checkers built for verification of C programs, and JavaPathFinder [26] is a software model checker developed to verify Java programs. Software model checking tools help developers to analyze programs against some desired properties. Current model checkers can verify control-dominated protocol properties [27] as dead-lock freedom (i.e., the system never reaches a state in which it cannot make any progress) and the properties are in qualitative nature [7]. Model checking on software applications brings developers more time to work on more complicated issues by verifying desired properties which can be checked with model checkers.

Model checking techniques are used in verification of web application analysis in many studies. In order to investigate the related studies, we performed a search on Google Scholar[1] with the keywords, “model checking”, “verification” and “web application”. The most related results found with the search are gathered for the literature review. In these studies, we examined that either a new model checker is developed or an existing model checker is used for verification of web applications.

[1] https://scholar.google.com.tr/

First, we briefly explain special model checkers developed for web application verification. MCWEB [28] is an example of model checker developed for verification of static web pages MCWEB is capable of verifying only static navigation properties. In our study we are also interested in verifying dynamic features of web applications by verifying access control of private pages. In 2004, Licata and Krishnamurthi [29] have built a model checker that uses source codes of web applications for automatic modelling. They performed a static analysis with a CFG (Control Flow Graph) which can guarantee that all possible executions are captured in a model. The model can be automatically rearranged by user operation definitions that transform the model to a WebCFG graph, a description of control flow graph with web interactions. The properties for verification check web properties associated with user operations. Unlike [29], we use the models created with dynamic analysis of web applications by means of web crawlers. Deutsch et al.'s model checker, WAVE [30] is developed for web designers to verify web applications using WebML, a high level notation for web application modelling. WAVE works with textual rule-based specifications and a temporal logic property described by user. The properties to be verified is reachability and business process of web applications. Although [30] can verify application specific properties that cannot be checked with other model checkers, using this model checker is not easy for non-expert users, because user has to input a textual description with the property in LTL Also in [28, 29], inexperienced users are not considered for property specifications.

The second category of the related studies covers the studies on verification of web applications with existing model checkers. Since we also use an existing model checker, NuSMV, to verify web applications, related studies in these category are examined in more detail. The relevant studies include formal modelling methods and different property specifications which vary according to modelling methods, model checker type and temporal logic used. Since our aim is to increase the user experience on model checking, the automation level of these studies is also an important aspect for us. In the automation level context, we consider how modelling and property specification is handled. For modelling we have described methods in two types as automated or not automated. In both automated and not automated methods, we give also source of modelling which can be a design model, source code or other type of resource. In order to give in which extend users are involved in property specification process, we provide details about whether the properties are predefined by authors of studies or can be generated by users. Besides automation levels, the presentation of counterexamples are provided.

We first give the related studies gathered in a chronologic order with one paragraph explanations. Then we present the categorization of the studies according to several criteria including modelling source, model checker type, property types, automation levels and counterexample interpretation.

AnWeb, a verification tool for supporting the design of web application, is developed by Di Sciascio et al. in 2002 [31]. In AnWeb model of web applications, called web graph, is designed as a directed graph where pages, links and windows are nodes. The generation of web graph requires the source code information; HTML source codes are modelled as static pages and PHP codes are modelled as dynamic pages. The resulting web graph model is converted into NuSMV input language and the properties are specified in CTL. The properties cover verification of reachability properties as absence of dead-end, generation of dynamic pages and consistency of frame structures. A frame is a HTML tag which is used to contain web contents in a specific part of web page. Consistency of frame structure verifies the hierarchy of a content located in a frame. In order to verify frames, web graph of a web page with frames are formed as a tree having frames

as child nodes. The specifications for web applications are converted to CTL by authors, and the property generation is not automated. Moreover, output of NuSMV is presented to user on browser without any translation.

Haydar et al. introduced an approach for generating a formal model of web applications by observing the external behavior and validating the resulted model [32]. The approach is implemented as a tool called WAUT. In the first step of WAUT, users select the desired attributes for verification. Then, the model is built automatically by means of observations of HTTP request and responses. The resulted model describes the navigation of web application including the interactions with the browser (e.g. back button). Conversion to a formal model is performed by transforming the model to XML-Promela language for aSpin, an extension of SPIN model checker. The web attributes, which are selected in the first step, are used in formulating the web attributes to LTL language. After verification process, the output of model checking is not transformed to any other version.

In 2005, Di Sciascio et al. developed a method for converting UML diagrams to formal models in order to be verified by model checker [33]. The conversion is made by means of a component named XMI2SMV which translates the UML diagram in XMI format to Web Application Graph (WAG). With this model, Di Sciascio extended the web graph in AnWeb by defining actions as a node in web graph. Consequently, in WAG, pages, links, windows and actions represented as nodes of the graph. The WAG can be converted to NuSMV input language automatically.

Castelluccia et al., developed the tool WaVer by extending the WAG of [33] with the addition of access policies [34]. WaVer verifies UML-design of web applications with NuSMV model checker. The model to be verified is converted from UML diagram by XMI2SMV component, and additionally, access control policies of authorized users and administrators are added to WAG model. The properties are formulated in CTL by authors. WaVer enables users to verify web application by a single automation process. However, property definition is not allowed to users. In the study, it is mentioned that the output of NuSMV execution allows to locate errors and make adjustments on design of web application, but the output is not converted to a more understandable version for demonstrating the errors on the design model for users.

Knapp and Zhang [35] used model checking for validating and verifying design of web applications. The model of a web application is created by integration the navigation and business logic models of web application. The models are in the UML-based Web Engineering (UWE) form, which is an extension of UML. SPIN model checker is used for validating the model. The properties are defined in LTL and check navigation and business processes of web applications. Property generation with respect to specifications of users and automated modelling are not considered in this study.

In 2007, Brambilla et al., extended the WAVE model checker [30] by adding a front-end using Model Driven Approach (MDA) [36]. In this approach, the design model inputted to WAVE is specified with WebML notation. To formalize model, a set of transformation process is used to covert model to WAVE model. Users do not have to input textual rule-based specification and a temporal logic property as in WAVE. Modelling is made manually in WebML notation. Users then verify the model with LTL formulas. Moreover, authors present a way for converting user specifications to LTL properties with a user interface [37]. In this study, interpretation of output

of model checking is also considered. As other model checkers produce counterexamples, WAVE also outputs a feedback of false-resulted properties. Since users who have designed the system in WebML may not understand an output generated with respect to variables of WAVE models, the feedback is presented to users with elements of model in WebML, which is the notation used while creating the model of web application by users.

Most of the previous studies focus on modelling, but Miao and Zeng emphasis on property generation [38]. The properties are generated automatically as CTL formulas with respect to a design diagram given by the user in the form of Object Relation Diagram (ORD), a directed graph representing web pages, software components, their relationship types, and relations, which is proposed by Kuang et al. [39]. The model to be verified, implementation model, is described manually in Kripke structure. Within this approach, consistency between implementation and design model is checked. Verification is completed by SMV model checker. In [38], the design diagram constitutes the base for production of properties. However, users are not directly included to property generation process, since they cannot select the attributes to be checked. Also interpreting counterexample of SMV for inexperienced users is not considered in [38].

In 2007 Guerra et al. developed AToM tool which automates the verification of security policies of web applications [40]. They use Ariande Development Method (ADM), a web engineering method for modelling the web application [41]. The source model in ADM, is converted to Petri net formalism to apply model checking against the properties in CTL. While verifying, main focus is on access control properties, but static and dynamic features can also be verified. Corresponding CTL expressions for given web model is built internally in this method. Results of model checking also displayed on the design model. Also the result depicted in the design model is summarized in a dialog box.

Han and Hofmeister's method FarNAV verifies adaptive navigational properties of web applications by means of NuSMV model checker [42]. The adaptive navigation properties specify navigation features dependent on session and role types of users. In FarNAV modelling is based on statecharts unlike other studies which are based on graph-structured models. Authors use parallel sub-states to represent learned navigation patterns of visitors. In the main sub-state a web page is represented as one state and a link between pages represented as transition. The properties, specified in CTL, check the reachability of web application against the mode of the user. In FarNAV, modelling and property definition process is handled manually. There is no specification about counterexamples transformation for increasing their understandability.

In 2010, Hallé et al. developed a plugin for web applications that prevents runtime navigation errors [43]. The model of web application, Navigation State Machine (NSM), is defined as state machine like FarNAV. Given NSM specifications in XML format, the plugin monitors the web application with respect to specified ACTL properties. NuSMV model checker verifies these properties and the plugin restricts the control flow, if the model checker falsified the related property. This study is different from previous ones in terms of aim of usage; the goal of this study is to restrict an unwanted navigation path before it happens. In this way, it protects the user from a security gap sourced from unexpected interactions that is not covered in web application implementation.

Homma et al, studied on a method of modelling and verification of web application design with model checkers [2]. They describe web applications with two separate models; internal state and page automation models. Page automation represents the navigation between web pages and internal state automation represents the internal transitions of application. These two models are expressed in Promela, input language of SPIN model checker. In order to verify the model in Promela, reachability properties defined by authors in LTL. The errors in web application design are aimed to be detected in this way. Automation of the modelling is not included and increasing user experience is not considered in this study.

3.1.1 Classification of the Related Studies

In this section, we give the classification of the related studies discussed in the previous subsection. First, we define five categorization criteria and then we present the classification of the related work in Table 2.

3.1.1.1 Categorization Criteria

The categorization of studies explained in preceding section are described based on five main criteria which are modelling source, model checker, property type, automation level and counterexample interpretation.

Modelling Source is a criterion for indicating from which source the model of the web application is extracted for verification purposes. The modelling can be begin from source code, design model or by monitoring the external behaviors of web applications.

Model Checker indicates which model checker tool is used.

Property Type describes the property specifications used in model checking. The categories for property types are determined according to survey of Alalfi et al. [44] in which the desired properties of web applications are explained in three categories as navigation, content and behavior. Navigation and content categories split into two subcategories as static and dynamic features. Moreover navigation category includes one more subcategory called interaction. Interaction navigation category is about the navigation properties which occur outside the boundaries of web application (e.g., interactions with the web browser). The last one, behavior, is divided into two subcategories as security and instruction-processing properties. The property types of related studies which are also included in Alalfi el al.'s survey, remain as Alalfi el al.'s categorizations. Our categorization covers static and dynamic navigation, static and dynamic content and security properties. Static navigation properties check reachability, absence of dead endsor consistency of a frame structure. Dynamic navigation properties cover the navigation which depend on user input or other behavior. Static content properties check the existence of web pages, links or other components of web application. Dynamic content properties check the existence of dynamically created objects. Security properties include the assertions of access control or session/cookies mechanism.

In Brambilla et al.'s study [36] the verification concerns do not match with a specific category of Alalfi el al.'s categorizations. They use a special model checker built for web analysis, WAVE

[30] which is capable of checking all of common temporal logic properties and not limited to syntactic analysis. Therefore, more complicated specifications as business process descriptions can be checked with [36]. The property type categorization of [36] is defined as business process properties. In our categorizations, the high level descriptions about property types are given, since we provide detailed explanations for each study in the previous section.

It is important to mention that the differentiation of property types depends on the extent of models. For example, models that cover dynamic features as user roles, input variables, enable to specify properties that can check dynamic features.

Automation Level specifies whether the modelling is automated or not and whether properties are already defined by authors of the study or automatically generated by user specifications. Also

In Table 1, the description of MCWebSoft, in terms of the categorization criteria we defined is given. It is important to remind that, although we make our implementation as capable to accept the web crawler tools' outputs as web models, modelling can be done manually too. Moreover, properties are described in two main categories as navigation and access control. Navigation properties cover reachability of homepage, absence of dead end and link consistency between web pages. We categorize navigation property as static and dynamic navigation, since the web crawlers we used for modelling performs dynamic analysis and they execute the web application with random or defined inputs while crawling. Further, access control properties are categorized as security properties. Besides, another feature of our study is transformation of counterexamples. Counterexamples produced are replayed on web browser to provide a better feedback of execution for inexperienced users.

Counterexample interpretation criterion is for indicating whether the counterexamples generated from model checker is interpreted to a more understandable version. Also, if such an interpretation is performed, the way of the transformation is specified. We denote the studies in which the counterexample interpretation is not considered with the 'None' word.

Table 1: Categorization of MCWebSoft

Modelling Source	Model Checker	Property Type	Automation Level	Counterexample Interpretation
Web Crawler tools, Crawljax and Micro-Crawler	NuSMV	Static and dynamic navigation and security	Modelling is automated. Properties can be selected by user.	Replay of counterexamples

In Table 2, categorization of related studies are given.

Table 2: Categorization of the related studies

Study (Author, Year)	**Modelling Source**	**Model Checker**	**Property Type**	**Automation Level**	**Counter example Interpretation**
Di Sciascio et al., 2002, [31]	Source code	NuSMV	Static and dynamic navigation	Modelling is automated. Properties are predefined.	None.
Haydar et al., 2004, [32]	HTTP requests/responses	aSpin	Static and dynamic navigation, static content and interaction navigation	Modelling is automated. Properties are generated with respect to user selections.	None
Di Scascio et al., 2005, [33] Castelluccia et al., 2006, [34]	Design model in Unified Modeling Language (UML)	NuSMV	Static and dynamic navigation	Modelling is automated. Properties are predefined.	None
Knapp and Zhang, 2006, [35]	Design models of navigation and business logic in UWE	SPIN	Static navigation and security properties	Modelling is not automated. Properties are predefined.	None
Brambilla et al., 2007, [36]	Design model in WebML notation	WAVE	Business process properties	Modelling is automated. Properties are defined by user.	The counterexamples are interpreted to users with elements of the design model.
Miao and Zeng, 2007 [38]	Source code	SMV	Static and dynamic content and navigation. And security properties	Modelling is not automated. Properties are automatically generated.	None

Table 2 (continued): Categorization of related studies

Guerra et al., 2007, [40]	Design model defined with, the Ariadne Development Method	Petri nets	Static and dynamic navigation, security properties	Modelling is automated. Properties are predefined.	Results of model checking are given on the source model and in a dialog window.
Han and Hofmeister, 2010, [42]	Navigation model based on statecharts	SMV	Static and dynamic navigation and dynamic content	Modelling is not automated. Properties are predefined	None
Halle et al., 2010, [43]	Navigation State Machine (NSM)	NuSMV	Static, dynamic and interaction navigation and security properties	Modelling is not automated. Properties are predefined.	None
Homma et al., 2010, [2]	Page and internal state automata models	SPIN	Static and dynamic navigation properties	Modelling is not automated. Properties are predefined.	None

As it can be seen in Table 2, web crawlers have not been used as modelling source in related studies. However, we mentioned in Section 2.2 that in [14] and [20], web crawlers are examined for their adaptability for model checking, and Demarty et al. developed a method for existing web crawlers in order to redefine the navigation maps for model checking [20].

Model checker that we used in this study, NuSMV, has been also used in other studies of web application verification. Therefore, converting the web application models to NuSMV models is covered in these studies which are [31, 33, 34, 43]. Property types that we cover in this study have been also used in several previous studies; [2, 3, 31] cover the reachability properties of our study. Moreover, [33, 34, 40, 42] include properties of access control, which is also considered in this thesis.

As for automation level, in related studies, mostly, properties are specified by the authors. Only in [34, 36, 38], users can be involved in property generation. In [34], users select the attributes they want to verify, then properties in temporal logic are automatically produced. In [36], users specify

the specifications by a virtual interface and properties are produced according to these specifications in LTL. In [38] users input a design model, describing how the system should behave, then by the design model temporal logic properties are generated automatically. In our approach, reachability properties are generated automatically. Also for checking link consistency between two states and access control of a state of web application, users select the web pages, states to be verified with the guidance of user interface, then MCWebSoft generates the properties in CTL. Moreover, we can see from Table 2 that most of the studies include the automation of modelling [31, 32, 33, 34, 36, 40], but automated modelling with web crawlers is not used.

In Table 2, only [36] and [40] consider the interpretation of the counterexamples by giving the output as design model specifications. In our study, we interpret counterexamples by converting the traces of them into a sequence of clicks starting at the initial page and reproducing the error on the actual web site. By this way, counterexamples can be understood by both designers and users of web site. We give literature review about interpretation of counterexamples in Section 3.2 by covering literature not only about web application verification but software model checking studies in general.

3.2 Interpreting Counterexamples

In Section 2.3, description, usage and structure of counterexamples are introduced. In this section, we do not examine counterexamples, but give examples of studies that interpret the counterexamples for increasing users' comprehension.

Counterexamples provide users an important feedback about violated property by giving the sequence of states causing the violation. However, there are several situations that complicate the understandability of this important outputs and causing users to spend considerable time for examining the counterexamples to find detected errors on real application. One of the situation that makes difficult to understand counterexamples is the complexity of state variables defined in the model for model checker. In counterexamples variables are displayed as they are defined in model checker input language. However, if users do not know the variables defined in model checker language, counterexamples can be confusing. To solve this problem, [36, 40] applied a transformation process from output of model checker to design model. However, this may not be enough for users to understand counterexamples, since model checking large systems can cause lengthy counterexamples for the large number of states and transitions. To decrease such complexities, Huang et al., grouped the errors produced from same initial state by using Bounded Model Checking and provide descriptive reports [45]. Also, Ball et al., added a special algorithm to SLAM [24] to report the cause of errors from error traces in counterexamples [46]. In [46], correct traces are used to determine where the errors are localized in a program.

In our study, we choose to replay the scenarios of counterexamples on real application with automatically generated executable scripts in order to help designers to localize and determine errors more easily. The idea for replaying counterexamples is attempted before in different studies. For example, Bandera [47], a tool using existing model checkers, SPIN and SMV, to analyze the Java programs, has a component to save the execution traces in counterexample and show the values of states for each step by giving moving backward and forward options to users. For the web model checker built by Licata and Krishnamurthi [29], the output of model checking is made suitable to be converted into the format of the WebVCR [48], a record and replay tool, to

enable developers to watch error traces, by giving traces as sequences of web pages and operations. VeriWeb [49], a dynamic navigation testing tool which uses VeriSoft for exercising all possible execution paths in a nondeterministic manner. In VeriWeb once errors are found, they are logged by VeriSoft and the scenarios of errors are saved in a separate file, as a model checker generates counterexamples for each violated property. Authors state that, the recorded error sequences are suitable to be replayed by WebVCR. The difference of our approach from VeriWeb and Licata and Krishnamurthi's study is that we do not only provide automated transformation of counterexamples into executable script, but also replay this scenarios without forcing users to use a record and replay tool to watch the error traces on the web application

3.3 Testing Web Applications with Model Checkers

In this section, some of the studies in testing web applications with model checking method are given. Although, the aim of our study is not testing, the concerns of generating a model of web application and converting the model to model checker language are similar to our study.

Testing with model checkers is an approach for web applications in the context of model-based testing. In testing with model checkers, counterexamples which are evidences of property violations, are used to generate test cases. By this way, test case generation problem is adapted to model checking. A well-known method for test case generation with model checkers is to create counterexamples for trap properties within a coverage strategy [50, 51, 52]. Trap properties are negation of temporal logic properties which describe the desired characteristics of a test case. Another approach uses mutation-based techniques to create counterexamples as test cases by applying mutation to model or temporal logic properties [53].

In [50], Törsel used NuSMV model checker to generate test cases. A domain-specific language is defined for specifying models of web applications. The model is described in XML and converted to NuSMV language for model checking. The application specific information about the tested web application such as XPath expressions, HTML tags, are put in a separate format, but not in the domain specific language. This is done for maintaining the possible changes in DOM structures from one file. As for our approach, we do not need to map the information about HTML elements, since the models of web applications are taken from web crawler output and the presence of HTML content is dependent on the crawler output. In case of any change in the web application, user should crawl the web application again and run MCWebSoft with the new output to update the error scenarios for a reliable result.

Törsel [50] converted the model specified in domain-specific language to NuSMV model by using nested relations and creating new variables for model checker input language. We also use these techniques while building NuSMV model by creating a trap state and using nested relations. (see Section 4.3 for detailed information).

In [51], a formal model for demonstrating navigational behavior of web applications, which is used to generate test cases is proposed. An Object Relation Diagram (ORD) is employed to model the web applications. In this study, a web application is modelled as a directed graph where set of web pages and software components are represented as nodes and the relations between these nodes are defined as edges. Trap properties are generated with respect to edge and node coverage

criteria and checked with model checker to generate counterexamples which are converted to test cases.

In [52] a technique for modelling web applications as Kripke Structure and optimizing the number of counterexamples which are converted to test cases are developed. The model of web application is constructed as Kripke structure with states corresponding to web pages or components and state transitions corresponding relations between web pages. In order to generate counterexamples, state, transition and transition composition coverage are expressed as trap properties. The number of resulted test cases is reduced by a special algorithm to avoid redundant test cases

CHAPTER 4

METHODOLOGY

This study aims to enable the users with no background in formal methods to use model checking on verification of web applications. Our approach consists of the following four main steps:

- Modelling web application as intermediate web model
- Generation of model in NuSMV and temporal logic properties
- Model checking with NuSMV
- Converting counterexamples to executable error scenarios

Model checking process is handled by NuSMV model checker which is added to our tool externally. Other steps are explained in subsections 4.2, 4.3, 4.4 and 4.5. The detailed workflow containing all details about the methods is given in Figure 6.

Modelling the subject web application as intermediate web model is the first step of our approach as shown on the left side in Figure 6. In order to build the model, first a base model is taken from a web crawler. Then, this model is converted to intermediate web model or a web model is defined manually as an intermediate web model from user. The basic unit of our model is called webstate. Webstate corresponds to state of DOM structures determined by changes aroused from user interactions with web applications. The second unit of model is element. Elements represent clickable page element of web application. The last unit is edge. Edge represents transition relations between webstates through elements. In intermediate web models, users can modify components of the intermediate web model which are webstate, element and edge. The detailed explanation of these terms are given in Section 4.2.

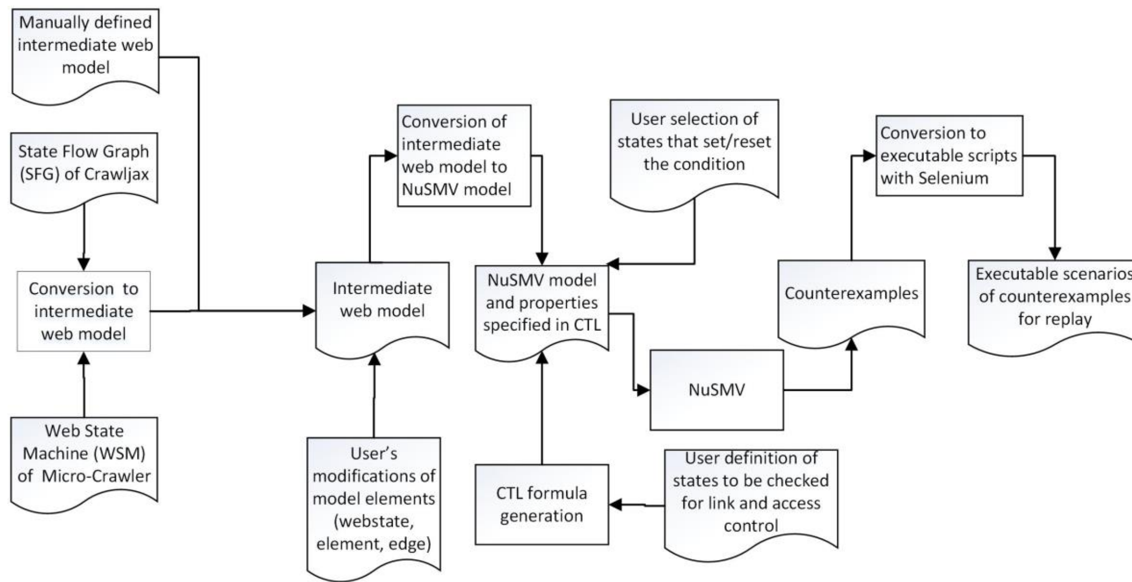

Figure 6: Workflow of the approach

The next step in the workflow is property and NuSMV model generations. Our tool generates CTL formulas without user intervention for navigational properties, which check if there are no dead ends and homepage (index) is reachable from every state of the application. The generation of the access control properties and the navigational properties checking reachability between two states (i.e., link consistency) require user guidance (Bottom left in Figure 6). The next activity in Figure 6 is running the model checker and parsing the resulting counterexamples. The final activity is the transformation of the counterexamples into executable scenarios to run with Selenium and animate the error cases on a browser with a dialog box explaining the violated property.

In the remainder of this chapter, to concrete the details about workflow steps, the proposed methodology is explained with a toy dynamic web application. The chapter starts with the explanation of this running example. Then methodology of our approach is given with the running example application.

4.1 Running example

This artificial application consists of twelve source files where six of them are PHP files and one of them contains functions in JavaScript. It is designed to illustrate the errors in navigation and access control functionalities that is targeted in this study. This web application contains one Ajax-enabled page, one login page, several private pages that needs to be accessed only after authentication of user, two dead end pages, and one page with broken link. The representation of the model of web application is given in Figure 7. This model is driven from SFG of Crawljax because of login operation that requires form filling only supported by Crawljax. In this figure the dashed line represents the transition Crawljax missed. This transition is added manually via the user interface of our tool.

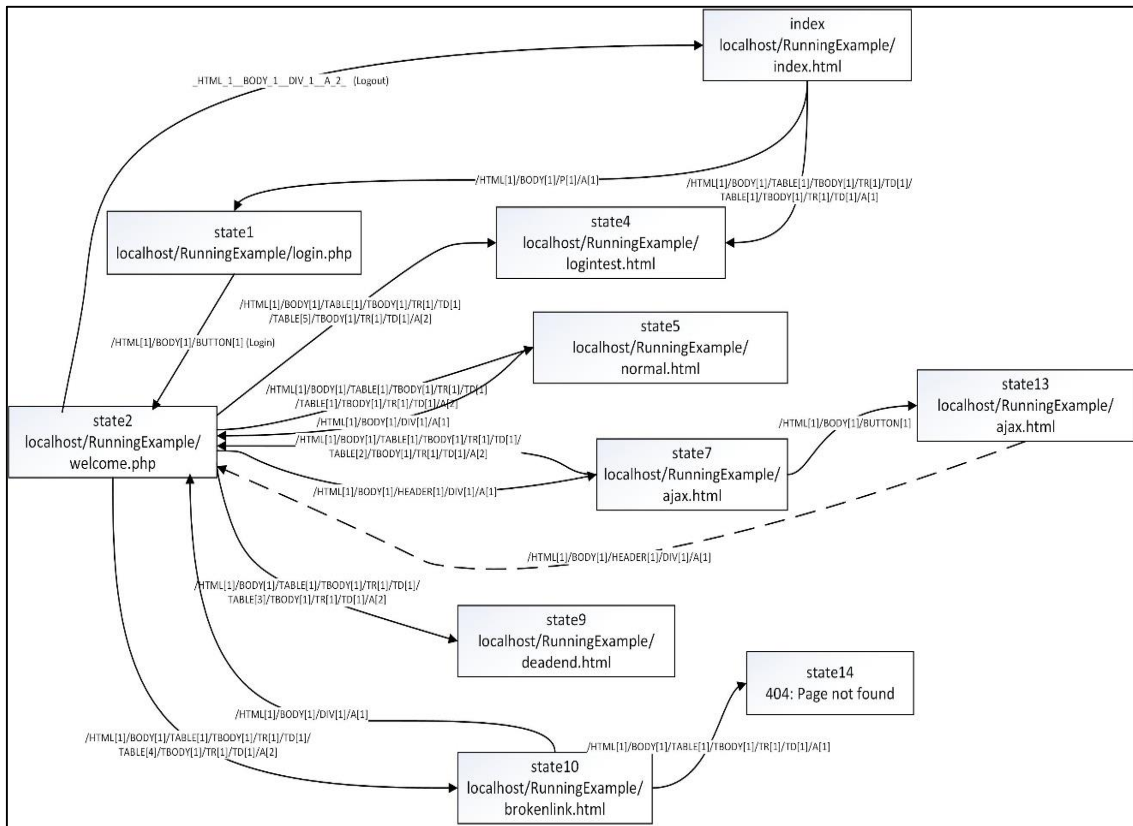

Figure 7: A representative model of the running example

As Figure 7 shows, there are 10 webstates in this example. The webstate named ‘state1’ performs the login activity. The XPath expression on the edge originating this webstate locates on a button element in the DOM tree. In this example, only the index page is designed to be publicly accessible. The webstate, ‘state4’ is intended to be a private page, but it is accessible without authorization. We expect that this error will be found by the access control property. The webstates, ‘state7’ and ‘state13’ share the same URL. The transition between these states is due to DOM manipulation with a JavaScript event. The webstate ‘state10’ represents a page that includes a broken link. The webstates, ‘state9’ and ‘state4’ are dead ends with no clickable elements. These webstates also violate that the index page is reachable from everywhere in the web application. This application includes login operation so we extract the FSM using Crawljax. All states were identified by Crawljax, but the transition between the webstates, ‘state13’ and ‘state2’ is not recognized. The webstate, ‘state13’ is created after an AJAX event is fired, which does not reload the whole page but changes a part of the DOM tree asynchronously. In such cases, Crawljax exercises the clickable elements in the changed part only. Since the link element in the webstate, ‘state13’ which directs it to the page in the webstate, ‘state2’ does not change after an AJAX event from the webstate, ‘state7’, Crawljax missed the link between the webstates, ‘state13’ and ‘state2’. The omission of this link would cause a false alarm stating that the webstate, ‘state13’ is a dead end. Therefore we add this link to the model manually by means of MCWebSoft.

4.2 Modelling Web Application as Intermediate Web Model

Modelling web applications starts from extracting a model of web application from web crawlers. The output of web crawlers, SFG for Crawljax or WSM for Micro-Crawler, is transformed into intermediate web model. Moreover, users can manually define the web model directly, as shown top left corner in Figure 6, without using web crawler outputs. The intermediate web model refers to a finite state model in XML based language and it is designed to cover common elements of WSM and SFG.

An intermediate web model is a 4 tuple directed graph $M = (W, E, \gamma, w_0)$ where,

- W is the set of webstates which are the DOM trees generated by the web application. A webstate consists of a name and a URL. Note that any two webstates can have the same URLs, but different names, since there could be more than one DOM trees with the same URLs due to partial reloading of a web page.
- E is the set of clickable page elements that change the current webstate when clicked on. An element in E is identified by an XPath expression that locates the element on a page.
- $\gamma: W \times E \rightarrow W$ is the transition relation, which maps each (w, e) for $w \in W$, $e \in E$ to a webstate, $w' \in W$. If we define this function as a curried function then $\gamma: W \rightarrow E \rightarrow W$ such that
  - $\gamma(W)$, returns all transition relations, $\gamma: E \rightarrow W$ from the set of webstates, W
  - $\gamma(W)(E) \subseteq W$, returns webstate $w' \in W$ which also equals to $\gamma(w, e)$, for $w \in W$ and $e \in E$
- w_0 is the initial state of application which is called 'index'.

Intermediate web model is created with variables that are also in SFG and WSM. The values which fill the variables of the intermediate web model is taken from SFG or WSM depending on the user choice for web crawler tool. The structure of these state machines are explained in sections 2.2.1 and 2.2.2. Let us repeat the definitions of these state machines here for convenience.

SFG is a directed graph, $< r, V, E, L >$ where

- r, is the initial state of application,
- V is the set of states of web application and symbolizes the vertices of the graph,
- E is the set of directed edges between vertices in graph and represents a relation between two states if and only if there exists a clickable element connecting them,
- L is a labeling function which assigns a label from a set of event types and DOM element properties to each edge.

WSM is a directed graph, $< V, E, \delta, v_0 >$ where

- V is the set of DOM trees in the application,
- E is the set of DOM paths,
- δ is a function, $\delta: V \times E \rightarrow V$ such that $\delta\ (v, e) = v'$, if clicking an element defined in the path $e \in E$, in $v \in V$ results in the DOM tree $v' \in V$,

- v_0 is the initial DOM tree.

By looking at these explanations, the set of states of SFG and the set of DOM tree of WSM correspond to the set of webstate of intermediate web model. The set of elements of SFG and the set of DOM paths of WSM refer to the set of elements of our intermediate web model. The edge construct of our intermediate web model which is shown with γ function corresponds to functions L of SFG and δ of WSM. Also the initial value of webstate, w_0, exists in in both SFG and WSM as the first state and the first DOM tree respectively.

```
1      <?xml version='1.0' encoding='UTF-8' ?>
2      <webstates>
3       <webstate>
4              <name>
5             index
6             </name>
7             <url>
8             http://localhost/RunningExample/index.php
9             </url>
10     </webstate>
       …
83     </webstates>
84     <elements>
85     <element>
86            <xpath>
87            /HTML[1]/BODY[1]/P[1]/A[1]
88            </xpath>
89     </element>
       …
150    </elements>
151    <edges>
152    <edge>
153           <from>
154           index
155           </from>
156           <to>
157           state1
158           </to>
159           <xpathOfElement>
160           /HTML[1]/BODY[1]/P[1]/A[1]
161           </xpathOfElement>
162    </edge>
       …
306    </edges>
```

Figure 8: The intermediate web model of the running example (Excerpt)

A part of the sample intermediate web model specification in XML format is shown in Figure 8 (see Appendix A for the complete version). The first lines declare the set of webstates, W. Then between the lines 84 and 150, E, the set of clickable page elements are introduced with the XPath expressions identifying them. Lastly, the transition relation is declared between the <edges> and

</edges> tags (lines 151-306). Each edge includes a transition $\gamma(w_i, e) = w_j$ where <from> tag specifies w_i, <to> tag specifies w_j and <XPathOfElement> specifies e which is the XPath expression that selects the element of w_i that needs to be clicked on to trigger the transition. Currently, a transition occurs only with click events and back button is not considered. This restriction results from web crawlers used. Including back buttons would be a simple extension but the user had to manually provide this information.

After the transformation to intermediate web model, users can refine it to include the webstates elements and edges, the transitions that are missed by the crawlers. The next activity in the workflow in Figure 6 is the property and NuSMV model generations which are explained in Section 4.3 and Section 4.4 respectively.

4.3 Generation of Model in NuSMV

Model checkers accept system models in their own specification language or in a restricted set of a programming language. In this study, we do not want to depend on a specific model checker or on a specific web crawler. The intermediate web model serves the independence in both aspects. In the previous section, the semantics of the intermediate web model and modelling web applications as intermediate web model is explained. This intermediate web model needs to be translated to the input language of the checker, NuSMV, to be verified by the model checker.

The transformation from intermediate web model to NuSMV model is not only syntactic conversion. The web model in NuSMV contains guards on transitions and a trap state. The intermediate web model is capable of representing the navigation inside the web application. However, it does not contain the information required for investigating access control and authorization errors. Moreover, the transition function defined in the intermediate model does not fit to type of transition functions that NuSMV model accepts. Therefore, transformation of web model to model checker model needs several additions.

To investigate access control errors, in the NuSMV model, we use Boolean variables to encode the logged in information. These guard conditions are added to NuSMV model and not exists in the intermediate web model. They can be used to specify role based authorizations. Another addition to NuSMV model is the trap state. Recall that a transition relation in Kripke structure must be left-total. However, the transition relation in the intermediate web model is not a left-total transition relation. A trap state dead_end_webstate, a webstate variable, is added to NuSMV model to enable that each webstate has a next webstate and to transform transition relations in intermediate web model as left total. Also, in intermediate web model, each transition is triggered with a page element. Therefore, we also introduce null_element, an element variable, to enable transitions to the trap state. Adding new variables to NuSMV model was also used in [50].

In order to apply model checking, the model to be verified should be in the form of Kripke structure [8]. Now we give the formal definition of our NuSMV model, which is a Kripke structure. Since the transformation to model checker model starts from intermediate web model, first we recall that; in Section 4.2, the intermediate web model is given as a directed graph, $M = (W, E, \gamma, w_0)$. In M , γ defines a transition relation, $\gamma: W \times E \rightarrow W$, from set of webstates W, and

set of page elements E to set of webstates that are produced after clicking page elements on webstates in domain set, such that $\gamma(W)$, returns all transition relations, $\gamma: E \rightarrow W$

Given an intermediate web model $M = (W, E, \gamma, w_0)$, a NuSMV model is a 4-tuple $K = (G, S, S_0, T)$ where,

- G is the set of guard conditions $\{g1, g2, \ldots\}$ which could be an empty set,
- $S \subseteq W \cup \{dead_end_webstate\} \times E \cup \{null_element\} \times \Pi^{|G|}\{True, False\}$
- $S_0 \subseteq S$ is the initial state where $S_0 =< index, null_element, \bot >$
- $T: S \rightarrow S$ is the transition relation such that
  Let $w, w' \in W, e, e' \in E$ and $l, l' \in \Pi^{|G|}\ \{True, False\}$. Let also $s, s' \in S$ where $s =< w, e, l >$. If $T(s) = s'$ then $\gamma(w) \neq \emptyset$ and $s' =< \gamma(w, e'), e', l' >$ or $\gamma(w) = \emptyset$ and $s' =< dead_end_webstate, null_element, \bot >$.

In the NuSMV model K, G, the set of guard conditions, would be an empty set, if a user did not define a guard condition. Guard conditions are for adding access control properties, which check access of pages requiring authorization of users to be reached. If G is an empty set, only navigation properties can be verified. Moreover, we assume that guard conditions constitute a set, since users could differentiate admin and regular authorization by using two guard conditions such as *loggedIn* and *adminLoggedIn*. However, in our implementation, there is only one guard condition which checks if the regular user logged in or not. In this setting, marking the states where regular user logs in or logs out is specifying the states where the guard condition is set or reset.

In this model, a state $s \in S$ contains the information of the element that brings the system to the state s. Since the initial state and the trap state has no such element, the element in the initial state and the trap state is set to null_element.

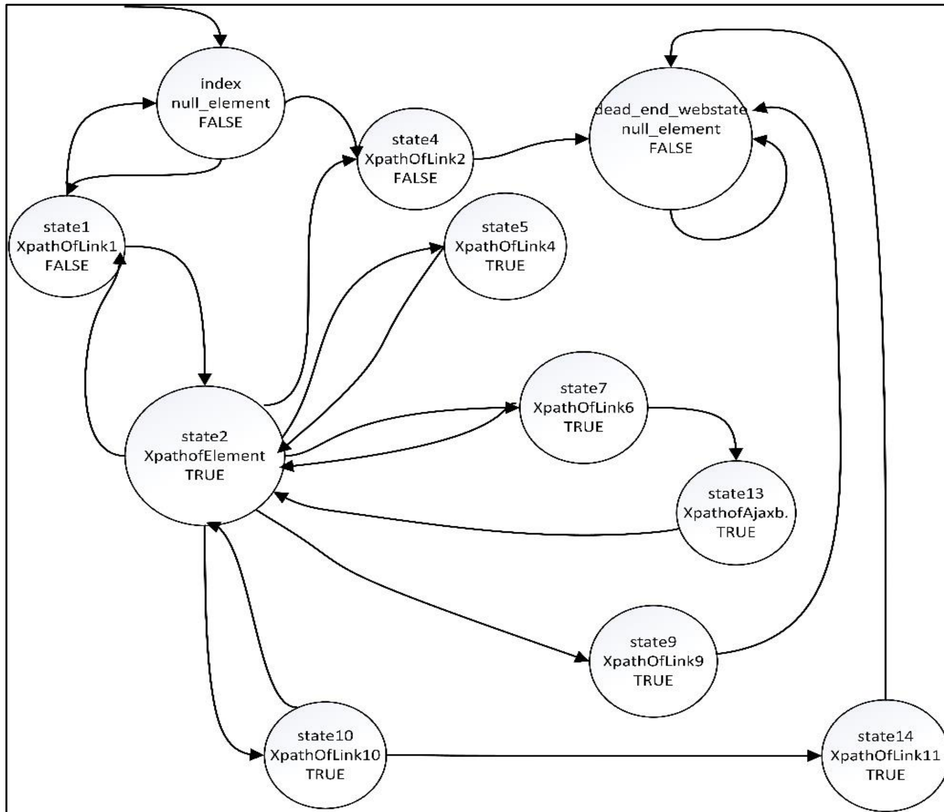

Figure 9: Kripke structure representation of the running example

In Figure 9, the representation of Kripke structure *K* of the running example is given. Circle shapes represent states and arrow shapes represent transitions. Each state in the figure has a webstate, which are depicted with their names as ‘index’, ‘state1’, ‘state2’, etc. an element which are displayed as ‘XpathOfLink1’, ‘XpathOfLink2’, etc. and a guard condition represented with Boolean variables. Moreover, the state (state2, XPathOfElement, TRUE), represented in the Kripke structure in Figure 9, is a symbolic state where XPathOfElement represents all elements which causes to state having the webstate named ‘state2’ to happen.

```
1      MODULE main
2      VAR
3      login:boolean;
4      webstate:{index,state2,…, dead_end_webstate};
5      element:{_HTML_1__BODY_1__P_1__A_1_,…, null_element};
7      ASSIGN
8      init(login):=FALSE;
9      init(webstate):= index;
10     init(element):= null_element;
11     next(webstate):= case
12       webstate=index &
next(element)=_HTML_1__BODY_1__P_1__A_1_:state1;
…
27       webstate=state13 &
next(element)=_HTML_1__BODY_1__HEADER_1__DIV_1__A_1_:state2;
28       webstate=state1 & next(login) = TRUE &
next(element)=_HTML_1__BODY_1__FORM_1__INPUT_3_:state2;
…
29       TRUE: dead_end_webstate;
30     esac;
31     next(element):= case
32        webstate=state1  : {_HTML_1__BODY_1__FORM_1__INPUT_3_};
…
39        webstate=state10:{_HTML_1__BODY_1__DIV_1__A_1_,
_HTML_1__BODY_1__TABLE_1__TBODY_1__TR_1__TD_1__A_1_};
40       webstate=dead_end_webstate:null_element;
41       TRUE: element;
42     esac;
43     …
44     next(login):= case
45       webstate=state1 & next(element) =
_HTML_1__BODY_1__FORM_1__INPUT_3_ : TRUE;
46       webstate=state2 & next(element)=
_HTML_1__BODY_1__DIV_1__A_2_:FALSE;
47       TRUE: login;
48     esac;
```

Figure 10: The NuSMV model of the running example (Excerpt)

In Figure 10, the excerpt of the web model for our running example, in the input language of NuSMV, is given (see Appendix B for the complete version). This code is generated from intermediate web model of running example which is given in Figure 8. The NuSMV model, depicted in Figure 10, is generated automatically in the input language of NuSMV. The NuSMV input consists of a main module definition and a number of CTL specifications. The CTL specifications are not shown in Figure 10, since they are discussed in Section 4.4 in detail. In this NuSMV specification, there are three variables in the main module, which are webstate, element and the guard condition, login. The domain of webstate (line 4) consists of the name attributes of the webstates defined in intermediate web model specification (see Figure 8). The domain of element consists of the modified XPath expressions that are defined with the <XPathOfElement> tags in <elements> definition in our modelling language. The brackets and backslashes in the XPath expressions are replaced with underscore to conform the NuSMV syntax rules (line 5).

The part starting with the ASSIGN keyword declares first the initial values of the variables (lines 7-10). These lines realize that the initial state of the system is $S_0 =< index, null_element, \perp>$. The init keyword is used for setting the initial values. Initially, element is set to null_element indicating that no page element has been clicked. The variable webstate is set to index which is the name of the first page when the web application is loaded. The guard condition login is set to false, meaning initially there is no guard to access initial value of webstate which is index.

After assigning initial values, the transition relations are declared by defining how to set the next values for each variable using a case expression. Each leg of case segment consists of a condition, semicolon, and a value. NuSMV chooses the value whose case condition is satisfied by the current execution state. The last case condition is TRUE for defining the default case. Consider the next relation for webstate between lines 11- 27. This part is generated using the information between the <edges> and </edges> tags in the intermediate web model specification. Recall that, an <edge> node contains <from>, <to> and <XPathOfElement> child nodes. Based on this information, a case condition is formed using the webstate name in the <from> tag and the modified XPath expression that matches the element clicked on the current page. The next value is formed using the webstate name in the <to> node of the edge. Here the case condition checks the next value of the element which is consistent with our semantics. The element in a state holds the information of which page element has brought the system to that state. Therefore, the variable element holds the last clicked page element which resides in the previous webstate. Therefore, we have inserted the next relation of element variable in the case clause for webstate, which is also used in [50]. Finally, the default case in line 29 defines that webstate becomes dead_end_webstate which is the trap state in the model.

The next value of element is randomly selected from the clickable elements in the DOM tree of the current webstate (lines 31 - 39). If current webstate is dead_end_webstate, next element is set to null_element to ensure that the system stays in the trap state (line 40). The default condition for the element variable is set to its current value with the expression in line 41. That is, if none of the case conditions is satisfied by a value of the element variable, it does not change its value.

In this example there is one guard condition. The user has defined the guard condition as login using our tool. The user also marked that login is set when the button element in the webstate, ‘state1’ that matches /HTML[1]/BODY[1]/FORM[1]/INPUT[3] is clicked. This information is implemented in line 45 in Figure 10. Note that this information also added the line 28 in next

relation of webstate. Line 28 defines that after a successful login the system will be in 'state2'. The user marked that login is reset when the element matching the expression /HTML[1]/BODY[1]/DIV[1]/A[2] on the webstate, 'state2' is clicked. This XPath expression locates a logout button in the DOM tree called 'state2'. This information is automatically transformed to the line 46 in the NuSMV code. By default, the value of the login variable is its current value (line 48).

4.4 Properties

In this study, properties are defined with the purpose of verifying or falsifying navigation and access control specifications of web applications. The properties are described under four categories which are index reachability, absence of dead end, link consistency, and access control. Violation of these properties leads to generation of counterexamples, and this way, detection of errors. In this section, we first give detailed explanations of the error categories aimed to be found. Details about the property specifications as CTL formulas are given in Section 4.4.3.

4.4.1 Navigation Errors

Navigation errors arise from incorrectness or lack of static navigation features as static links. Most of the early works focus on these type of errors by accepting web applications as hypermedia applications. Absence of broken links, reachability, consistency of frame structure and features related to estimating the cost of navigation are some of the criteria checked in the scope of navigation of web applications [44]. In our study, two sub categories are checked within the navigation title which are

- absence of dead ends,
- reachability.

De Alfaro [3] classified these type of properties as necessities of a good design at global level and used model checking to check them. We consider dead end pages as the web pages which do not have a next page to go within the web application. The web pages which can be navigated with only back button is regarded as dead ends. For instance, web pages with special extensions such as pdf, jpg are dead end pages, since user has to use back button for exiting these type of pages Also, the web pages with only external links are considered as dead ends, since there is no internal link to navigate web application. Moreover, error pages displayed with HTTP status code as 404 (Not Found) are considered as dead ends, since with these type of pages users have no choice, but pushing the back button to return web application. In the second subcategory, reachability of homepage, the web pages which cannot reach home page are regarded as errors. All dead end pages are also errors which violate reachability to homepage. Also, users can check the existence of a link between two different states in web application by link consistency properties, whose violations constitute the errors belonging reachability category.

4.4.2 Access Control Errors

Web applications can have services or contents for specific types of users. Users must be authorized for reaching such services and contents through an access control mechanism. The

access control mechanism can be employed on front end with web links or content or on back end with database configurations which restrict specific data fields to particular users. Providing a strong access control mechanism is a significant requirement in web security. In the OWASP 2013 top ten list of web application security [54], "A7-Missing Function Level Access Control" and in the CWE 2011 Top 25 most dangerous software errors list [55] "Missing Authorization" items address the access control errors and how significant they are in web security.

In this study, with the help of form filling enabled web crawler tool, Crawljax, we reveal states of web applications that is secured with the authorization mechanism, but also can be reached without authentication of users, without providing username and password information. For example, in a shopping site, only registered users who provide user id and password must reach the state of web application that indicates the payment for shopping is successfully completed. Otherwise, this indicates an access control problem of state displaying "payment is completed" message and in our analysis, we aim to determine such states of web applications. The analysis is performed on existing web applications and the reason behind the errors, design or implementation, is irrelevant from our analysis.

4.4.3 CTL Formulas

Web applications modelled as a NuSMV model presents all potential execution paths that may occur. The properties defined in CTL specify the desired paths and occurrences of the variables in the NuSMV model. Table 3 lists all the properties that we defined for verification of NuSMV model. CTL formula generation is performed by our tool, MCWebSoft, and users are not exposed to CTL syntax. The CTL formulas for specifying navigation properties in index reachability and absence of dead end categories are generated in fully automated manner. The generation of the link consistency and access control property formulas requires users only to choose the corresponding states.

Table 3: Navigation and access control properties in CTL

Category	**CTL Formula**	**Description**
Index Reachability (automated)	$AG(webstate = state_i \rightarrow EF\ (webstate = index))$ For each $state_i$ of the model	Index page is reachable from all other reachable states.
Absence of Dead End (automated)	$AG(webstate = state_i \rightarrow \neg EX(webstate = dead_end_webstate))$ $AG(webstate = state_i \rightarrow EX((webstate =$ index$) \vee .. \vee (webstate = state_{i-1}) \vee (webstate = state_{i+1}) \vee .. \vee (webstate = state_n)$ For each $state_i$ of the model, and for n is the number of webstates	Every reachable state has a next state in the transition.

Table 3 (continued): Navigation and access control properties in CTL

Link Consistency (semi-automated)	$AG(webstate = state_p \rightarrow EF(webstate = state_q))$	It is always possible to reach $state_q$ from $state_p$ (user chooses $state_p$ and $state_q$)
Access Control (semi-automated)	$\neg EF(webstate = state_p \wedge login = FALSE)$	$State_p$ is accessible only through login action (user chooses $state_p$)

The automated reachability properties state that index webstate is always reachable from all other webstates of web application and every reachable webstate always has a next page (no dead ends).

In this table, the index reachability could have been specified as $AG(EF(webstate = index))$. However, in that case NuSMV would generate only one counterexample even if there were more than one webstates which cannot reach the index. Hence we chose to write CTL formulas for each webstate in the set of reachable webstates. Also, absence of dead-end property is defined for all webstates for the same reason, to generate counterexamples giving all webstates in web application which violates the property.

Link consistency as another property specifying reachability, states that it is always possible to reach from a given state to a specific other state. For instance, it is always possible to reach login page from index page. It is enough for users to select the two webstates to check if they were connected by a clickable page element in web applications.

Access control checks if a state is reached after a login activity is performed. To define access control properties, the model should have been extracted by Crawljax and the user has to mark the webstates where the guard condition becomes true and false. In the special case where there is only one guard called login, user has to mark the webstates that login and logout activity take place. To mark these activities, firstly users specify authentication details as id and values for user name and password fields with the help of MCWebSoft interface. Then they select the XPath expression of page element where they fire click events for login and logout and also they select the webstates of web application where these events start and end. The selection is made from a list box where all webstates and elements of web applications are hold. After these actions, a user can ask MCWebSoft to check that a webstate is never accessed without proper authorization.

In Table 3, CTL properties are given according to syntax described in Section 2.1. The CTL formulas defined in NuSMV input language for checking index reachability, dead end absence and access control properties with the model of our running example are provided in Appendix B.

4.5 Converting Counterexamples to Executable Error Scenarios

Counterexamples obtained from the output of NuSMV execution consist of element variables with XPath expressions, and webstate variables with the name of the state of the web page. Also, if exists, value of the guard condition, that lead to violation of a property exists in counterexamples. Recall that guard conditions are added to NuSMV model, when users define conditions in order to add access control properties. While defining conditions, users must provide username and password values for login operation with the user interface of our tool. These values are also added to executable scripts to enable replay of error traces after login operation.

When it comes to our running example, NuSMV generates 7 counterexamples. The index reachability is violated in three webstates; ‘state9’, ‘state4’ and ‘state14’. The same webstates violate the absence of dead end properties as well. Since we check access control property, only with private pages, pages can be accessed after the login page, the webstate named ‘state4’ violated the access control property as expected. A counterexample is generated for this property which means that the webstate, ‘state4’ is reachable without authentication of user. The complete output of NuSMV for verification of running example is given in Appendix C.

Each violated property can be defined as an error on the system model of web application and counterexamples are error scenarios or traces. These error traces are enough for replaying errors on web browser. Therefore, the error traces are transformed into executable JUnit scripts to increase the user experience on understanding model checker output. The error scenarios are replayed on Mozilla Firefox where each step displays which portion of the web site is clicked. At the end of the scenario a dialog box displays the violation of which property has been replayed.

Our tool uses Selenium WebDriver libraries to automate the browser for replaying the error scenarios. We adopted the Java client driver libraries for binding the Selenium commands to JUnit code. The error scenarios replayed by Selenium are the JUnit tests that our tool generated using the counterexamples of NuSMV.

The conversion of NuSMV counterexamples to JUnit code is performed with a template JUnit file. The template file consists of imports of Selenium and JUnit packages, and fixture setUp and tearDown methods. SetUp initializes the browser and tearDown shows the dialog box. Each counterexample provides a sequence of states beginning from an initial state and ending at the state where the property violated. This sequence is used to fill in the test method of the template file. Here we exploit that the elements encode XPath expressions.

```
-- specification AG (webstate = state4 -> EF webstate = index)  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 2.1 <-
  login = FALSE
  webstate = index
  element = null_element
-> State: 2.2 <-
  webstate = state4
  element =
_HTML_1__BODY_1__TABLE_1__TBODY_1__TR_1__TD_1__TABLE_1__TBODY_1__TR_1__TD_1_
_A_1_
```

Figure 11: A counterexample generated by NuSMV for the running example

Figure 11 shows a counterexample generated by NuSMV. The first line in Figure 11 shows what property has been violated which is $AG\ (webstate\ =\ state4\ \rightarrow\ EF\ webstate\ =\ index)$. Next the sequence of states leading to error is given. The property is violated since the webstate, 'state4' cannot reach to the index page. Therefore the counterexample shows a trace reaching to the webstate 'state4'. The trace shows the violation is reached at the index page ($webstate = index$) when clicking on the element with the distorted XPath expression shown on the last line of the figure.

```
  …
@Before
  public void setUp() throws Exception {
    driver = new FirefoxDriver();
    driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);
    driver.manage().window().maximize();
    baseUrl = "http://localhost/RunningExample/index.php";
}
@Test
  public void test() throws Exception {
    driver.get(baseUrl);
    openDialog("clicking to
/HTML[1]/BODY[1]/TABLE[1]/TBODY[1]/TR[1]/TD[1]/TABLE[1]/TBODY[1]/TR[1]/TD
[1]/A[1]");
    driver.findElement(By.xpath("
/HTML[1]/BODY[1]/TABLE[1]/TBODY[1]/TR[1]/TD[1]/TABLE[1]/TBODY[1]/TR[1]/TD
[1]/A[1]")).click();
}
@After
  public void tearDown() throws Exception {
    Thread.sleep(10000);
    String note="violation of the AG (webstate = state4 -> EF webstate =
index))  property, Homepage is not reachable from this page";
    openDialog(note);
    driver.close();
```

Figure 12: Auto generated JUnit code from the counterexample in Figure 11 (Excerpt)

Figure 12 shows the excerpt of the corresponding JUnit code of counterexample given in Figure 11. (see Appendix D for the compete JUnit class) The first line of Figure 11 is transformed as a string constant assigned to note variable in the teardown method in Figure 12. The trace depicting the sequence of states and element leading to error in counterexample is implemented in the test method in Figure 12. The test method starts the execution by setting opening the base URL on the browser. Then it clicks on the element matching the XPath expression. Note that the use of distorted XPath expressions in the NuSMV model makes the body of test method filled easily with string replacement. The '_' characters need to be converted to brackets or '/'. This matching also enables traceability between models for expert user.

CHAPTER 5

MCWebSoft

In this chapter, MCWebSoft is explained with its system overview and implementation details, MCWebSoft was developed using Eclipse EE IDE for Web Developers with Java Standard Edition Development Kit 7.0. In order to apply model checking, we integrated NuSMV packages. To automate the web browser for replay counterexamples, we added Selenium and JUnit packages to our implementation. As for the user interface, Swing package of Java was adopted.

5.1 System Overview

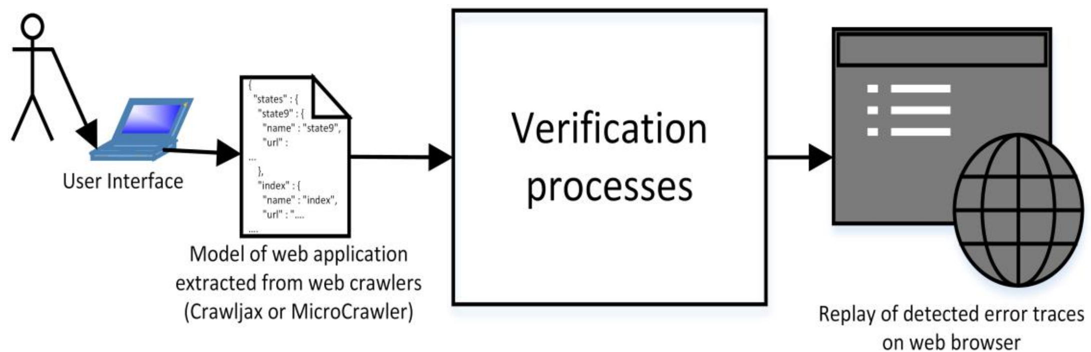

Figure 13: A high-level overview of the system

In the previous chapter, we explain the methodology of MCWebSoft. Before giving details about implementation, a high-level overview of the tool is given in Figure 13. In this figure, MCWebSoft takes one input which is the output of web crawlers, Micro-Crawler or Crawljax, associated with the web application to be verified. Then the tool outputs the verification result of web application as replay of detected errors on the web browser, MozillaFirefox.

As it is mentioned in Chapter 4, during the modelling and property specification phases, user can intervene the system by modifying the model of web application or generating new properties of access control or link consistency to be verified on NuSMV model of web application.

In Figure 14, a more detailed representation of the verification processes box, given in Figure 13, is presented. Conversion of web crawler output (SFG or WSM) to intermediate web model, conversion of intermediate web model to NuSMV model, model checking with NuSMV and converting counterexamples to JUnit classes are main processes of MCWebSoft represented in this figure.

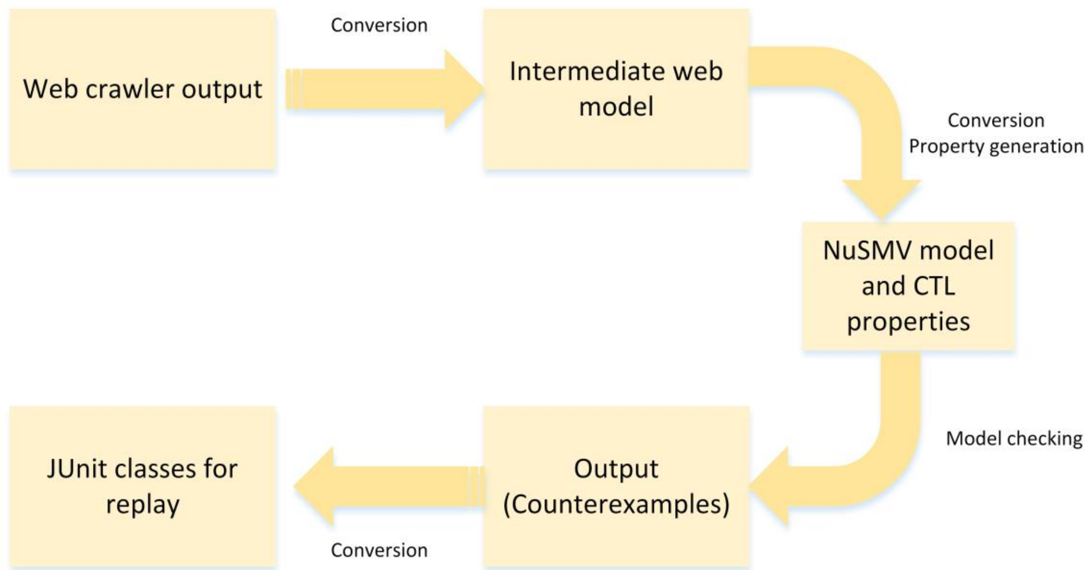

Figure 14: The detailed representation of the verification processes depicted in Figure 13

5.2 Implementation Details

Implementation details of MCWebSoft is depicted in three subsections which are decomposition description and description of classes, dependency description and the third party tools and packages used. In the first subsection, packages and classes constituting the system are explained. Second subsection introduces relations between classes with class diagrams. Last subsection involves the external tools and packages added to our implementation.

5.2.1 Decomposition Description and Description of Classes

In this section, explanation of the decomposition of MCWebSoft is given with the package structure. Moreover, classes of the packages are described by giving information about functions and purposes.

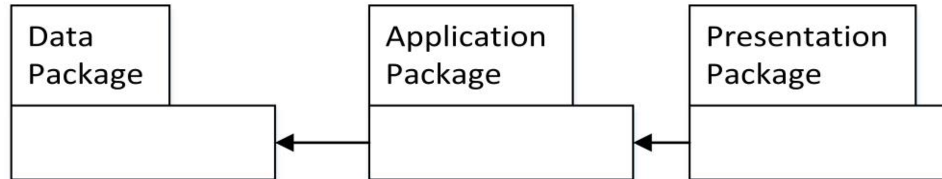

Figure 15: Package structure of MCWebSoft

Figure 15 depicts the package structure of MCWebSoft. The tool consists of three packages which are data package, application package and presentation package. Data package includes the classes holding the data variables and definitions. Application package contains classes which creates, changes the file types and executes the files. Presentation package includes the class created to form the user interface of the tool.

5.2.1.1 Data Package

Data package includes 6 classes which are WebState, Element, Edge, MicroCrawlerIntFormat, CrawljaxIntFormat and Trace and 1 interface which is IntFormat.

WebState Class

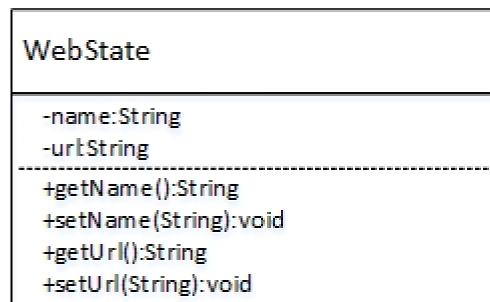

Figure 16: WebState Class

The purpose of Webstate class is to set and get the data of webstate variable. The webstate variable is comprised of two components which are name of webstate and URL address of webstate.

Element Class

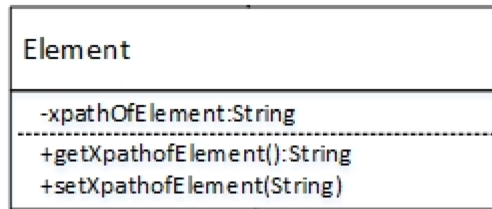

Figure 17: Element Class

The purpose of Element class is to set and get the data of page element. The XPath expression of element is given as attribute of Element class.

Edge Class

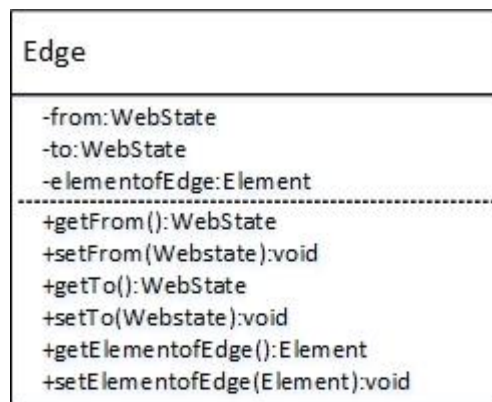

Figure 18*:* Edge Class

The purpose of Edge class is to set/get the data of edge variable, which provides the transition relation information. The edge variable consists of three components which are from, a webstate object describing from which webstate the edge begins, to, another webstate object referring by which webstate the edge ends and an element object, a page element connecting two webstates with the transition relation.

MicroCrawlerIntFormat Class

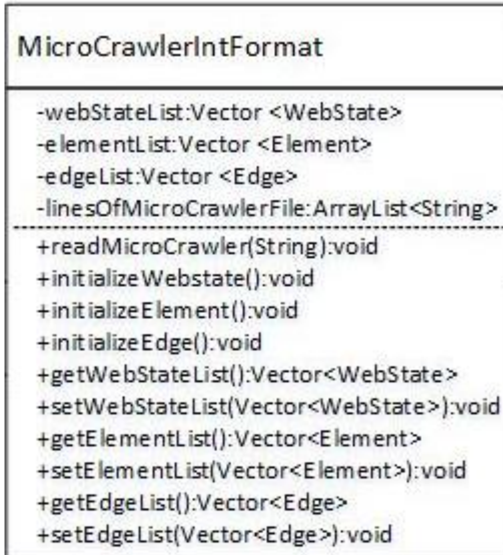

Figure 19: MicroCrawlerIntFormat Class

MicroCrawlerIntFormat class's purpose is to read MicroCrawler output file in order to render the components of intermediate web model. The function of the class is to read output of Micro-Crawler, WSM, and initialize the webstate, element and edge values in this state machine.

CrawljaxIntFormat Class

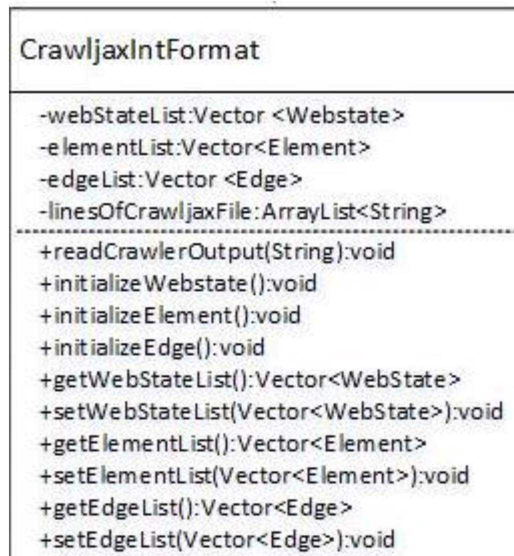

Figure 20: CrawljaxIntFormat Class

CrawljaxIntIntFormat class's purpose is to read Crawljax output file to capture the attributes of intermediate web model. The function of the class is to read output of Crawljax, SFG, and initialize the webstate, element and edge values in this model.

IntFormat Interface

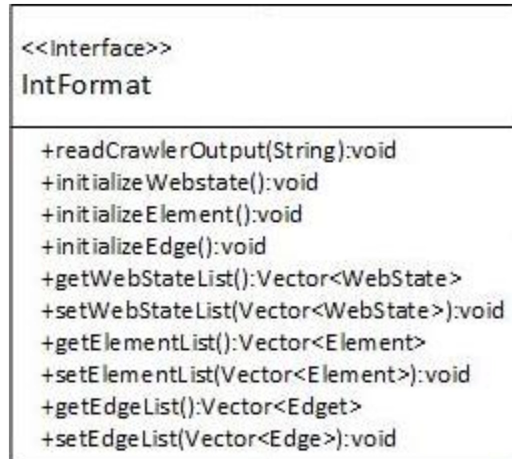

Figure 21: IntFormat Interface

The IntFormat interface is created to include the names of common methods of CrawljaxIntFormat and MicroCrawlerIntFormat in order to use polymorphism. By this way, it is possible to pass the arguments of MicroCrawlerIntFormat and CrawljaxIntFormat objects to the same methods. Moreover, this interface enables to be independent from web crawlers. Since common methods for reading and initializing the components of web crawlers are put in this interface, we can add new crawlers to the system without changing the rest of the code.

Trace Class

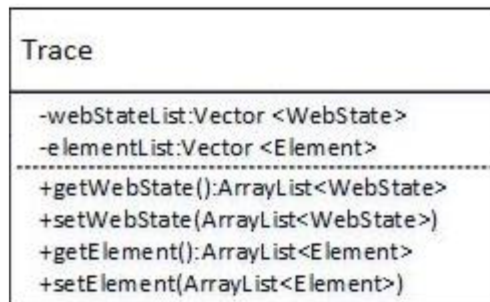

Figure 22: Trace Class

The purpose of Trace class is to get and set data values of counterexamples from the output of execution of NuSMV. Trace class is responsible for getting and setting the list of webstate and element values in error traces of counterexamples.

5.2.1.2 Application Package

Application package includes 4 classes which are CreateIntFormat, IntFormatToNuSMV, NuSMVExecuter and CounterexampleToScript.

CreateIntFormat Class

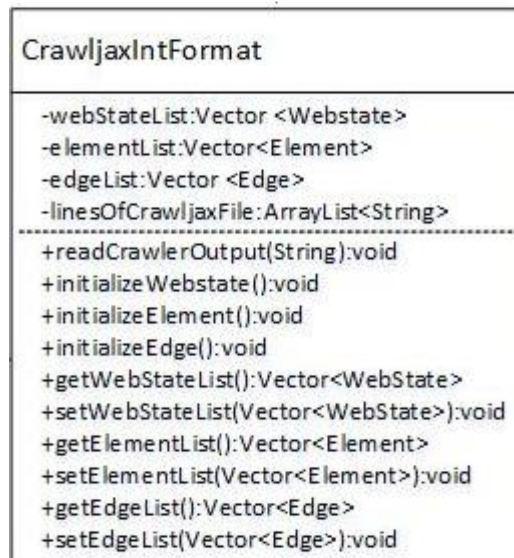

Figure 23: CreateIntFormat Class

The aim of creating CreateIntFormat class is to print the values of webstate, element and edge of crawler outputs to the intermediate web model. The function of the class is to get and set the path of the XML file including the intermediate web model and printing webstate, element and edge values as nodes of the XML-based model and creating the file with respect to this information.

IntFormatToNuSMV Class

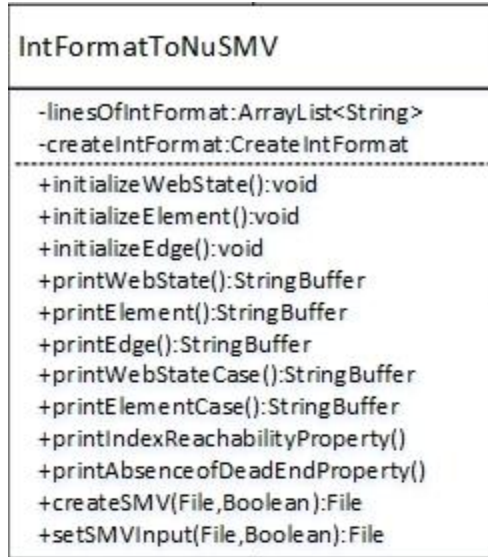

Figure 24: IntFormatToNuSMV Class

IntFormatToNuSMV class is created for converting the intermediate web model, an XML file, containing webstate and edge values, to NuSMV input language. The function of the class is to initialize the values of webstate, element and edge variables, printing the initialized values, initial values, transition relations and default reachability properties in NuSMV input language and finally creating and setting the NuSMV file.

NuSMVExecuter Class

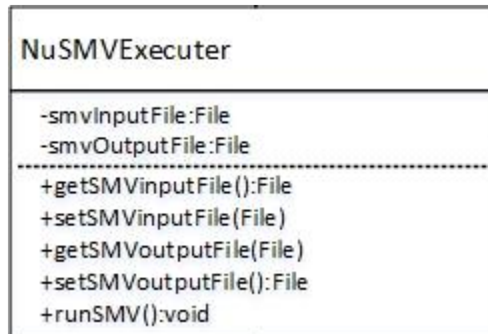

Figure 25: NuSMVExecuter Class

The aim at creating NuSMVExecuter class is to run the NuSMV input file with the model checker. The function of the class is to execute the NuSMV input file and get, set the NuSMV input and output files.

CounterexampleToScript Class

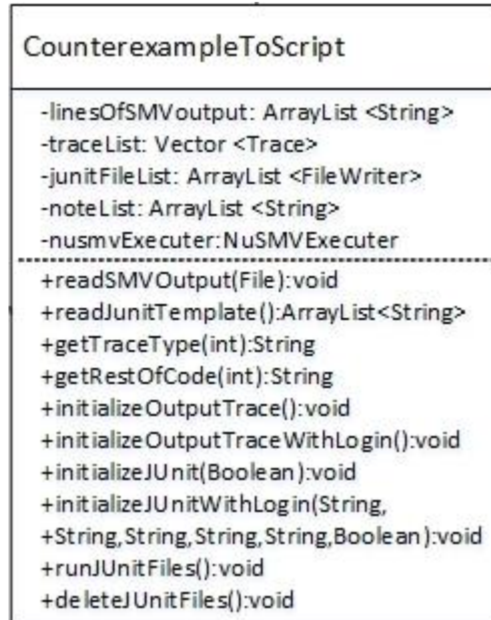

Figure 26: CounterExampleToScript Class

The purpose of CounterexampleToScript class is to create JUnit classes from counterexamples and running them. The class has the function of reading output file of NuSMV execution, template JUnit file, initializing the trace class with respect to output of NuSMV, filling the template JUnit file with traces of counterexamples according to NuSMV models with the login variable or without the login variable, executing the JUnit files and deleting the JUnit files.

5.2.1.3 Presentation Package

Presentation package includes 1 class named GUI and 9 interfaces constituting the listeners which are implemented as inner class of GUI class.

GUI Class

GUI

-mainFrame:JFrame
-generatedFileFrame:JFrame
-mainButtonPanel:JPanel
-mainConsoleTA:JTextArea
-mainPropertiesTA:JTextArea
-generatedFileTA:JTextArea
-isCrawljax: Boolean
-isMicroCrawler: Boolean
-chooseInputButton:JButton
-modifyWebModelButton: JButton
-runSMVButton: JButton
-createScriptButton:JButton
-addLinkConsButton:JButton
-defineLoginButton:JButton
-defineLogoutButton:JButton
-modifyWebModelButton: JButton
-createScriptButton:JButton
-replayButton: JButton
-listOfStartWebstate: JList
-listOfEndWebstate: JList
-listOfElements: JList
-usernameText: JTextField
-usernameIdText: JTextField
-passwordText: JTextField
-passwordIdText: JTextField
-createIntFormat: CreateIntFormat
-intformatToNuSMV:IntFormatToNuSMV
-nusmvExecuter: NuSMVExecuter
-counterexampleToScript:
CounterexampleToScript

+fillWebstate(String):ArrayList<String>
+fillElement(String): ArrayList<String>
+bringSMVModel():void
+readFile(File):StringBuffer
+getFilePath():String
+setFilePath(String):void
+main(String[]):void

Figure 27: GUI Class

The GUI class has 9 inner classes which are listeners for describing the events when a button is clicked. The names of these listeners are InputListener, ModifyWebModelListener, RunSMVListener, DefineLoginListener, DefineLogoutListener, AddAccessControlListener, AddLinkConsListener, CreateScriptListener and ReplayListener. GUI class is created for enabling

users to manage all the operations related to MCWebSoft's verification process. Functions of the GUI class, which are enabled by listeners, are to choose the input option, browse the input file, modify web model, create NuSMV model and generate default reachability properties, define login edge, define logout edge, generate access control property, generate link consistency property, run NuSMV model checker, create JUnit Files, replay the JUnit files. In addition to help users for understanding the flow of application GUI class has functions to show a console which explains the process details and all generated files, intermediate web model, NuSMV model, output of NuSMV, in separate windows.

5.2.2 Dependency Description

Relations between classes of the system are depicted in this section. The class diagram of MCWebSoft is explained with 5 figures. Figures 28, 29 and 30 show classes in Data, Application and Presentation package respectively. Figures 31 and 32 display the relations between classes of two different packages. Also the complete class diagram of the tool is provided in Appendix E.

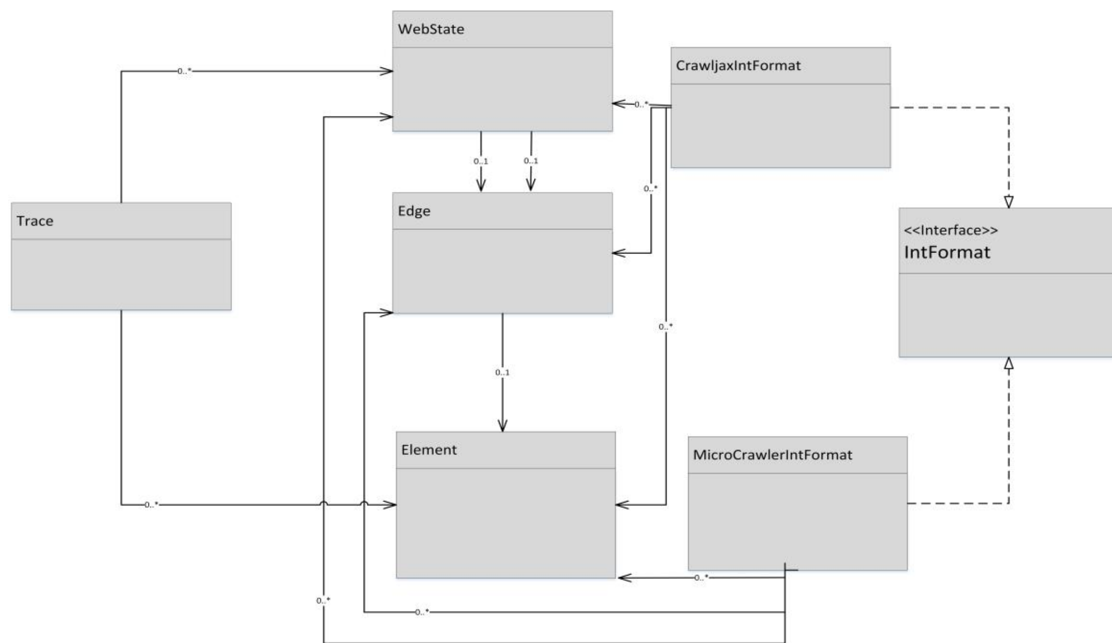

Figure 28: Class diagram of the Data Package

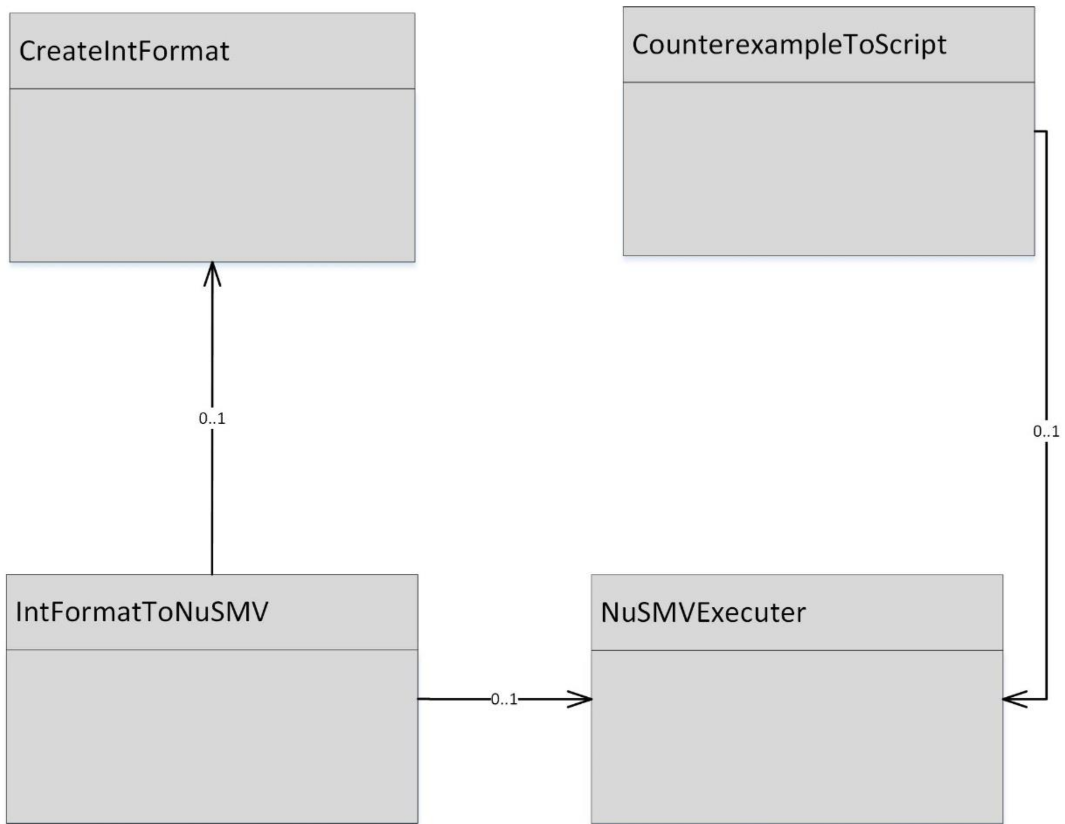

Figure 29: Class diagram of the Application Package

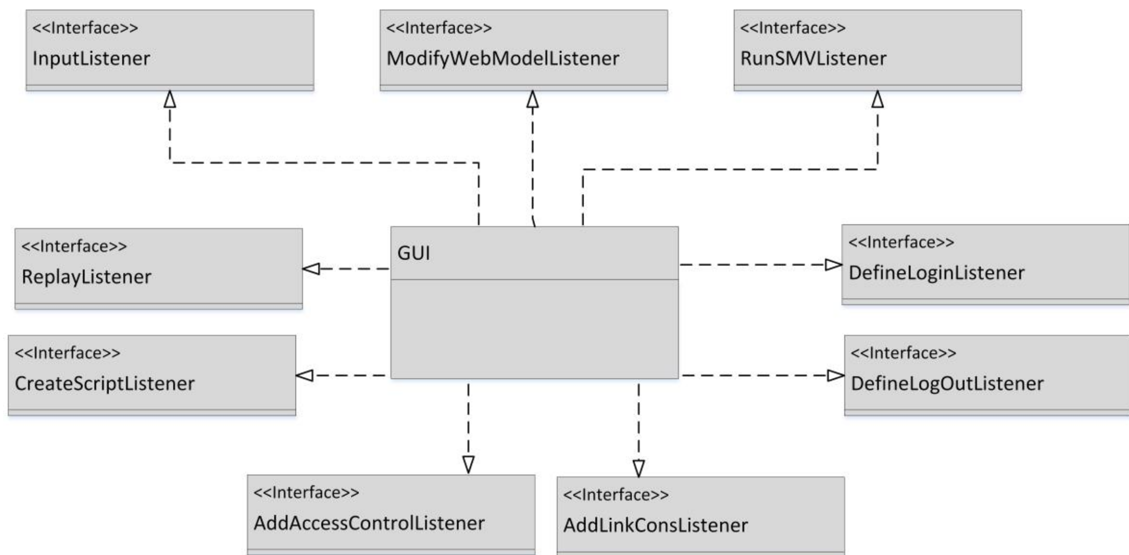

Figure 30: Class diagram of the Presentation Package

Relations between the classes which are located in different packages are showed in Figure 31 and Figure 32.

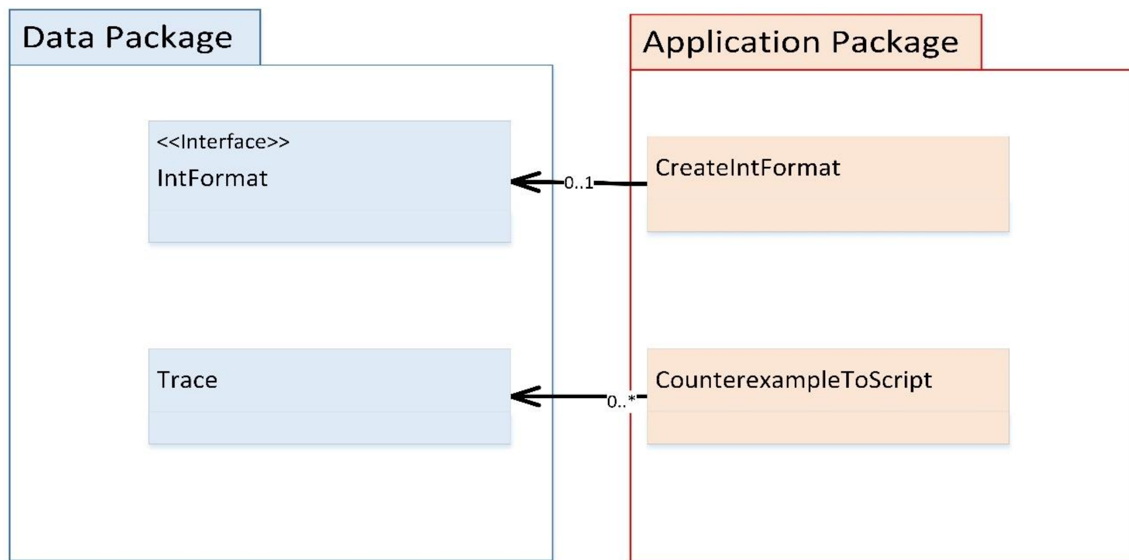

Figure 31: Relations between classes of data and application packages

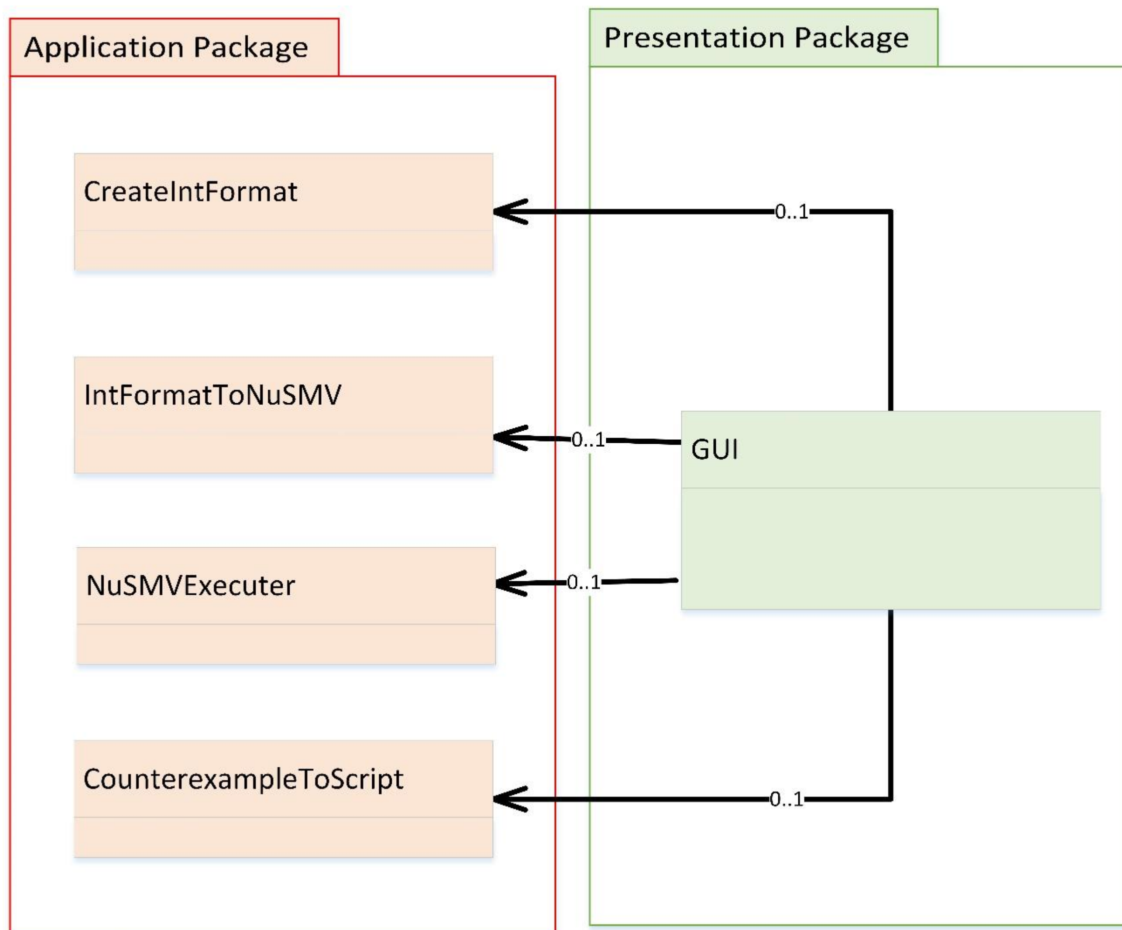

Figure 32: Relations between classes of application and presentation packages

5.2.3 Third Party Tools and Packages Used

5.2.3.1 NuSMV 2.5.4

NuSMV is an open source model checker. It is an extension of SMV model checker which is the first symbolic model checker tool. We used is NuSMV 2.5.4, which is the second version of NuSMV. Moreover, we employed the special version of the tool which is portable to the Microsoft Windows environment, which is available at the nusmv web site[1]. In the implementation, the execution file, named “nusmv.exe”, is run on console using Java.

5.2.3.2 Selenium

Selenium is an open-source tool for automating web browsers [56]. Selenium provides several usage options. One of them is Selenium IDE, which is an add-on for Mozilla Firefox that can

[1] http://nusmv.fbk.eu/NuSMV/download/getting-v2.html

record and replay the user interactions with the browser. This add-on also can convert commands for these interactions to an executable script in a specific language as Java and C#. Selenium Web Driver is another option which enables users to drive browser locally or on a remote machine using Selenium Server. In order to create and run Selenium test cases by a specific language, one should get a Web Driver language binding library for that language.

We have used Selenium WebDriver libraries to create Selenium scripts in Java in order to replay the scenarios of counterexamples. As a test engine, JUnit was employed to create scripts for replay. The set of commands we have used are standard ones as ‘open’, ‘click’ and ‘sendKeys’. In Selenium, there are also assertion functions which are used to verify test outputs, but our purpose for using Selenium is to replay the web browser, not testing. Therefore, we did not use assertion functions. In order to use the set of commands, page elements exposed to these commands should be identified. In Selenium, for identifying a page element, one of the HTML tag attributes (e.g., name, id or XPath expression) of the element should be provided. Since we have unique XPath expressions for each state of web application taken from web crawler outputs, XPath of elements were used as identifiers when parsing the commands of Selenium.

The Selenium libraries we imported for our implementation belong to the package, org.openqa.selenium.

org.openqa.selenium is a selenium package imported to the template file, template.java, which is used for creating JUnit classes for replaying the error scenarios. This package is required to realize the actions to open the web browser, Mozilla FireFox, identify a page element by its XPath expression and filling the forms or clicking it on subject web application automatically.

5.2.3.3 com.google.gson

The gson is an open source java library to convert JSON elements to java objects or vice versa. In our study this package is used for reading the file containing the crawler output of Crawljax web crawler which is a JSON file. With the help of gson, the output file parsed and separated to its components and transformed to the XML-based intermediate web model.

5.2.3.4 org.junit

The JUnit packages are imported to the template file called “template.java” from which JUnit classes to replay the error traces are produced. To automate the browser, a test generator must be used, we have chosen JUnit for this purpose. To add annotations (@Before, @Test and @After) that forms the actions of browser automations, JUnit packages are required.

CHAPTER 6

EVALUATION

In this chapter, the evaluation process of MCWebSoft is presented. The evaluation was performed in three steps. First a user study was conducted to measure the ease of use and the usefulness of the tool. Second we evaluated the error detection effectiveness. In order to evaluate the error detection performance with navigation properties, 30 real web applications were used. To measure the access control error detection performance, we used fault injection on a real application. The evaluations to assess error detection performance of the tool were carried on an Intel Core i5-3210 machine with 4 GB RAM running Windows 7. In the evaluations, web crawlers, Crawljax and Micro-Crawler, are downloaded from GitHub[1,2] and web application models are extracted using the latest versions of the web crawlers that are available.

[1] https://github.com/crawljax/crawljax/releases/tag/crawljax-3.6
[2] https://github.com/WebMole/Micro-Crawler

6.1 User Study

MCWebSoft helps users having no experience or knowledge about formal methods to apply model checking for verifying web applications. Therefore, one concern of this study is enabling inexperienced users to use model checking method. A user study was conducted to measure usefulness and ease of use of MCWebSoft in this respect.

The focus group consists of participants who are experienced about programming or web developing in different levels and inexperienced about model checking and formal methods. A questionnaire based on constructs of Technology Acceptance Model (TAM) were prepared to analyze user's responds to our tool.

TAM is an information system theory model designed to explore the technology acceptance and usage. In TAM, perceived ease of use and perceived usefulness factors predict the intention to use of a technology. And the intention to use factor effects the actual usage behavior [57]. Figure 33 depicts the constructs of TAM and their relations. Perceived ease of use is defined as "the degree to which a person believes that using a particular system would be free of physical and mental effort" and perceived usefulness is described as "the degree to which a person believes that using a particular system would enhance his or her job performance" [57]. TAM suggests a theoretically strong approach for the study of software acceptance that can be used in usability evaluations [58]. In the original version, there are 6 items measuring the perceived ease of use and 6 items measuring the perceived usefulness.

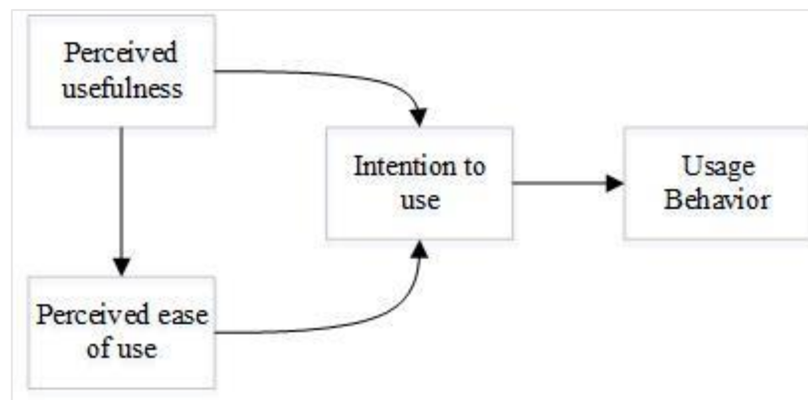

Figure 33: Technology Acceptance Model (TAM) (Adapted from [59])

The questionnaire of our user evaluation constitutes of 5 items with 5-point Likert scaled responses. 5 means strongly agree and 1 means strongly disagree. The questionnaire items are provided in Table 4. Besides these questionnaire items, we also provided a free text for the participants' comments at the end of the study. The questionnaire of our user study is given in Appendix F.

The user evaluation was conducted with 8 participants. Our focus group consists of the web admin of Informatics Institute, two experienced web developers worked with the web development frameworks Drupal and Dreamweaver, three graduate students of Informatics Institute two of

which are experienced Java developers, one undergraduate student with programming experience, and one graduate student with a web development experience with PHP and MySql. All of the participants had no experience in formal methods.

Within the user study, first the author of the tool, gave a ten minute presentation on how to use the tool and a five minute presentation of the subject web application. Then four verification tasks were given to the users. The tasks were loading an extracted model, starting the verification with auto generated properties, defining their own link consistency properties, and observing the reanimation of error scenarios.

During our evaluation process, users with no formal methods background succeeded to define their own properties and checked a web application with a model checker for the first time. After completing the tasks, they filled the questionnaire. In comment section of the user study, the participants reported tour tool to be useful for detecting and visualizing errors in their comments. Two participants suggested the tool to give a summary report for errors as improvement.

Table 4: Items of the user study evaluation and responses of all participants

Participants	**Questions**				
	I would find it useful	It would make easier to detect errors	Error scenario replay would improve my performance	I would find it easy to use	My interaction with the tool is clear and understandable
P1[1]	4	4	4	2	2
P2[1]	4	5	4	3	2
P3[1]	4	5	4	3	2
P4[2]	5	5	5	3	4
P5[3]	5	5	5	5	4
P6[3]	5	5	5	5	4
P7[4]	5	4	4	4	5
P8[4]	4	5	5	3	3

Table 4 presents Likert scaled responses of participants to each item of questionnaire. Superscripts of participants in the table are used to represent their experience in web development and programming. Participants showed with the 1 in superscript (e.g. $P1^1$) represent experienced web developers, participants indicated with the 2 in superscript (e.g. $P4^2$) represent little experienced web developers, participants indicated with the 3 in superscript (e.g. $P6^3$) represent experienced programmers and participants indicated with the 4 in superscript (e.g. $P8^4$) represent little experienced programmers.

Table 5 presents the analysis of these responses. These results confirm that the approach and the replay of error scenario are perceived as useful. The ease of use scores are lower than usefulness scores, which is acceptable since MCWebSoft is not developed as a commercial product.

Table 5: Perceived usefulness and perceived ease of use results

Factor	Questions	Median	Mean	Min	Max
Perceived usefulness	I would find it useful	4.5	4.5	4	5
	It would make easier to detect errors	5	4.75	4	5
	Error scenario replay would improve my performance	4.5	4.5	4	5
Perceived ease of use	I would find it easy to use	3	3.5	2	5
	My interaction with the tool is clear and understandable	3.5	3.25	2	5

6.2 Error Detection Evaluation

In this section MCWebSoft's error detection evaluations are given. The section is divided into two subsections as navigation errors and access control errors. In navigation errors section, the errors detected by reachability properties, which are automatically generated by MCWebSoft, are evaluated and discussed. In access control subsection, we evaluate and discuss the capability of MCWebSoft for finding errors in access control mechanism.

6.2.1 Navigation Errors

The reachability error detection effectiveness of MCWebSoft is evaluated by verifying navigational properties which check index reachability and absence of dead end. We used 30 real web applications for verification the properties. Table 6 gives the list of the subject web applications. The subject applications were crawled by Micro-Crawler and Crawljax for model extraction. Micro-Crawler requires all files to be in localhost. Therefore, we downloaded all the files in these sites with WinHTTrack Website Copier [60] and then crawl web applications with Micro-Crawler. With Crawljax, there was no need to copy web sites. Crawljax provides configuration options for crawling. By using these options, the maximum time for crawling is

determined as 120 minutes. To crawl all possible webstates in that time, the maximum level of depth is determined as 20. The web models are not manually modified, they are inputted and verified by our tool as it their original version yielded from crawler tools.

Table 6: Subject web applications used in verification of navigation properties

ID	**URL**
W1	http://www.seleniumhq.org/
W2	http://www.brain.ii.metu.edu.tr/eng_dworld.html
W3	http://www.acarturk.net/
W4	http://www.metu.edu.tr/~merkan/
W5	http://www.uyms2015.yasar.edu.tr/2015/index.html
W6	http://www.puppeteershadowplay.com/index.html
W7	http://www.tr.china-embassy.org/tur/
W8	http://www.kktcbe.org.tr/www/tr/anasayfa.asp
W9	http://www.mfa.gov.hu/kulkepviselet/TR/tr/
W10	http://www.ceng.metu.edu.tr/~altingovde/
W11	http://www.kovan.ceng.metu.edu.tr/~sinan/
W12	http://www.eonur.eu/
W13	http://www.ceng.metu.edu.tr/~manguoglu/
W14	http://www.ceng.metu.edu.tr/~polat/
W15	http://www.metu.edu.tr/~koksalan/
W16	http://www.metu.edu.tr/~stugba/index.html
W17	http://www.metu.edu.tr/~aliy/
W18	http://www.metu.edu.tr/~sdag/index.html#
W19	http://www.metu.edu.tr/~ymetin/
W20	http://www.metu.edu.tr/~sbilmis/
W21	http://www.retina.cs.bilkent.edu.tr/
W22	http://www.cs.bilkent.edu.tr/~erman/
W23	http://www.art.bilkent.edu.tr/ozguc/index.html
W24	http://www.cs.bilkent.edu.tr/~ozturk/index.html
W25	http://www.umram.bilkent.edu.tr/~ergin/
W26	http://www.volkan.bilkent.edu.tr/

Table 6 (continued): Subject web applications of navigation errors evaluation

W27	http://www.isbcs2015.ii.metu.edu.tr/index.html
W28	http://www.atilkurttekin.com/
W29	http://www.fince.bilkent.edu.tr/
W30	http://www.infobil.bilkent.edu.tr

We consider two main criteria while we select the web applications for our evaluation process. The small sized and dynamic web applications are put into subject list. We limit our evaluation to web applications which are small sized, since crawling time that we determined may not be enough for completing crawl the entire web application for large-sized web applications. The resulted incomplete crawler outputs may cause incorrectly reported errors by our tool. The details about size and crawl outputs (average number of webstates and edges) are provided in Section 6.3. The reason why we choose dynamic web applications is that the web crawlers we use are developed for AJAX-based web applications.

Since, the subject web applications are public domain applications, we do not have the source code or information about the development details of them. Therefore, the web technologies used in the development of the subject web applications are determined by the Web Developer browser extension [61], which provides the information about web applications. By means of the Web Developer, the technical details of subject web applications in Table 6, except W9, can be determined. According to the data provided with Web Developer extension, the subject web applications are developed with different back-end languages such as PHP, ASP.NET or Java and web server software such as Apache or IIS (Internet Information Server). Some of the subject web applications employ JavaScript programming language, whose influence on DOM trees can be detected by web crawlers we used, Micro-Crawler and Crawljax. (see Table 16 in Appendix G for more detail).

The properties checking navigation are divided into two categories which are labeled as index reachability and absence of dead end. These properties are automatically generated with the NuSMV model.

At the end of the verification process, which is conducted with the navigation properties, 72 counterexamples with Micro-Crawler and 847 counterexamples with Crawljax output models are generated. However, when we investigate the error scenarios in the counterexamples, we observed that some counterexamples were spurious; they do not represent real faults on web applications. We named these type of counterexamples as false alarms. In Table 7, the numbers of false alarms and real faults are given with respect to the verified property types and the web crawler that extracted the model of web application (see Table 17 and 18 in Appendix G for more detail).

Table 7: Results of verification of navigation properties

Web Crawler	Micro-Crawler				Crawljax			
Property Type	Index Reachability		Absence of Dead End		Index Reachability		Absence of Dead End	
Counter example Type	Real faults	False Alarms	Real faults	False Alarms	Real faults	False Alarms	Real faults	False Alarms
Results	25	17	20	10	54	416	57	320

In Figures 34 and 35 the pie charts corresponding to the results are shown. The percentage of real and false alarms with the numbers which are also given in Table 7 are depicted in the pie charts.

Results of the Verification Process with Micro-Crawler

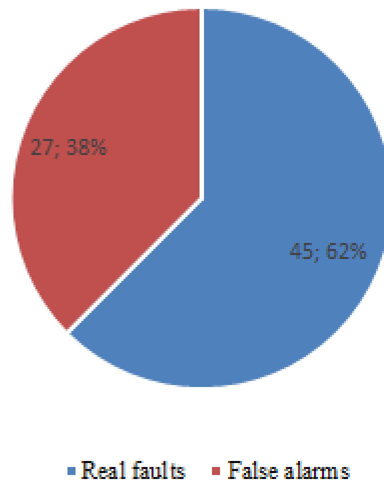

Figure 34: Verification results with respect to Micro-Crawler's output

As it is seen in Figure 34, at the end of the verification procedure with Micro-Crawler, the number of real faults were found as 45 and this corresponds to 62% of generated counterexamples. The number of false alarms are 27 which corresponds to 38% of counterexamples.

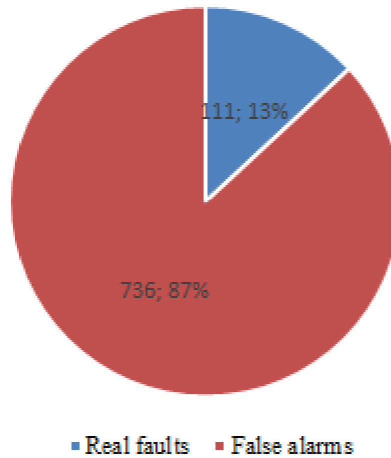

Figure 35: Verification results with respect to Crawljax's output

In Figure 35, the results of the verification process with Crawljax, show that 13% of counterexamples are real faults and 87% of them are false alarms. Among all of the counterexamples generated with Crawljax model, the number of false alarms are 736 and the number of real faults are 111.

The false alarms are due to the incorrectness of the model extracted. For example, in W14 the crawler missed an edge between a webstate and index page, which caused two false alarms. To investigate this situation, we selected three subject applications (W14, W24 and W28) and refined the web model obtained with Micro-Crawler using *only* the animated error scenarios. We fixed the model with the option of modifying the FSMs of our tool. Table 8 shows the number of error scenarios, number of false alarms, and the model size in terms of number of webstates and edges. The top part shows these values for the extracted model and the bottom part shows them after the models are refined. After the model refinement, where we added the edges missed by the crawler, all of the false alarms were eliminated. The time spent in analyzing the scenarios and refining the intermediate web models was about 15-20 minutes for each of web applications.

Table 8: Number of real faults and false alarms generated before and after model refinement

	ID	Number of error scenarios	Number of false alarms	Number of webstates	Number of edges
Before modification	W14	6	2	13	74
	W24	12	4	16	78
	W28	8	6	11	29
After refinement	W14	4	0	13	75
	W24	8	0	16	80
	W28	2	0	11	32

6.2.2 Access Control Errors

Evaluation of MCWebSoft for detecting access control errors is performed with the fault injection technique. In order to eliminate the possibility that the randomly chosen web applications had no access control errors, the errors were injected to a real web application. Then the faulty versions are verified against the access control properties. We injected faults into an open source web application developed with PHP, called Restaurant Script[1]. It is a pizza restaurant system enabling registered users to manage food orderings. Figure 36 shows the model of the Restaurant Script web site. In this figure, each rectangular box represents a webstate, name of the webstate and the name of the web page corresponding to this webstate is given for each one. Arrows represents the edges between the webstates.

[1]The source code of the application is downloaded from http://sourceforge.net/projects/restaurantmis/?source=directory

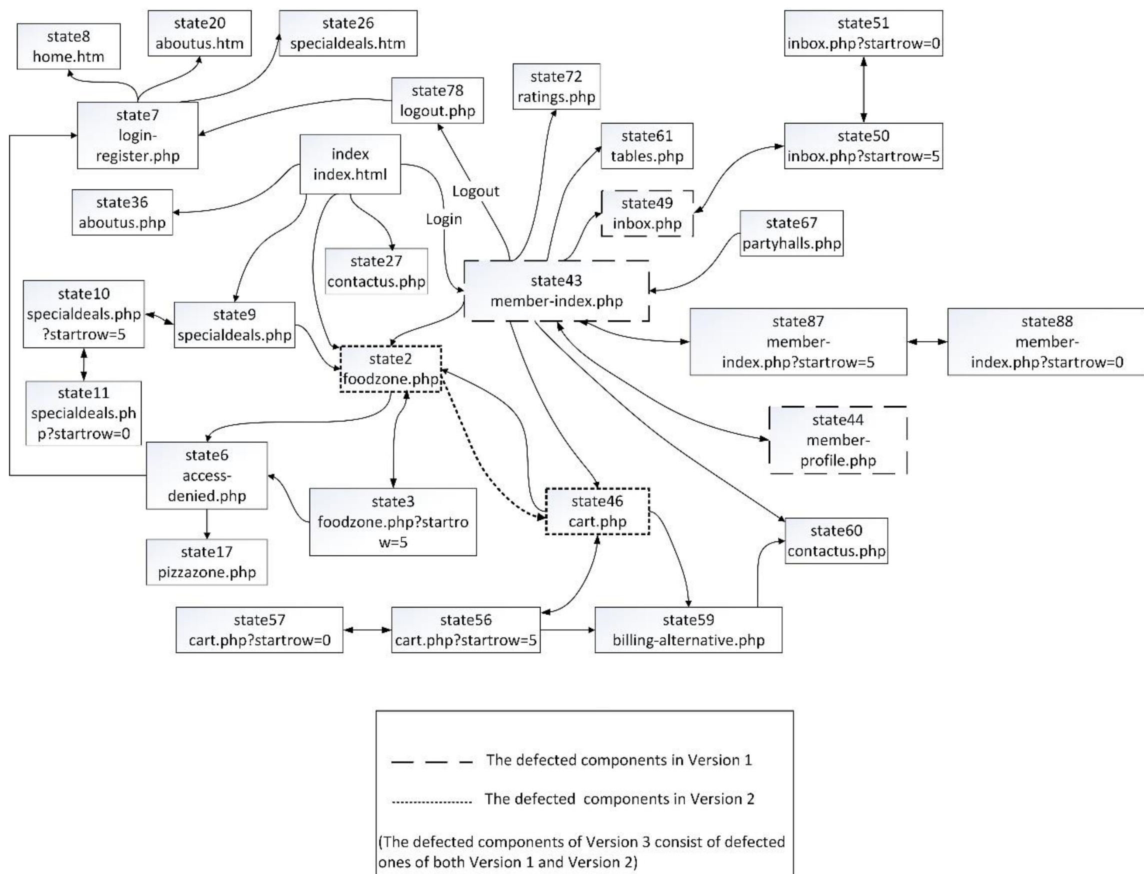

Figure 36: The model of the Restaurant Script web site

We check the Restaurant Script website with MCWebSoft by access control properties verifying that the private webstates are only accessible with login action. In Figure 36, private webstates are represented as rectangular boxes which can be accessed only after the webstates named 'state 43'. According to the results, there is no counterexamples and so no replaying error scenarios. With this result, we verify that the subject website has no access control error.

In this application, the PHP file "auth.php" checks the authenticate values for login operation. In this experiment, a fault injection into a PHP file was commenting out the lines that refer to auth.php (see Figure 37). We created three faulty versions of this application called Version 1, Version 2 and Version 3. In Version 1, only membership files were changed. In Version 2 only the cart operations files were changed, and in Version 3 both cart operations and membership files were defected. In Figure 36, defected webstates and edges are represented as dashed lines.

```
<?php
    //require_once('auth.php');
    require_once('admin/locale.php');
?>
```

Figure 37: Code fragment causing access control errors

For this experiment Crawljax 3.6 was used for model extraction since it enables defining values for form filling. Then, for each version, we defined the edges where login and logout activities occur on the application. During the property specification, we selected the webstates requiring authorization (i.e., the nonpublic webstates) for access control property generation. The same set of nonpublic webstates were declared for each version. Finally, the NuSMV model and the properties were sent to model checker. Table 9 shows the number of counterexamples generated for each fault-seeded version of the Restaurant Script application.

Table 9: Results of verification of access control properties

Versions	Number of Counterexamples
Version 1	14
Version 2	3
Version 3	22

The verification results show that in Version 1, in total 14 webstates violated access control properties. In Version 2, a total of 3 webstates and in Version 3 a total of 22 private webstates, which require authorization of the user, can be accessed without login. Details about these counterexamples are given in Table 10, 11 and 12. The names of webstates and page names which correspond to these webstates leading to counterexamples are introduced in these tables. These webstates can be seen in Figure 36. However, the webstates of the model in Figure 36 correspond to the original version of the subject web site, which is not fault-seeded. Although, we have showed the defected components in this figure, there can be webstates which are found with faulty versions and not in the model displayed in Figure 36 or vice versa.

Table 10: The webstates detected in Version 1

Name of Webstate	Page Name
state43 state87, state63, state65 state88, state66, state21	member-index.php member-index.php?startrow=5, member-index.php?startrow=0,
state49 state51 state50	inbox.php, inbox.php?startrow=0, inbox.php?startrow=5
state44, state22	member-profile.php
state72	ratings.php
state78	logout.php

Table 10 gives webstates which are revealed by MCWebSoft with Version 1 of the Restaurant Script web site. In Version 1, membership pages were accessed without login due to fault injection and the webstates detected correspond to membership files.

Table 11: The webstates detected in Version 2

Name of Webstate	Page Name
state46, state62	cart.php
state78	logout.php

Table 11 gives webstates which are revealed by MCWebSoft with Version 2 of the Restaurant Script web site. In Version 2, cart operation pages were accessed without login due to fault injection. According to the results, two webstates that belong to cart operations can be accessed without login operation since they belong to the "cart.php" web page. In addition, the webstate corresponding to logout page is accessed due to defecting cart operations pages.

Table 12: The webstates detected in Version 3

Name of Webstate	Page Name
state43, state15 state87, state66 state88, state68	member-index.php member-index.php?startrow=5, member-index.php?startrow=0,
state49 state51, state34 state50, state33	inbox.php, inbox.php?startrow=0, inbox.php?startrow=5
state44, state37, state25	member-profile.php
state72	ratings.php
state78	logout.php
state46, state23 state57, state34 state56, state35	cart.php cart.php?startrow=0 cart.php?startrow=5

Table 12 gives webstates which are revealed by MCWebSoft with Version 3 of the Restaurant Script web site. In Version 3, both membership pages and cart operation pages were accessed without login due to fault injection. As it can be seen, there are more webstates found with Version 3 than Version 2 and Version 1, which is reasonable since Version 3 is created by defecting both

membership and cart operation files. With these evaluations, it was observed that all authorization problems were identified and there were no false alarms.

6.3 Performance Evaluation

In order to evaluate the performance of MCWebSoft, we give the size of subject web applications and total time of verification of each subject web site by MCWebSoft. The time is obtained by adding the time spend by MCWebSoft for converting web crawler models to intermediate web model, generating NuSMV model and properties, verifying the properties and creating executable scripts with respect to counterexamples. In order to present the size of evaluated web sites, the number of webstates, the number of edges in the web crawler output models which are formed as intermediate web model, the average DOM size and the file size of subject web applications are given.

6.3.1 Performance of navigation verification

In order to evaluate the performance of MCWebSoft in terms of verification of navigational properties, 30 case studies, which are given in Table 6, were examined. The details of the verification processes are explained in Section 6.2.1. As well as giving time and space used by MCWebSoft, the factors which have significant effect on the verification time is detected by correlation analysis.

6.3.1.1 Results of the Size Measurement

In this section the distribution of subject web applications is presented in terms of their size values. Numbers of webstates and edges, DOM size and file size values are given with this purpose. In Figure 38, 39, 40 and 41, the distribution of the size measurements of the subject web applications, which are given in Table 6, is depicted with boxplot figures. The file size information is obtained by measuring the size of copied version of subject web applications which is used to verify Micro-Crawler output. DOM size value is taken from output of Crawljax for each subject web application. However, we have two values for webstate, edge and time values for each web crawler output. Therefore the averages of the numbers of webstate, edge, attained with Crawljax and Micro-Crawler outputs are given for performance analysis. Moreover, the separate values of webstate and edge numbers for each of two crawlers are provided in Table 20 in Appendix G.

Boxplot is a graph to display the distribution of quantitative data. They represent the values as separated by their quartiles. The vertical line in box shape denotes the median value, the data value within the range of box shape is between first and third quartiles. The lines extending from box is called whiskers and show the smallest and largest value of the data set. The points on the figure symbolize the outliers. The numbers on the outliers refer the subject web application with the order number in Table 6.

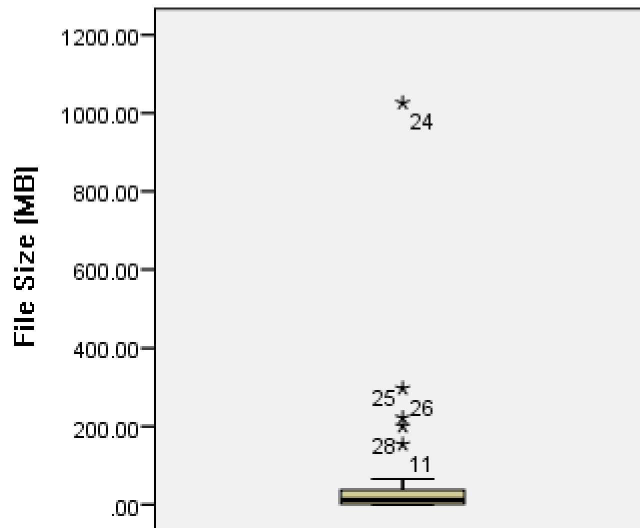

Figure 38: Boxplot of file size of subject web applications

In Figure 38, file size analysis of the subject applications is presented. According to the results, file sizes can change between 0.1 MB and 1026 MB, mean of them is 74.2 MB, median is 12.23 MB and standard deviation is 194.7 MB. As it is seen from the figure, there are 5 outliers. In the boxplot figure, it is also observed that the vertical line representing median and the box is skewed to the low part of the range line which means that size of the subject web applications is general have lower values.

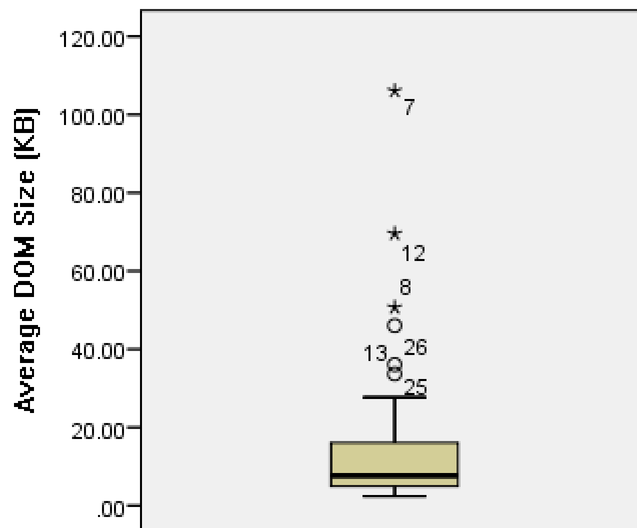

Figure 39: Boxplot of average DOM size of subject web applications

Figure 39 gives the boxplot of average DOM size values of the subject web applications. The average DOM size value is taken from the output of the crawl process done by Crawljax tool. The

results show that DOM size is between 2.4 KB and 106.1 KB, and have a mean of 18.3 KB with standard deviation 23.3 KB. Median of DOM size values is 7.65 KB. As in Figure 38, the outliers also exist in Figure 39. Also, as in the case of Figure 39 most of the data is gathered in low values, so the average DOM size of all of the subject web applications are skewed to small values.

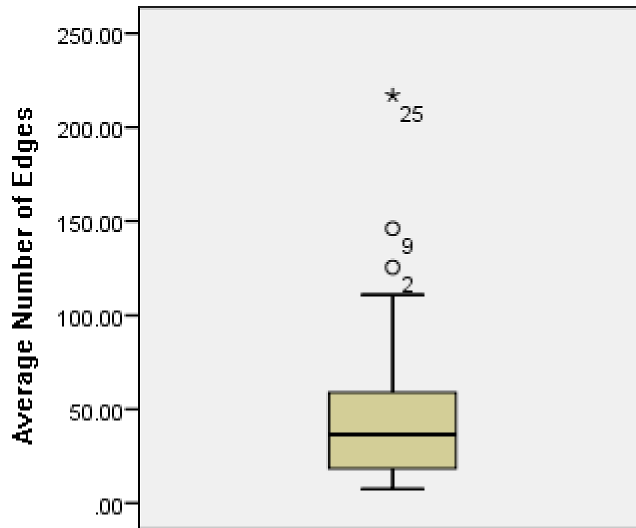

Figure 40: Boxplot of average number of edges in intermediate web models subject web applications

In Figure 40, the values of edges numbers as average of output of two crawler outputs are depicted in boxplot. According to the results, average number of edges can change between 7.5 and 217, mean of the edge values is 50.7, median is 36.5 and standard deviation is 47.7. There are three outliers in the boxplot. Also, as other variables, most of the data of the average number of edges set is skewed to low end of the range.

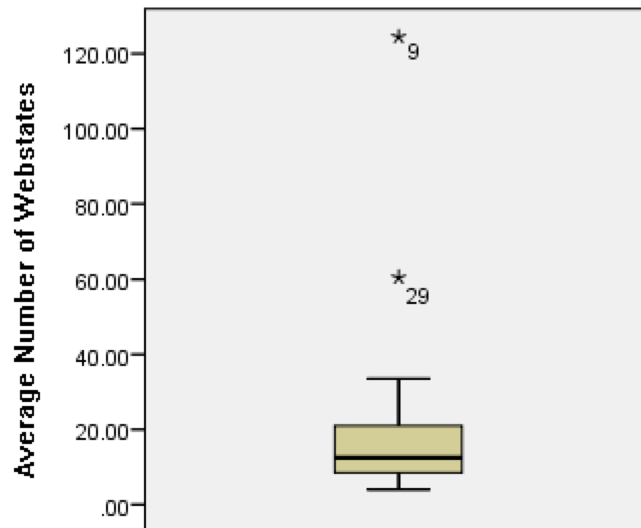

Figure 41: Boxplot of average number of webstates in intermediate web models subject web applications

Figure 41 gives the boxplot of the average number of webstates obtained from the output models of the Crawljax and Micro-Crawler tools. The results of descriptive analysis show the average number of webstates is between 4 and 124.5, the mean is 19.6 with standard deviation 23.1 and the median is 12.5. As it was shown in boxplot, there are two outliers and average number of webstates are skewed to low ranges.

Table 13 shows the descriptive statistics of size measurements which is illustrated in boxplots in Figures 38-41.

Table 13: Results of size measurements of subject web applications in Table 6

	Minimum	**Maximum**	**Mean**	**Median**	**Std. Deviation**
File Size(MB)	0.1	1026	74.2	12.23	194.7
Average DOM Size(KB)	2.4	106.1	18.3	7.65	23.3
Average Number of Webstates	4.0	124.5	19.6	12.5	23.1
Average Number of Edges	7.5	217	50.7	36.5	47.7

According to the results, subject web applications have low size values in both number of webstate and edge values, DOM sizes and file sizes.

6.3.1.2 Results of Performance Evaluation

In this section, we examine the time and space used by our tool in verifying the subject web applications. The time for verification is measured by adding the time spend for converting web crawler outputs to intermediate web model, generating NuSMV model and properties, verifying the properties and creating executable scripts with respect to counterexamples by MCWebSoft. We perform two verification analysis with MCWebSoft for each subject web applications, since we have two web crawler outputs for each web site. Therefore, average time for each verification process is given. The time measurements for each version of web applications are provided in Table 20 in Appendix G.

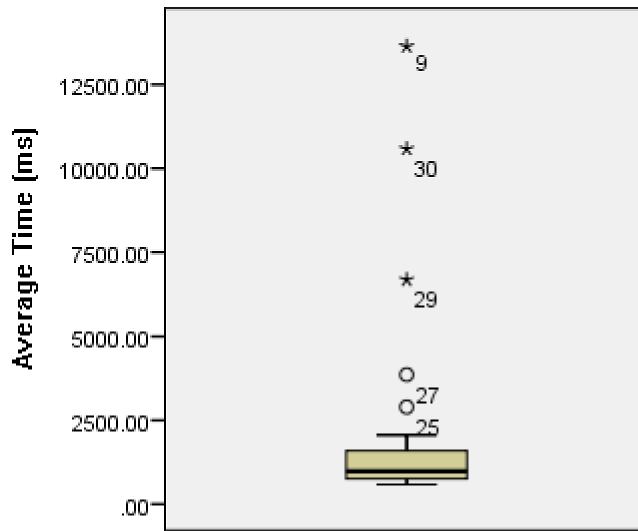

Figure 42: Boxplot of average time spend in verification of subject web applications

Figure 42 displays the boxplot of time spend by MCWebSoft on verification of each web applications given in Table 6. The whole process for each subject took 586.5 ms at minimum and 13.645 ms at maximum. The median of time measurements is 977 ms and the mean is 2076.9 ms with standard variation of 3018.1 ms.

Table 14 shows the descriptive statistics of time measurements which is illustrated in boxplots in Figure 42.

Table 14: Time measurement results of subject web applications in Table 6

	Minimum	**Maximum**	**Mean**	**Median**	**Std. Deviation**
Average Time(ms)	586.5	13645	2076.9	977	3018.1

Time and memory measurements are two of the important concerns of the model checking domain. According to these results, average time values for evaluating 30 web applications is approximately 2 seconds with approximately 3 seconds standard deviation.

In order to find which factors affect the time of evaluation, we performed correlation analysis, with size, average DOM size, average number of webstate and average number of edges. It was found that only average number of edges and average number of webstates have an important effect on time. In Figures 43 and 44, these relations are given with scatterplots.

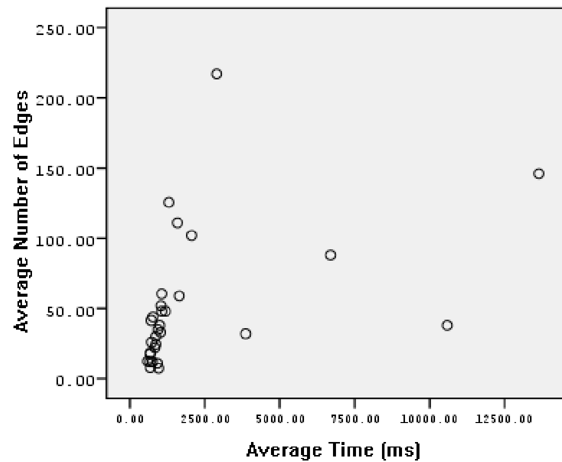

Figure 43: Scatterplot of average number of edges and average time

In Figure 43, the relation between the average number of edges and average time spend on verification is displayed. The correlation between average number of edges and time was found significant at the 0.01 level (p=0.019). According to this result, web applications having more edges, namely the web applications which have more links and buttons that can change the DOM structures, can take more time for our tool's verification process.

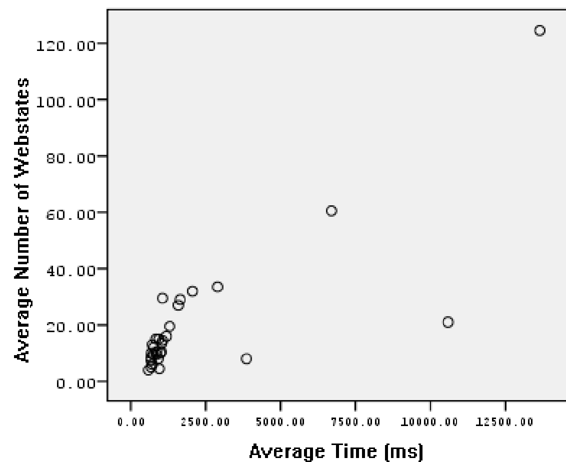

Figure 44: Scatterplot of average number of webstates and average time

In Figure 44, the relation between the average number of webstates and average time are displayed. The correlation analysis shows that the significance value of the correlation of the average number of webstates and time is smaller than 0,005. By this result, we can say that the web applications which have more webstates take more time for our tool's verification process.

The results of correlation analysis saying that webstate and edge numbers have a positive effect on evaluation time, is not surprising, since we verified the web sites based on their models as intermediate web model which includes webstate and edge components. Moreover, we have detected that file size and average DOM size of a web application have an insignificant effect on the time spent by MCWebSoft. The size of web application files includes all web objects in the web application including jpeg and gif files. Therefore this insignificant effect is reasonable, since these image objects have no effect on the verified model of web application, i.e., NuSMV model. Moreover, average DOM size has insignificant effect on time of verification, since the DOM size measurement is taken from Crawljax output, but the time values are the average of the measurements of the verification with both Micro-Crawler and Crawljax output models.

According to the results, only components of the intermediate model, webstate and edge, can affect the time of verification. With this conclusion, we affirmed that the time for model checking is dependent on the model verified by model checking [62].

6.3.2 Performance of Access Control Verification

The results of the space and time measurements of access control verification with the purpose of evaluating the performance of MCWebSoft are given in this section. One web application is used for our access control evaluation (see section 6.2.2 for details). Table 15 gives the results of performance evaluations of each fault-seeded version of the Restaurant Script web site.

Table 15: Results of performance evaluation of the Restaurant Script web site

Versions	Size(MB)	Average DOM Size(KB)	Number of Webstates	Number of Edges	Time (ms)
Version 1	1.14	4.257	28	49	963
Version 2		5.945	50	63	1240
Version 3		5.662	49	63	1293

According to the results in Table 15, size of web application is 1.14 MB, since the webstate and edge numbers change for each one for three fault-seeded versions of web applications, the time for the tool to complete the verification were different; 963, 1240 and 1293 ms respectively.

As in navigation verification, in access control verification the time values are reasonable for model checking. Time for executing these three versions is approximately 1 seconds (963 ms, 1240 ms and 1293 ms for three versions of the subject web application respectively). The numbers of webstates obtained from Crawljax web crawler are between 28 and 50, the numbers of edges are between 49 and 63 respectively.

CHAPTER 7

DISCUSSION AND CONCLUSION

7.1 Summary

This thesis introduces an approach for verification of web applications with model checking and presents an implementation of this approach as a tool-chain, called MCWebSoft. In this context, we aim to help inexperienced users in formal methods with the challenging tasks as modelling, property specification, and locating the errors from counterexamples. Our tool benefits from two Ajax-enabled web crawlers for extracting models of web applications. Also, an intermediate web model between crawler output and model checker input is developed to enable modifications on the web model extracted from web crawler. MCWebSoft also assists in defining CTL properties checking access control and link consistency and automatically generates CTL properties for assessing reachability of web applications. After the verification process with model checking, generated counterexamples are automatically converted to executable scripts to be replayed on web browser for allowing of visualizing the detected error traces on runtime.

The developed tool was evaluated in three parts. In the first part, with a user study we evaluated the usefulness and ease of use of our tool. The focus group had no previous knowledge on model checking techniques, but familiar with web development and programming in various levels..

They found MCWebSoft useful for detecting and visualizing errors. However, according to the results, the ease of use could be improved

In the second part, we ran experiments on 30 real web applications to investigate the effectiveness in detecting navigational errors. Lastly, we evaluated the effectiveness of access control error detection by fault injection on an open source web application. It is observed that MCWebSoft is successful in finding errors in real web sites. Also, false alarms can be detected by the tool with the aid of error scenario execution. To eliminate false alarms, the refinement on the intermediate web models was performed and it was found sufficient in evaluations.

7.2 Contributions

As web applications become increasingly common and have critical functionalities, verification of these applications gains importance. Verification allows designers to correct inconsistencies in desired features of web applications. Navigation and access control of authorized services are two important features which deserve to be confirmed and corrected before leading bigger problems. Model checking is a rigid and reliable technique for verification, but using this techniques requires knowledge of mathematical concepts. In this thesis an implementation method for verification of web applications with model checkers for inexperienced users is presented.

Our approach combines different methods as using web crawlers for model extraction, converting navigational models to NuSMV, checking navigation properties with a model checker and converting NuSMV counterexamples to executable scripts. In previous related studies, these methods were used separately (see Chapter 3), the main contribution that we present lies in the combination of the many different techniques in one approach and implementation of these approaches as a tool-chain. Also, usage of some methods differ from other studies; converting counterexamples to executable scripts as enabling automatic replay of counterexamples was not implemented, but generating counterexamples which can be replayed by a capture/replay tool was developed before [29, 49].

Moreover, we introduce the first study that uses Crawljax and Micro-Crawler for model checking. Although, previously the adaptability of output model of web crawler tools to model checking is questioned and further analysis are suggested on web crawlers, the results of error detection performance gave us valuable results by revealing the deficiencies of these web crawler tools' performance. We have detected that web crawler tools cause incorrect results when checking navigation properties. According to our evaluation results conducted with 30 web applications, Crawljax-generated models led more false alarms, false positive results, than Micro-Crawler. (Micro-Crawler: 38% false alarm, Crawljax: 87% false alarm). We have detected that the reason why Crawljax generates more false alarms is the error in algorithm of crawling; Crawljax does not recognize the clickable page elements with same tag attributes, even they are located in the different pages of the web application.

In our study we concentrate on easing the use of model checker by supporting web crawler output models as input, generating CTL properties with user interface guidance and replaying counterexample scenarios on web browser. Also, users are integrated as part of the tool by manual editing of the model. Moreover, within the user study evaluation, participants who have never

used model checking method succeed to use MCWebSoft and verify web applications with model checker, NuSMV with our system. Also, results of perceived usefulness show that participants find our tool useful for animating detected error traces.

7.3 Limitations

Our limitations in terms of in which extent we verify the web applications is dependent on the model of web applications extracted from web crawlers, Crawljax and Micro-Crawler. Therefore some type of errors could not be detected by MCWebSoft. For example, if a user can access a private page by a specific URL with a web browser, MCWebSoft cannot find this error. The reason is that the web crawlers successively click the clickable elements to create the navigation map of web applications and they cannot record the web pages, states which cannot be reached from index page. Moreover, if the web application has an error which causes to access private pages without filling true inputs or if the web application fails after the login action, our tool would not recognize this deficiency. Nevertheless, by investigating the models extracted from web crawlers, users can expose these types of errors.

Besides, limitations of our study are given in two aspects; limitation of MCWebSoft and limitations of the evaluation. The former one includes the analysis of the completeness and soundness of the fault detection process. The latter one, the limitation of the evaluation, explains the validity problems of the evaluation process.

7.3.1 Completeness and Soundness

The web crawlers used for this study, Micro-Crawler and Crawljax, do not ensure that the complete web application will be crawled and recorded. Mesbah et al. state that the set of found states in Crawljax in the inferred state machine, SFG, is not complete [4]. With the evaluations conducted with Micro-Crawler, we have detected that the state machine generated from Micro-Crawler is not complete too. In our implementation, an option to modify the model of web crawler is provided, and in this way users can complete the missing states in the model of web application. However, modifying the web crawler model may not be enough to ensure its completeness. Therefore, our study does not guarantee the completeness, in other words, the errors found may not be the full set of bugs in the application in their category.

Due to the incompleteness of models of web crawlers, our method does not produce sound results. In other words, the errors found in the model may not be really a fault in web applications. We call these type of errors as false alarms. In our study, in order to create an intermediate web model, only one of the web crawler's, Micro-Crawler or Crawljax, output model is required. Combining the output of these two crawlers would be an option for decreasing the possibility of incomplete results and for relieving the user from making a choice between two web crawlers. However, this process is not straightforward for the differences in capabilities of web crawlers, and hence extent of output models. Only Crawljax is able to crawl web applications, based on specific data inputs, and extract the map of authorized part of web applications. Therefore, we cannot combine two web crawler outputs for that private, part of application. Moreover, we created XML-based model, the intermediate web model format in order to be independent from web crawler outputs. Although we have used Micro-Crawler and Crawljax as two web crawlers in this study, the intermediate

web model allows to add another type of web crawler to our implementation. For all of these reasons, combining two web crawler outputs is not covered in this study.

7.3.2 Threads to Validity of the Evaluation

In the findings of evaluations, there can be several threats to our evaluation in terms of its external validity. Moreover, there are several concerns about repeatability of our evaluation which are presented in this section

First, we present the concerns about repeatability of our experiments. The performance of MCWebSoft was measured on one personal computer, the results of time evaluation may not be the same when we repeat them in another computer due to performance of computer that the evaluation is conducted with. Moreover, the web applications that we used as subject for our evaluations are open to public domain. Therefore, there is a possibility that they can be modified over time which may lead to be impossible to duplicate the evaluations with same results.

External Validity describes for which extent the result of our evaluations can be generalized. In this category, we have threads for three steps of our evaluations.

First, the user study is conducted with limited sample size; 8 participants with the knowledge of programming at different levels and no knowledge about model checking was investigated. The participants have different background in web development or programming. However, this may not be enough to generalize the results of our user group to all people who are in the same category.

Second, the error detection evaluation for navigation properties is performed with 30 web applications and most of them are small sized web applications. The technologies used for development of subject web applications have a large variety. However, the evaluation results for subject web applications may not be enough to be generalized to all web applications. Also we cannot generalize the results of access control evaluation to all web applications. While measuring the performance of access control error detection, we used only one subject web application and by fault seeding we check the error detection against three versions of this site. This may not be enough for concluding that the results hold for all web applications.

7.4 Future Work

In this study, an approach to combine usage of web crawlers to model checking method to verify web applications and increasing the user experience about the model checking is developed and implemented as a software application. Although, MCWebSoft makes it possible to apply model checking to web applications by users with no experience, in the user study, ease of use scores of the tool is not so high. Therefore, it is aimed to improve the usability of the tool by developing a more professional virtual interface in the future.

In Section 7.3.1, combining the output of both Micro-Crawler and Crawljax is mentioned as an option for decreasing incompleteness of web models. A method for combining two web crawlers can be developed in future work. In this study, we aim to be independent from web crawlers, so we did not focus on improving web crawler outputs.

MCWebSoft verifies access of one type of user, which is the authenticated user, at one single time of execution. However, in general, web applications could have other user types with different roles having separate navigation paths and access control concerns. Also in Section 4.2, we defined our model of web applications in Kripke structure as compatible with more than one type of user by describing guard conditions as a set. However, while implementing we consider one guard condition, which is the login information of regular users. With our current implementation, to check different types of user roles in web applications, users should extract the output of web application with respect to each one of user types, then input them to our tool one by one to control their access. In the future, the function of verification of multiple different users' access control at one time is aimed to be implemented.

Moreover, the number of property categories can be increased in the future. In this study, MCWebSoft can check navigation and access control properties. Other than these, certificates of browsers can be verified or we can verify the web pages which require HTTPS protocol by checking URL components that is saved for each webstates. In addition, back button is not considered in this study. This restriction is due to the models web crawlers extracted. Including back buttons would be a simple extension, but user had to manually provide this information.

REFERENCES

[1] Upadhyay, P. (2012). The Role of Verification and Validation in System Development Life Cycle, *IOSR Journal of Computer Engineering (IOSRJCE) ISSN: 2278-0661 Volume 5, Issue 1 (Sep-Oct. 2012)*, PP 17-20

[2] Homma, K., Izumi, S., Abe, Y., Takahashi, K., & Togashi, A. (2010). Using the model checker spin for web application design. *In Applications and the Internet (SAINT), 2010 10th IEEE/IPSJ International Symposium on* (pp. 137-140). IEEE.

[3] De Alfaro, L. (2001). Model Checking the World Wide Web, *In Computer Aided Verification* (pp. 337-349). Springer Berlin Heidelberg

[4] Mesbah, A., van Deursen, A., & Lenselink, S. (2012). Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB), 6(1)*, 3.

[5] Le Breton, G., Maronnaud, F., & Hallé, S. (2013). Automated exploration and analysis of Ajax web applications with WebMole. *In Proceedings of the 22nd international conference on World Wide Web companion* (pp. 245-248). International World Wide Web Conferences Steering Committee.

[6] Cimatti, A., Clarke, E., Giunchiglia, F., & Roveri, M. (1999). NuSMV: A new symbolic model verifier. *In Computer Aided Verification* (pp. 495-499). Springer Berlin Heidelberg.

[7] Baier, C., & Katoen, J. (2008). *Principles of model checking*. Cambridge, Mass., London: MIT Press

[8] Fraser, G., Wotawa, F., & Ammann, P. E. (2009). Testing with model checkers: a survey. Software Testing, *Verification and Reliability, 19(3)*, 215-261.

[9] Clarke, E. M., and Grumberg, O. (1999). *Model Checking*. Cambridge, Mass.: MIT Press.

[10] Huth, M., & Ryan, M. (2000). *Logic in computer science: Modelling and reasoning about systems*. Cambridge, England: Cambridge University Press.

[11] McMillan, K. L. (1993). *Symbolic model checking*. Boston: Kluwer Academic Press.

[12] "Internet Growth and Statistics: Credits and Background". [Online] Available at: http://www.mit.edu/people/mkgray/net/background.html [Accessed 25 Jun. 2015].

[13] Pinkerton, B. (1994). Finding what people want: Experiences with the WebCrawler. *In Proceedings of the Second International World Wide Web Conference (Vol. 94*, pp. 17-20).

[14] van Deursen, A., Mesbah, A., & Nederlof, A. (2015). Crawl-based analysis of web applications: Prospects and challenges. *Science of Computer Programming, 97*, 173-180.

[15] Le Hégaret, P., Wood, L. and Robie, J. (2000). *What is the Document Object Model?*. [Online] W3.org. Available at: http://www.w3.org/TR/DOM-Level-2-Core/introduction.html [Accessed 9 Aug. 2015].

[16] Mesbah, A., & Van Deursen, A. (2009). Invariant-based automatic testing of AJAX user interfaces. *In Proceedings of the 31st International Conference on Software Engineering* (pp. 210-220). IEEE Computer Society.

[17] Bezemer, C.-P., Mesbah, A., & Van Deursen, A. (2009). Automated security testing of Web widget interactions. *In Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE'09). ACM Press, New York, NY,* 81–91.

[18] Roest, D., Mesbah, A., & van Deursen, A. (2010). Regression testing Ajax applications: Coping with dynamism. *In Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST'10). IEEE Computer Society, Los Alamitos, CA,* 128–136.

[19] Mesbah, A. and Prasad, M. R. (2011). Automated cross-browser compatibility testing. *In Proceedings of the 33rd ACM/IEEE International Conference on Software Engineering (ICSE'11). ACM Press, New York, NY*, 561–570.

[20] Demarty, G., Maronnaud, F., Le Breton, G., & Hallé, S. (2013). SiteHopper: Abstracting navigation state machines for the efficient verification of web applications. *In Web Services and Formal Methods* (pp. 103-117). Springer Berlin Heidelberg.

[21] Clarke, E. M., Jha, S., Lu, Y., Veith, H. (2002). Tree-like counterexamples in model checking. *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science, Washington, DC, U.S.A. IEEE Computer Society: Silver Spring, MD, 2002*; 19–29. ISBN 0-7695-1483-9.

[22] Bojańczyk, M. (2008). The common fragment of ACTL and LTL. *In Foundations of Software Science and Computational Structures* (pp. 172-185). Springer Berlin Heidelberg.

[23] Andrews, T., Qadeer, S., Rajamani, S. K., Rehof, J., & Xie, Y. (2004). Zing: A model checker for concurrent software. *In Computer Aided Verification* (pp. 484-487). Springer Berlin Heidelberg.

[24] Ball, T., & Rajamani, S. K. (2001). The SLAM toolkit. *In Computer aided verification* (pp. 260-264). Springer Berlin Heidelberg.

[25] Musuvathi, M., Park, D. Y., Chou, A., Engler, D. R., & Dill, D. L. (2002). CMC: A pragmatic approach to model checking real code. *ACM SIGOPS Operating Systems Review*, *36(SI)*, 75-88.

[26] Havelund, K., & Pressburger, T. (2000). Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer, 2(4),* 366-381.

[27] Jhala, R., & Majumdar, R. (2009). Software model checking. *ACM Computing Surveys (CSUR), 41(4)*, 21.

[28] Alfaro, L., Henzinger, T. A., & Mang, F. Y. (2001). *MCWEB: A model-checking tool for Web site debugging*. In World Wide Web Conference (WWW)--Poster Proceedings.

[29] Licata, D. R., & Krishnamurthi, S. (2004). Verifying interactive Web programs. *In Automated Software Engineering, 2004. Proceedings. 19th International Conference* on (pp. 164-173). IEEE.

[30] Deutsch, A., Marcus, M., Sui, L., Vianu, V., & Zhou, D. (2005). A verifier for interactive, data-driven web applications. *In Proceedings of the 2005 ACM SIGMOD international conference on Management of data* (pp. 539-550). ACM.

[31] Di Sciascio, E., Donini, F. M., Mongiello, M., & Piscitelli, G. (2002). AnWeb: a system for automatic support to web application verification. *In Proceedings of the 14th international conference on Software engineering and knowledge engineering* (pp. 609-616). ACM.

[32] Haydar, M., Petrenko, A., & Sahraoui, H. (2004). Formal verification of web applications modeled by communicating automata. *In Formal Techniques for Networked and Distributed Systems–FORTE 2004* (pp. 115-132). Springer Berlin Heidelberg.

[33] Di Sciascio, E., Donini, F. M., Mongiello, M., Totaro, R., & Castelluccia, D. (2005). Design verification of web applications using symbolic model checking. *In Web Engineering* (pp. 69-74). Springer Berlin Heidelberg.

[34] Castelluccia, D., Mongiello, M., Ruta, M., & Totaro, R. (2006). Waver: A model checking-based tool to verify web application design. *Electronic notes in theoretical Computer Science, 157(1)*, 61-76.

[35] Knapp, A., & Zhang, G. (2006). Model Transformations for Integrating and Validating Web Application Models. *In Modellierung (Vol. 22*, p. 24).

[36] Brambilla, M., Cabot, J., & Moreno, N. (2007). Tool support for model checking of web application designs. *In Web Engineering* (pp. 533-538). Springer Berlin Heidelberg.

[37] Brambilla, M., Deutsch, A., Sui, L., & Vianu, V. (2005). The role of visual tools in a web application design and verification framework: a visual notation for LTL formulae. *In Web Engineering* (pp. 557-568). Springer Berlin Heidelberg.

[38] Miao, H., & Zeng, H. (2007). *Model checking-based verification of web application. In Engineering Complex Computer Systems, 2007. 12th IEEE International Conference* on (pp. 47-55). IEEE.

[39] Kung, D. C., Liu, C. H., & Hsia, P. (2000). An object-oriented web test model for testing web applications. *In Quality Software, 2000. Proceedings. First Asia-Pacific Conference* on (pp. 111-120). IEEE.

[40] Guerra, E., Sanz, D., Díaz, P., & Aedo, I. (2007). A transformation-driven approach to the verification of security policies in web designs. *In Web Engineering* (pp. 269-284). Springer Berlin Heidelberg.

[41] Díaz, P., Montero, S., & Aedo, I. (2005). Modelling hypermedia and web applications: the Ariadne Development Method. *Information Systems, 30(8)*, 649-673.

[42] Han, M., & Hofmeister, C. (2006). Modeling and verification of adaptive navigation in web applications. *In Proceedings of the 6th international conference on Web engineering* (pp. 329-336). ACM.

[43] Hallé, S., Ettema, T., Bunch, C., & Bultan, T. (2010). Eliminating navigation errors in web applications via model checking and runtime enforcement of navigation state machines. *In Proceedings of the IEEE/ACM international conference on Automated software engineering* (pp. 235-244). ACM.

[44] Alalfi, M. H., Cordy, J. R., & Dean, T. R. (2009). Modelling methods for web application verification and testing: state of the art. Software Testing, *Verification and Reliability, 19(4)*, 265-296.

[45] Huang, Y. W., Yu, F., Hang, C., Tsai, C. H., Lee, D. T., & Kuo, S. Y. (2004). Verifying web applications using bounded model checking. *In Dependable Systems and Networks, 2004 International Conference* on (pp. 199-208). IEEE.

[46] Ball, T., Naik, M., & Rajamani, S. K. (2003). From symptom to cause: localizing errors in counterexample traces. *In ACM SIGPLAN Notices (Vol. 38, No. 1*, pp. 97-105). ACM.

[47] Corbett, J. C., Dwyer, M. B., & Hatcliff, J. (2000). Bandera: A source-level interface for model checking java programs. *In Proceedings of the 22nd international conference on Software engineering* (pp. 762-765). ACM.

[48] Anupam, V., Freire, J., Kumar, B., & Lieuwen, D. (2000). Automating Web navigation with the WebVCR. *Computer Networks, 33(1)*, 503-517.

[49] Benedikt, M., Freire, J., & Godefroid, P. (2002). VeriWeb: Automatically testing dynamic web sites. *In Proceedings of 11th International World Wide Web Conference (WW W'2002).*

[50] Torsel, A. M. (2013). A Testing tool for Web applications using a domain-specific modelling language and the NuSMV model checker. *In Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference* on (pp. 383-390). IEEE.

[51] Zeng, H., & Miao, H. (2007). Auto-generating test sequences for web applications. *In Web Engineering* (pp. 301-305). Springer Berlin Heidelberg.

[52] Li, L., Miao, H., & Chen, S. (2010). Test generation for web applications using model-checking. *In Software Engineering Artificial Intelligence Networking and Parallel/Distributed Computing (SNPD), 2010 11th ACIS International Conference on* (pp. 237-242). IEEE.

[53] Ammann, P. E., Black, P. E., & Majurski, W. (1998, December). Using model checking to generate tests from specifications. *In Formal Engineering Methods, 1998. Proceedings. Second International Conference* on (pp. 46-54). IEEE.

[54] "Top 10 2013-Top 10 – OWASP". [Online] Available at: https://www.owasp.org/index.php/Top_10_2013-Top_10 [Accessed 25 Jun. 2015].

[55] "CWE -2011 CWE/SANS Top 25 Most Dangerous Software Errors". [Online] Available at: http://cwe.mitre.org/top25/index.html#CWE-862 [Accessed 25 Jun. 2015].

[56] "Selenium - Web Browser Automation". [Online] Available at: http://www.seleniumhq.org/ [Accessed 25 Jun. 2015].

[57] Davis, F. D. (1989). Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS quarterly*, 319-340.

[58] Miller, M., Ranier, R. and Corley, J. (2003). Predictors of Engagement and Participation in an On-Line Course. *Online Journal of Distance Learning Administration, [Online] 6(1).*

Available at: http://www.westga.edu/~distance/ojdla/spring61/miller61.htm [Accessed 9 Aug. 2015].

[59] Morris, G. M., Dillon, A., (1997). *The Influence of User Perceptions on Software Utilization: Application and Evaluation of a Theoretical Model of Technology Acceptance* [Online]. Available at: https://ford.ischool.utexas.edu/html/2081/1143/User%20Preceptions.htm

[60] “HTTrack Website Copier - Free Software Offline Browser (GNU GPL)”. [Online] Available at: https://www.httrack.com/ [Accessed 25 Jun. 2015].

[61] “Web Developer”. [Online] Available at: http://chrispederick.com/work/web-developer/ [Accessed 9 Aug. 2015].

[62] Schnoebelen, P. (2002). The Complexity of Temporal Logic Model Checking. *Advances in modal logic, 4*(393-436), 35.

APPENDICES

APPENDIX A: INTERMEDIATE WEB MODEL OF RUNNING EXAMPLE


```
<?xml version='1.0' encoding='UTF-8' ?>
<webstates>
        <webstate>
                <name>
                index
                </name>
                <url>
                http://localhost/RunningExample/index.php
                </url>
        </webstate>
        <webstate>
                <name>
                state9
                </name>
                <url>
                http://localhost/RunningExample/Dead%20End.html
                </url>
        </webstate>
        <webstate>
                <name>
                state5
                </name>
                <url>
                http://localhost/RunningExample/Normal.html
                </url>
        </webstate>
        <webstate>
        <name>
                state7
                </name>
                <url>
```

```
		http://localhost/RunningExample/Ajax.html
		</url>
	</webstate>
	<webstate>
		<name>
		state2
		</name>
		<url>
		http://localhost/RunningExample/welcome.php
		</url>
	</webstate>
	<webstate>
		<name>
		state1
		</name>
		<url>
		http://localhost/RunningExample/login.php
		</url>
	</webstate>
	<webstate>
		<name>
		state4
		</name>
		<url>
		http://localhost/RunningExample/Login%20Test.html
		</url>
	</webstate>
	<webstate>
		<name>
		state13
		</name>
		<url>
		http://localhost/RunningExample/Ajax.html
		</url>
	</webstate>
	<webstate>
		<name>
		state10
		</name>
		<url>
		http://localhost/RunningExample/Broken%20Link.html
		</url>
	</webstate>
	<webstate>
		<name>
		state14
		</name>
		<url>
		http://localhost/RunningExample/page.html
		</url>
	</webstate>
</webstates>
```

```
<elements>
    <element>
        <xpath>
        /HTML[1]/BODY[1]/P[1]/A[1]
        </xpath>
    </element>
    <element>
        <xpath>
        /HTML[1]/BODY[1]/FORM[1]/INPUT[3]
        </xpath>
    </element>
    <element>
        <xpath>
        /HTML[1]/BODY[1]/DIV[1]/A[2]
        </xpath>
    </element>
    <element>
        <xpath>
        /HTML[1]/BODY[1]/TABLE[1]/TBODY[1]/TR[1]/TD[1]/TABLE[1]/TBODY[1]/TR[1
        ]/TD[1]/A[1]
        </xpath>
    </element>
    <element>
        <xpath>
        /HTML[1]/BODY[1]/TABLE[1]/TBODY[1]/TR[1]/TD[1]/TABLE[1]/TBODY[1]/TR[1
        ]/TD[1]/A[2]
        </xpath>
    </element>
    <element>
        <xpath>
        /HTML[1]/BODY[1]/DIV[1]/A[1]
        </xpath>
    </element>
    <element>
        <xpath>
        /HTML[1]/BODY[1]/TABLE[1]/TBODY[1]/TR[1]/TD[1]/TABLE[2]/TBODY[1]/TR[1
        ]/TD[1]/A[2]
        </xpath>
    </element>
    <element>
        <xpath>
        /HTML[1]/BODY[1]/HEADER[1]/DIV[1]/A[1]
        </xpath>
    </element>
    <element>
        <xpath>
        /HTML[1]/BODY[1]/TABLE[1]/TBODY[1]/TR[1]/TD[1]/TABLE[3]/TBODY[1]/TR[1
        ]/TD[1]/A[2]
        </xpath>
    </element>
    <element>
        <xpath>
```

/HTML[1]/BODY[1]/TABLE[1]/TBODY[1]/TR[1]/TD[1]/TABLE[4]/TBODY[1]/TR[1
]/TD[1]/A[2]
</xpath>
</element>
<element>
<xpath>
/HTML[1]/BODY[1]/TABLE[1]/TBODY[1]/TR[1]/TD[1]/TABLE[5]/TBODY[1]/TR[1
]/TD[1]/A[2]
</xpath>
</element>
<element>
<xpath>
/HTML[1]/BODY[1]/BUTTON[1]
</xpath>
</element>
<element>
<xpath>
/HTML[1]/BODY[1]/TABLE[1]/TBODY[1]/TR[1]/TD[1]/A[1]
</xpath>
</element>
</elements>
<edges>
<edge>
<from>
index
</from>
<to>
state1
</to>
<xpathOfElement>
/HTML[1]/BODY[1]/P[1]/A[1]
</xpathOfElement>
</edge>
<edge>
<from>
state2
</from>
<to>
index
</to>
<xpathOfElement>
/HTML[1]/BODY[1]/DIV[1]/A[2]
</xpathOfElement>
</edge>
<edge>
<from>
index
</from>
<to>
state4
</to>
<xpathOfElement>

/HTML[1]/BODY[1]/TABLE[1]/TBODY[1]/TR[1]/TD[1]/TABLE[1]/TBODY[1]/TR[1
]/TD[1]/A[1]
</xpathOfElement>
</edge>
<edge>
<from>
state2
</from>
<to>
state5
</to>
<xpathOfElement>
/HTML[1]/BODY[1]/TABLE[1]/TBODY[1]/TR[1]/TD[1]/TABLE[1]/TBODY[1]/TR[1
]/TD[1]/A[2]
</xpathOfElement>
</edge>
<edge>
<from>
state5
</from>
<to>
state2
</to>
<xpathOfElement>
/HTML[1]/BODY[1]/DIV[1]/A[1]
</xpathOfElement>
</edge>
<edge>
<from>
state2
</from>
<to>
state7
</to>
<xpathOfElement>
/HTML[1]/BODY[1]/TABLE[1]/TBODY[1]/TR[1]/TD[1]/TABLE[2]/TBODY[1]/TR[1
]/TD[1]/A[2]
</xpathOfElement>
</edge>
<edge>
<from>
state7
</from>
<to>
state2
</to>
<xpathOfElement>
/HTML[1]/BODY[1]/HEADER[1]/DIV[1]/A[1]
</xpathOfElement>
</edge>
<edge>
<from>

state2
</from>
<to>
state9
</to>
<xpathOfElement>
/HTML[1]/BODY[1]/TABLE[1]/TBODY[1]/TR[1]/TD[1]/TABLE[3]/TBODY[1]/TR[1
]/TD[1]/A[2]
</xpathOfElement>
</edge>
<edge>
<from>
state2
</from>
<to>
state10
</to>
<xpathOfElement>
/HTML[1]/BODY[1]/TABLE[1]/TBODY[1]/TR[1]/TD[1]/TABLE[4]/TBODY[1]/TR[1
]/TD[1]/A[2]
</xpathOfElement>
</edge>
<edge>
<from>
state10
</from>
<to>
state2
</to>
<xpathOfElement>
/HTML[1]/BODY[1]/DIV[1]/A[1]
</xpathOfElement>
</edge>
<edge>
<from>
state2
</from>
<to>
state4
</to>
<xpathOfElement>
/HTML[1]/BODY[1]/TABLE[1]/TBODY[1]/TR[1]/TD[1]/TABLE[5]/TBODY[1]/TR[1
]/TD[1]/A[2]
</xpathOfElement>
</edge>
<edge>
<from>
state7
</from>
<to>
state13
</to>

<xpathOfElement>
/HTML[1]/BODY[1]/BUTTON[1]
</xpathOfElement>
</edge>
<edge>
<from>
state10
</from>
<to>
state14
</to>
<xpathOfElement>
/HTML[1]/BODY[1]/TABLE[1]/TBODY[1]/TR[1]/TD[1]/A[1]
</xpathOfElement>
</edge>
<edge>
<from>
state13
</from>
<to>
state2
</to>
<xpathOfElement>
/HTML[1]/BODY[1]/HEADER[1]/DIV[1]/A[1]
</xpathOfElement>
</edge>
</edges>

APPENDIX B: NuSMV MODEL OF RUNNING EXAMPLE

```
MODULE main
VAR

login: boolean;
webstate:{index,state9,state5,state7,state2,state1,state4,state13,state10,state14,dead_end_webstate};
element:{_HTML_1__BODY_1__P_1__A_1_,_HTML_1__BODY_1__FORM_1__INPUT_3_,_HTML_
1__BODY_1__DIV_1__A_2_,_HTML_1__BODY_1__TABLE_1__TBODY_1__TR_1__TD_1__TABL
E_1__TBODY_1__TR_1__TD_1__A_1_,_HTML_1__BODY_1__TABLE_1__TBODY_1__TR_1__TD
_1__TABLE_1__TBODY_1__TR_1__TD_1__A_2_,_HTML_1__BODY_1__DIV_1__A_1_,_HTML_1
__BODY_1__TABLE_1__TBODY_1__TR_1__TD_1__TABLE_2__TBODY_1__TR_1__TD_1__A_2_
,_HTML_1__BODY_1__HEADER_1__DIV_1__A_1_,_HTML_1__BODY_1__TABLE_1__TBODY_1
__TR_1__TD_1__TABLE_3__TBODY_1__TR_1__TD_1__A_2_,_HTML_1__BODY_1__TABLE_1__
TBODY_1__TR_1__TD_1__TABLE_4__TBODY_1__TR_1__TD_1__A_2_,_HTML_1__BODY_1__T
ABLE_1__TBODY_1__TR_1__TD_1__TABLE_5__TBODY_1__TR_1__TD_1__A_2_,_HTML_1__B
ODY_1__BUTTON_1_,_HTML_1__BODY_1__TABLE_1__TBODY_1__TR_1__TD_1__A_1_,null_el
ement};
ASSIGN
init(login):=FALSE;
init(webstate):=index;
init(element):=null_element;

next(webstate):= case
webstate=index & next(element)=_HTML_1__BODY_1__P_1__A_1_:state1;
webstate=state2 & next(element)=_HTML_1__BODY_1__DIV_1__A_2_:index;
webstate=index &
next(element)=_HTML_1__BODY_1__TABLE_1__TBODY_1__TR_1__TD_1__TABLE_1__TBODY_
1__TR_1__TD_1__A_1_:state4;
webstate=state2 &
next(element)=_HTML_1__BODY_1__TABLE_1__TBODY_1__TR_1__TD_1__TABLE_1__TBODY_
1__TR_1__TD_1__A_2_:state5;
webstate=state5 & next(element)=_HTML_1__BODY_1__DIV_1__A_1_:state2;
webstate=state2 &
next(element)=_HTML_1__BODY_1__TABLE_1__TBODY_1__TR_1__TD_1__TABLE_2__TBODY_
1__TR_1__TD_1__A_2_:state7;
webstate=state7 & next(element)=_HTML_1__BODY_1__HEADER_1__DIV_1__A_1_:state2;
webstate=state2 &
next(element)=_HTML_1__BODY_1__TABLE_1__TBODY_1__TR_1__TD_1__TABLE_3__TBODY_
1__TR_1__TD_1__A_2_:state9;
webstate=state2 &
next(element)=_HTML_1__BODY_1__TABLE_1__TBODY_1__TR_1__TD_1__TABLE_4__TBODY_
1__TR_1__TD_1__A_2_:state10;
webstate=state10 & next(element)=_HTML_1__BODY_1__DIV_1__A_1_:state2;
webstate=state2 &
next(element)=_HTML_1__BODY_1__TABLE_1__TBODY_1__TR_1__TD_1__TABLE_5__TBODY_
1__TR_1__TD_1__A_2_:state4;
webstate=state7 & next(element)=_HTML_1__BODY_1__BUTTON_1_:state13;
webstate=state10 &
next(element)=_HTML_1__BODY_1__TABLE_1__TBODY_1__TR_1__TD_1__A_1_:state14;
```

```
webstate=state13 & next(element)=_HTML_1__BODY_1__HEADER_1__DIV_1__A_1_:state2;


webstate=state1 &  next(login)= TRUE & next(element)=_HTML_1__BODY_1__DIV_1__A_2_:state2;
TRUE:dead_end_webstate;
esac;

next(element):= case
webstate=state1:{_HTML_1__BODY_1__DIV_1__A_2_};
webstate=index:{_HTML_1__BODY_1__P_1__A_1_,_HTML_1__BODY_1__TABLE_1__TBODY_1_
_TR_1__TD_1__TABLE_1__TBODY_1__TR_1__TD_1__A_1_};
webstate=state5:{_HTML_1__BODY_1__DIV_1__A_1_};
webstate=state7:{_HTML_1__BODY_1__HEADER_1__DIV_1__A_1_,_HTML_1__BODY_1__BUTT
ON_1_};
webstate=state2:{_HTML_1__BODY_1__DIV_1__A_2_,_HTML_1__BODY_1__TABLE_1__TBODY
_1__TR_1__TD_1__TABLE_1__TBODY_1__TR_1__TD_1__A_2_,_HTML_1__BODY_1__TABLE_1
__TBODY_1__TR_1__TD_1__TABLE_2__TBODY_1__TR_1__TD_1__A_2_,_HTML_1__BODY_1_
_TABLE_1__TBODY_1__TR_1__TD_1__TABLE_3__TBODY_1__TR_1__TD_1__A_2_,_HTML_1_
_BODY_1__TABLE_1__TBODY_1__TR_1__TD_1__TABLE_4__TBODY_1__TR_1__TD_1__A_2_,
_HTML_1__BODY_1__TABLE_1__TBODY_1__TR_1__TD_1__TABLE_5__TBODY_1__TR_1__TD
_1__A_2_};
webstate=state13:{_HTML_1__BODY_1__HEADER_1__DIV_1__A_1_};
webstate=state10:{_HTML_1__BODY_1__DIV_1__A_1_,_HTML_1__BODY_1__TABLE_1__TBOD
Y_1__TR_1__TD_1__A_1_};
webstate=dead_end_webstate:null_element;
TRUE:element;
esac;

next(login):= case
webstate=state1& next(element)= _HTML_1__BODY_1__DIV_1__A_2_:TRUE;
webstate=state2& next(element)= _HTML_1__BODY_1__DIV_1__A_2_:FALSE;
TRUE: login;
esac;

--INDEX IS REACHABLE FROM ALL STATES
SPEC AG(webstate=state9 -> EF(webstate=index));
SPEC AG(webstate=state5 -> EF(webstate=index));
SPEC AG(webstate=state7 -> EF(webstate=index));
SPEC AG(webstate=state2 -> EF(webstate=index));
SPEC AG(webstate=state1 -> EF(webstate=index));
SPEC AG(webstate=state4 -> EF(webstate=index));
SPEC AG(webstate=state13 -> EF(webstate=index));
SPEC AG(webstate=state10 -> EF(webstate=index));
SPEC AG(webstate=state14 -> EF(webstate=index));


--THERE IS NO DEAD END
SPEC AG(webstate=index & login = TRUE -> !EX(webstate=dead_end_webstate));
SPEC AG(webstate=state1 & login = TRUE -> !EX(webstate=dead_end_webstate));
SPEC AG(webstate=state9 -> !EX(webstate=dead_end_webstate));
SPEC AG(webstate=state5 -> !EX(webstate=dead_end_webstate));
SPEC AG(webstate=state7 -> !EX(webstate=dead_end_webstate));
```

```
SPEC AG(webstate=state2 -> !EX(webstate=dead_end_webstate));
SPEC AG(webstate=state4 -> !EX(webstate=dead_end_webstate));
SPEC AG(webstate=state13 -> !EX(webstate=dead_end_webstate));
SPEC AG(webstate=state10 -> !EX(webstate=dead_end_webstate));
SPEC AG(webstate=state14 -> !EX(webstate=dead_end_webstate));

--ACCESS CONTROL PROPERTY
SPEC !EF((webstate=state9) & login =FALSE);
--ACCESS CONTROL PROPERTY
SPEC !EF((webstate=state5) & login =FALSE);
--ACCESS CONTROL PROPERTY
SPEC !EF((webstate=state7) & login =FALSE);
--ACCESS CONTROL PROPERTY
SPEC !EF((webstate=state2) & login =FALSE);
--ACCESS CONTROL PROPERTY
SPEC !EF((webstate=state4) & login =FALSE);
--ACCESS CONTROL PROPERTY
SPEC !EF((webstate=state13) & login =FALSE);
```

APPENDIX C: OUTPUT OF NuSMV FOR RUNNING EXAMPLE

```
*** This is NuSMV 2.5.4 (compiled on Fri Oct 28 14:13:29 UTC 2011)
*** Enabled addons are: compass
*** For more information on NuSMV see <http://nusmv.fbk.eu>
*** or email to <nusmv-users@list.fbk.eu>.
*** Please report bugs to <nusmv-users@fbk.eu>

*** Copyright (c) 2010, Fondazione Bruno Kessler

*** This version of NuSMV is linked to the CUDD library version 2.4.1
*** Copyright (c) 1995-2004, Regents of the University of Colorado

*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat
*** Copyright (c) 2003-2005, Niklas Een, Niklas Sorensson

-- specification AG (webstate = state9 -> EF webstate = index)  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  login = FALSE
  webstate = index
  element = null_element
-> State: 1.2 <-
  webstate = state1
  element = _HTML_1__BODY_1__P_1__A_1_
-> State: 1.3 <-
  login = TRUE
  webstate = state2
  element = _HTML_1__BODY_1__DIV_1__A_2_
-> State: 1.4 <-
  webstate = state9
  element                                                                  =
_HTML_1__BODY_1__TABLE_1__TBODY_1__TR_1__TD_1__TABLE_3__TBODY_1__TR_1__TD
_1__A_2_
-- specification AG (webstate = state5 -> EF webstate = index)  is true
-- specification AG (webstate = state7 -> EF webstate = index)  is true
-- specification AG (webstate = state2 -> EF webstate = index)  is true
-- specification AG (webstate = state1 -> EF webstate = index)  is true
-- specification AG (webstate = state4 -> EF webstate = index)  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 2.1 <-
  login = FALSE
```

```
  webstate = index
  element = null_element
-> State: 2.2 <-
  webstate = state4
  element                                                                          =
_HTML_1__BODY_1__TABLE_1__TBODY_1__TR_1__TD_1__TABLE_1__TBODY_1__TR_1__TD
_1__A_1_
-- specification AG (webstate = state13 -> EF webstate = index)  is true
-- specification AG (webstate = state10 -> EF webstate = index)  is true
-- specification AG (webstate = state14 -> EF webstate = index)  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 3.1 <-
  login = FALSE
  webstate = index
  element = null_element
-> State: 3.2 <-
  webstate = state1
  element = _HTML_1__BODY_1__P_1__A_1_
-> State: 3.3 <-
  login = TRUE
  webstate = state2
  element = _HTML_1__BODY_1__DIV_1__A_2_
-> State: 3.4 <-
  webstate = state10
  element                                                                          =
_HTML_1__BODY_1__TABLE_1__TBODY_1__TR_1__TD_1__TABLE_4__TBODY_1__TR_1__TD
_1__A_2_
-> State: 3.5 <-
  webstate = state14
  element = _HTML_1__BODY_1__TABLE_1__TBODY_1__TR_1__TD_1__A_1_
-- specification AG ((webstate = index & login = TRUE) -> !(EX webstate = dead_end_webstate))  is true
-- specification AG ((webstate = state1 & login = TRUE) -> !(EX webstate = dead_end_webstate))  is true
-- specification AG (webstate = state9 -> !(EX webstate = dead_end_webstate))  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 4.1 <-
  login = FALSE
  webstate = index
  element = null_element
-> State: 4.2 <-
  webstate = state1
  element = _HTML_1__BODY_1__P_1__A_1_
-> State: 4.3 <-
  login = TRUE
  webstate = state2
  element = _HTML_1__BODY_1__DIV_1__A_2_
-> State: 4.4 <-
  webstate = state9
```

```
  element =
_HTML_1__BODY_1__TABLE_1__TBODY_1__TR_1__TD_1__TABLE_3__TBODY_1__TR_1__TD
_1__A_2_
-> State: 4.5 <-
  webstate = dead_end_webstate
-- specification AG (webstate = state5 -> !(EX webstate = dead_end_webstate))  is true
-- specification AG (webstate = state7 -> !(EX webstate = dead_end_webstate))  is true
-- specification AG (webstate = state2 -> !(EX webstate = dead_end_webstate))  is true
-- specification AG (webstate = state4 -> !(EX webstate = dead_end_webstate))  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 5.1 <-
  login = FALSE
  webstate = index
  element = null_element
-> State: 5.2 <-
  webstate = state4
  element =
_HTML_1__BODY_1__TABLE_1__TBODY_1__TR_1__TD_1__TABLE_1__TBODY_1__TR_1__TD
_1__A_1_
-> State: 5.3 <-
  webstate = dead_end_webstate
-- specification AG (webstate = state13 -> !(EX webstate = dead_end_webstate))  is true
-- specification AG (webstate = state10 -> !(EX webstate = dead_end_webstate))  is true
-- specification AG (webstate = state14 -> !(EX webstate = dead_end_webstate))  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 6.1 <-
  login = FALSE
  webstate = index
  element = null_element
-> State: 6.2 <-
  webstate = state1
  element = _HTML_1__BODY_1__P_1__A_1_
-> State: 6.3 <-
  login = TRUE
  webstate = state2
  element = _HTML_1__BODY_1__DIV_1__A_2_
-> State: 6.4 <-
  webstate = state10
  element =
_HTML_1__BODY_1__TABLE_1__TBODY_1__TR_1__TD_1__TABLE_4__TBODY_1__TR_1__TD
_1__A_2_
-> State: 6.5 <-
  webstate = state14
  element = _HTML_1__BODY_1__TABLE_1__TBODY_1__TR_1__TD_1__A_1_
-> State: 6.6 <-
  webstate = dead_end_webstate
-- specification !(EF (webstate = state9 & login = FALSE))  is true
-- specification !(EF (webstate = state5 & login = FALSE))  is true
```

```
-- specification !(EF (webstate = state7 & login = FALSE))  is true
-- specification !(EF (webstate = state2 & login = FALSE))  is true
-- specification !(EF (webstate = state4 & login = FALSE))  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 7.1 <-
  login = FALSE
  webstate = index
  element = null_element
-> State: 7.2 <-
  webstate = state4
  element =
_HTML_1__BODY_1__TABLE_1__TBODY_1__TR_1__TD_1__TABLE_1__TBODY_1__TR_1__TD
_1__A_1_
-- specification !(EF (webstate = state13 & login = FALSE))  is true
```

APPENDIX D: A JUNIT CLASS TO REPLAY FOR RUNNING EXAMPLE

```
package junit;

import java.util.concurrent.TimeUnit;
import javax.swing.JOptionPane;
import javax.swing.JDialog;
import org.junit.*;
import org.openqa.selenium.*;
import org.openqa.selenium.firefox.FirefoxDriver;
public class Template1{

  private WebDriver driver;
  private String baseUrl;


   @Before
  public void setUp() throws Exception {
    driver = new FirefoxDriver();
    driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);
    driver.manage().window().maximize();
    baseUrl = "http://localhost/RunningExample/index.php";
}


    @Test
  public void test() throws Exception {
    driver.get(baseUrl);
    openDialog("clicking                                                                    to
/HTML[1]/BODY[1]/TABLE[1]/TBODY[1]/TR[1]/TD[1]/TABLE[1]/TBODY[1]/TR[1]/TD[1]/A[1]");
   driver.findElement(By.xpath("
/HTML[1]/BODY[1]/TABLE[1]/TBODY[1]/TR[1]/TD[1]/TABLE[1]/TBODY[1]/TR[1]/TD[1]/A[1]")).c
lick();
}

   @After
  public void tearDown() throws Exception {
     Thread.sleep(10000);
     String note="violation of the AG (webstate = state4 -> EF webstate = index)  property, Homepage is not
reachable from this page";
     openDialog(note);
     driver.close();
}

  public void openDialog(String note){
     if(note.length()>200){
     JOptionPane optionPane = new NarrowOptionPane();
```

```
    optionPane.setMessage(note);
    optionPane.setMessageType(JOptionPane.INFORMATION_MESSAGE);
    JDialog dialog = optionPane.createDialog(null, "Width 100");
    dialog.setVisible(true);
}
else
     JOptionPane.showMessageDialog(null, note);
}

  class NarrowOptionPane extends JOptionPane {
    NarrowOptionPane() {
}

  public int getMaxCharactersPerLineCount() {
    return 100;
}
}}
```

APPENDIX E: CLASS DIAGRAM OF MCWebSoft

APPENDIX F: THE QUESTIONNAIRE

WEB SİTESİ MODEL DENETLE ARACI KULLANILABİLİRLİK ANKETİ

Eğitim Durumuz, Mezun Olduğunuz Bölüm:

____

Mesleğiniz:

____

Bundan sonraki sorular, deney sırasında kullandığınız yazılım aracıyla ilgili deneyimlerinizle ilgilidir. Sorular hakkında derecelendirme 1'den 5'e kadar, yapılacaktır.

1. Kesinlikle Katılmıyorum
2. Katılmıyorum
3. Nötr
4. Katılıyorum
5. Kesinlikle Katılıyorum

Model Denetlemeyle web sitelerini doğrulamak için geliştirilen bu yazılım aracıyla ilgili kendinize en yakın bulduğunuz cevabı yuvarlak içine alınız.

SORULAR

1. Yararlı bir yazılım aracı olduğunu düşünüyorum.

   1 2 3 4 5

2. Web sitelerinin hatalarının bulunmasında kolaylık sağlayacak bir araçtır.

   1 2 3 4 5

3. Bulunan hataların canlandırılmasının yararlı olacağını düşünüyorum.

1 2 3 4 5

4. Kullanımı kolay bir araçtır.

1 2 3 4 5

5. Kullanım açısından anlaşılabilir bir araçtır.

1 2 3 4 5

Öneriler:

_ _

APPENDIX G: NAVIGATIONAL VERIFICATION RESULTS

Table 16: Web technologies of the subject web applications

ID	URL	Technology
W1	http://www.seleniumhq.org/	Google App Engine Web Server Java Programming Language PayPal Payment Processor
W2	http://www.brain.ii.metu.edu.tr/eng_dworld.html	Apache Web Server
W3	http://www.acarturk.net/	Apache Web Server PleskHosting Panel WordPress 4.1CMSBlog PHP Programming Language
W4	http://www.metu.edu.tr/~merkan/	Apache Web Server
W5	http://www.uyms2015.yasar.edu.tr/2015/index.html	JQuery JavaScript Framework
W6	http://www.puppeteershadowplay.com/index.html	Google Analytics Analytics LiteSpeed Web Server
W7	http://www.tr.china-embassy.org/tur/	Apache 2.4.4 Web Server mod_jk 1.2.37 Web Server Extension UNIX Operating System Apache Tomcat Web Server HeadJS (50% sure) JavaScript Framework
W8	http://www.kktcbe.org.tr/www/tr/anasayfa.asp	jQueryJavaScript Framework Microsoft ASP.NETWeb Framework PleskHosting Panel Windows Server Operating System
W9	http://www.mfa.gov.hu/kulkepviselet/TR/tr/	(could not be determined)
W10	http://www.ceng.metu.edu.tr/~altingovde/	Apache Web Server

Table 16 (continued): Web technologies of the subject web applications

W11	http://www.kovan.ceng.metu.edu.tr/~sinan/	Apache 2.2.22 Web Server Debian Operating System
W12	http://www.eonur.eu/	jQuery JavaScript Framework
W13	http://www.ceng.metu.edu.tr/~manguoglu/	Apache Web Server StatCounter Analytics HeadJS (50% sure) JavaScript Framework
W14	http://www.ceng.metu.edu.tr/~polat/	Apache Web Server Google Font API Font Script
W15	http://www.metu.edu.tr/~koksalan/	Apache Web Server
W16	http://www.metu.edu.tr/~stugba/index.html	Apache Web Server
W17	http://www.metu.edu.tr/~aliy/	Apache Web Server
W18	http://www.metu.edu.tr/~sdag/index.html#	Apache Web Server Google Font API Font Script
W19	http://www.metu.edu.tr/~ymetin/	Apache Web Server
W20	http://www.metu.edu.tr/~sbilmis/	Apache Web Server
W21	http://www.retina.cs.bilkent.edu.tr/	JQuery JavaScript Framework
W22	http://www.cs.bilkent.edu.tr/~erman/	Apache 2.2.15 Web Server OpenSSL 0.9.8p Web Server Extension PHP 5.3.2 (40% sure) Programming Language UNIX Operating System mod_ssl 2.2.15 Web Server Extension
W23	http://www.art.bilkent.edu.tr/ozguc/index.html	Apache Web Server
W24	http://www.cs.bilkent.edu.tr/~ozturk/index.html	Apache 2.2.15 Web Server OpenSSL 0.9.8p Web Server Extension PHP 5.3.2 (40% sure) Programming Language UNIX Operating System mod_ssl 2.2.15 Web Server Extension
W25	http://www.umram.bilkent.edu.tr/~ergin/	jQuery JavaScript Framework
W26	http://www.volkan.bilkent.edu.tr/	Apache Web Server
W27	http://www.isbcs2015.ii.metu.edu.tr/index.html	JQuery JavaScript Framework

Table 16 (continued): Web technologies of the subject web applications

W28	http://www.atilkurttekin.com/	Google Analytics Analytics IIS 6.0 Web Server Microsoft ASP.NET (50% sure) Web Framework Windows Server Operating System
W29	http://www.fince.bilkent.edu.tr/	Apache Web Server
W30	http://www.infobil.bilkent.edu.tr	Apache Web Server

Table 17: Results of verification of navigation properties with Micro-Crawler

ID	URL	Property			
		Index page is reachable from all states of web application		There is no dead end in web application	
		# of Detected Real faults	# of Detected False Alarms	# of Detected Real faults	# of Detected False Alarms
W1	http://www.seleniumhq.org/	.00	4.00	2.00	.00
W2	http://www.brain.ii.metu.edu.tr/eng_dworld.html	2.00	4.00	.00	2.00
W3	http://www.acarturk.net/	4.00	.00	.	.
W4	http://www.metu.edu.tr/~merkan/	.00	7.00	.	.
W5	http://www.uyms2015.yasar.edu.tr/2015/index.html	.	.	.	.
W6	http://www.puppeteershadowplay.com/index.html	.	.	.	.
W7	http://www.tr.china-embassy.org/tur/	.	.	.	.
W8	http://www.kktcbe.org.tr/www/tr/anasayfa.asp	.	.	.	.
W9	http://www.mfa.gov.hu/kulkepviselet/TR/tr/	.	.	.	.
W10	http://www.ceng.metu.edu.tr/~altingovde/	2.00	.00	.	.
W11	http://www.kovan.ceng.metu.edu.tr/~sinan/	.	.	.00	1.00
W12	http://www.eonur.eu/	.00	1.00	3.00	.00
W13	http://www.ceng.metu.edu.tr/~manguoglu/	3.00	.00	2.00	1.00
W14	http://www.ceng.metu.edu.tr/~polat/	2.00	1.00	.	.
W15	http://www.metu.edu.tr/~koksalan/	.	.	.	.

Table 17 (continued): Results of verification of navigation properties with Micro-Crawler

ID	URL	Property			
		Index page is reachable from all states of web application		There is no dead end in web application	
		# of DetectedReal faults	# of Detected False Alarms	# of Detected Real faults	# of Detected False Alarms
W16	http://www.metu.edu.tr/~stugba/index.html	.	.	.00	1.00
W17	http://www.metu.edu.tr/~aliy/	.	.	.	.
W18	http://www.metu.edu.tr/~sdag/index.html#	.	.	1.00	.00
W19	http://www.metu.edu.tr/~ymetin/	1.00	.00	2.00	.00
W20	http://www.metu.edu.tr/~sbilmis/	2.00	.00	1.00	.00
W21	http://www.retina.cs.bilkent.edu.tr/	1.00	.00	.	.
W22	http://www.cs.bilkent.edu.tr/~erman/	.	.	.	.
W23	http://www.art.bilkent.edu.tr/ozguc/index.html	.	.	4.00	2.00
W24	http://www.cs.bilkent.edu.tr/~ozturk/index.html	4.00	2.00	3.00	.00
W25	http://www.umram.bilkent.edu.tr/~ergin/	3.00	.00	.	.
W26	http://www.volkan.bilkent.edu.tr/	.	.	1.00	.00
W27	http://www.isbcs2015.ii.metu.edu.tr/index.html	.	.	1.00	1.00
W28	http://www.atilkurttekin.com/	1.00	5.00	.	.
W29	http://www.fince.bilkent.edu.tr/	.	.	.00	2.00
W30	http://www.infobil.bilkent.edu.tr	.	.	.	.

Table 18: Results of verification of navigation properties with Crawljax

ID	URL	Property			
		Index page is reachable from all states of web application		There is no dead end in web application	
		# of Detected Real faults	# of Detected False Alarms	# of Detected Real faults	# of Detected False Alarms
W1	http://www.seleniumhq.org/	.00	13.00	.00	6.00
W2	http://www.brain.ii.metu.edu.tr/eng_dworld.html	1.00	14.00	1.00	10.00
W3	http://www.acarturk.net/	.00	8.00	.00	7.00
W4	http://www.metu.edu.tr/~merkan/	.00	8.00	.00	7.00
W5	http://www.uyms2015.yasar.edu.tr/2015/index.html	.00	14.00	.00	14.00
W6	http://www.puppeteershadowplay.com/index.html	.00	6.00	.	.
W7	http://www.tr.china-embassy.org/tur/	3.00	42.00	3.00	27.00
W8	http://www.kktcbe.org.tr/www/tr/anasayfa.asp	.00	37.00	.00	34.00
W9	http://www.mfa.gov.hu/kulkepviselet/TR/tr/	19.00	216.00	19.00	111.00
W10	http://www.ceng.metu.edu.tr/~altingovde/	1.00	2.00	1.00	3.00
W11	http://www.kovan.ceng.metu.edu.tr/~sinan/	2.00	5.00	2.00	5.00
W12	http://www.eonur.eu/	3.00	9.00	3.00	6.00
W13	http://www.ceng.metu.edu.tr/~manguoglu/	.00	4.00	.00	1.00
W14	http://www.ceng.metu.edu.tr/~polat/	1.00	6.00	1.00	5.00
W15	http://www.metu.edu.tr/~koksalan/	4.00	7.00	4.00	4.00

Table 18 (continued): Results of verification of navigation properties with Crawljax

ID	URL	Property			
		Index page is reachable from all states of web application		There is no dead end in web application	
		# of Detected Real faults	# of Detected False Alarms	# of Detected Real faults	# of Detected False Alarms
W16	http://www.metu.edu.tr/~stugba/index.html	.00	3.00	.00	3.00
W17	http://www.metu.edu.tr/~aliy/	.00	13.00	.00	13.00
W18	http://www.metu.edu.tr/~sdag/index.html#	.00	2.00	.00	2.00
W19	http://www.metu.edu.tr/~ymetin/	1.00	16.00	1.00	6.00
W20	http://www.metu.edu.tr/~sbilmis/	3.00	.00	3.00	7.00
W21	http://www.retina.cs.bilkent.edu.tr/	2.00	14.00	5.00	10.00
W22	http://www.cs.bilkent.edu.tr/~erman/	1.00	11.00	1.00	8.00
W23	http://www.art.bilkent.edu.tr/ozguc/index.html	.00	11.00	.00	1.00
W24	http://www.cs.bilkent.edu.tr/~ozturk/index.html	3.00	11.00	3.00	8.00
W25	http://www.umram.bilkent.edu.tr/~ergin/	28.00	16.00	28.00	10.00
W26	http://www.volkan.bilkent.edu.tr/	.00	8.00	.00	3.00
W27	http://www.isbcs2015.ii.metu.edu.tr/index.html	.00	7.00	.00	7.00
W28	http://www.atilkurttekin.com/	.00	2.00	.00	2.00
W29	http://www.fince.bilkent.edu.tr/	.00	109.00	.00	101.00
W30	http://www.infobil.bilkent.edu.tr	1.00	26.00	1.00	17.00

Table 19: Results of the performance evaluation of subject web applications used in verification of navigation properties with average values of webstates, edges and time measurements

ID	URL	Average # of webstates	Average # of edges	Average Time (ms)	DOM Size (KB)	File Size (MB)
W1	http://www.seleniumhq.org/	9.50	12.00	753.50	27.74	11.80
W2	http://www.brain.ii.metu.edu.tr/eng_dworld.html	19.50	125.50	1303.50	9.02	20.90
W3	http://www.acarturk.net/	9.50	30.00	862.50	7.10	11.30
W4	http://www.metu.edu.tr/~merkan/	8.50	12.00	676.50	15.38	2.59
W5	http://www.uyms2015.yasar.edu.tr/2015/index.html	15.00	22.00	837.50	6.77	17.70
W6	http://www.puppeteershadowplay.com/index.html	10.00	18.50	685.50	4.50	3.84
W7	http://www.tr.china-embassy.org/tur/	32.00	102.00	2061.00	106.07	37.60
W8	http://www.kktcbe.org.tr/www/tr/anasayfa.asp	29.00	59.00	1650.50	50.69	65.30
W9	http://www.mfa.gov.hu/kulkepviselet/TR/tr/	124.50	146.00	13645.00	16.07	8.27
W10	http://www.ceng.metu.edu.tr/~altingovde/	5.00	8.00	683.00	8.20	1.20
W11	http://www.kovan.ceng.metu.edu.tr/~sinan/	10.00	38.00	996.50	12.30	154.00
W12	http://www.eonur.eu/	29.50	60.50	1061.50	69.53	.97
W13	http://www.ceng.metu.edu.tr/~manguoglu/	4.50	7.50	957.50	33.67	2.98
W14	http://www.ceng.metu.edu.tr/~polat/	12.00	44.00	770.00	6.44	1.57
W15	http://www.metu.edu.tr/~koksalan/	10.50	33.00	1017.50	6.65	.70
W16	http://www.metu.edu.tr/~stugba/index.html	6.00	41.50	716.00	4.62	.12
W17	http://www.metu.edu.tr/~aliy/	14.50	48.00	1060.50	5.67	.16

Table 19 (continued): Results of the performance evaluation of subject web applications used in verification of navigation properties with average values of webstates, edges and time measurements

ID	URL	Average # of webstates	Average # of edges	Average Time (ms)	DOM Size (KB)	File Size (MB)
W18	http://www.metu.edu.tr/~sdag/index.html#	4.00	12.50	586.50	15.36	1.47
W19	http://www.metu.edu.tr/~ymetin/	27.00	111.00	1589.50	4.69	26.90
W20	http://www.metu.edu.tr/~sbilmis/	13.00	26.00	727.50	2.39	21.70
W21	http://www.retina.cs.bilkent.edu.tr/	15.00	35.00	939.00	6.99	12.90
W22	http://www.cs.bilkent.edu.tr/~erman/	10.50	24.50	869.00	3.30	38.20
W23	http://www.art.bilkent.edu.tr/ozguc/index.html	13.50	52.00	1035.00	4.97	.20
W24	http://www.cs.bilkent.edu.tr/~ozturk/index.html	16.00	48.00	1188.50	14.48	1026.00
W25	http://www.umram.bilkent.edu.tr/~ergin/	33.50	217.00	2897.00	36.07	222.00
W26	http://www.volkan.bilkent.edu.tr/	8.00	11.00	923.00	46.04	297.00
W27	http://www.isbcs2015.ii.metu.edu.tr/index.html	8.00	32.00	3862.00	3.89	.46
W28	http://www.atilkurttekin.com/	7.50	17.50	671.00	7.03	200.00
W29	http://www.fince.bilkent.edu.tr/	60.50	88.00	6695.00	10.51	16.80
W30	http://www.infobil.bilkent.edu.tr	21.00	38.00	10585.00	3.97	22.60

Table 20: The numbers of webstates, edges and the values of time measurements for each web crawler output whose average values are given in Table 4 (C means Crawljax, M means Micro-Crawler)

ID	URL	Web Crawler	# of Webstates	# of Edges	Time (ms)
W1	http://www.seleniumhq.org/	C	14	17	1039
		M	5	7	468
W2	http://www.brain.ii.metu.edu.tr/eng_dworld.html	C	20	34	1224
		M	19	217	1383
W3	http://www.acarturk.net/	C	9	9	775
		M	10	51	950
W4	http://www.metu.edu.tr/~merkan/	C	9	10	817
		M	8	14	536
W5	http://www.uyms.yasar.edu.tr/2015/index.html	C	15	15	1123
		M	15	29	552
W6	http://www.puppeteershadowplay.com/index.html	C	10	18	887
		M	10	19	484
W7	http://www.tr.china-embassy.org/tur/	C	54	140	3433
		M	10	64	689
W8	http://www.kktcbe.org.tr/www/tr/anasayfa.asp	C	38	39	2198
		M	20	79	1103
W9	http://www.mfa.gov.hu/kulkepviselet/TR/tr/	C	236	259	26501
		M	13	33	789
W10	http://www.ceng.metu.edu.tr/~altingovde/	C	4	3	821
		M	6	13	545
W11	http://www.kovan.ceng.metu.edu.tr/~sinan/	C	12	23	1204
		M	8	53	789
W12	http://www.eonur.eu/	C	13	12	937
		M	46	109	1186
W13	http://www.ceng.metu.edu.tr/~manguoglu/	C	5	4	842
		M	4	11	1073

Table 20 (continued): The numbers of webstates, edges and the values of time measurements for each web crawler output whose average values are given in Table 4 (C means Crawljax, M means Micro-Crawler)

W14	http://www.ceng.metu.edu.tr/~polat/	C	11	14	794
		M	13	74	746
W15	http://www.metu.edu.tr/~koksalan/	C	12	15	952
		M	9	51	1083
W16	http://www.metu.edu.tr/~stugba/index.html	C	6	22	811
		M	6	61	621
W17	http://www.metu.edu.tr/~aliy/	C	14	13	1219
		M	15	83	902
W18	http://www.metu.edu.tr/~sdag/index.html#	C	4	6	653
		M	4	19	520
W19	http://www.metu.edu.tr/~ymetin/	C	27	59	1496
		M	27	163	1683
W20	http://www.metu.edu.tr/~sbilmis/	C	20	43	800
		M	6	9	655
W21	http://www.retina.cs.bilkent.edu.tr/	C	22	45	1324
		M	8	25	554
W22	http://www.cs.bilkent.edu.tr/~erman/	C	13	12	1162
		M	8	37	576
W23	http://www.art.bilkent.edu.tr/ozguc/index.html	C	19	67	1104
		M	8	37	966
W24	http://www.cs.bilkent.edu.tr/~ozturk/index.html	C	16	18	1351
		M	16	78	1026
W25	http://www.umram.bilkent.edu.tr/~ergin/	C	45	108	3445
		M	22	326	2349
W26	http://www.volkan.bilkent.edu.tr/	C	9	10	963
		M	7	12	883

Table 20 (continued): The numbers of webstates, edges and the values of time measurements for each web crawler output whose average values are given in Table 4 (C means Crawljax, M means Micro-Crawler)

W27	http://www.isbcs2015.ii.metu.edu.tr/index.html	C	8	7	6988
		M	8	57	736
W28	http://www.atilkurttekin.com/	C	4	6	564
		M	11	29	778
W29	http://www.fince.bilkent.edu.tr/	C	111	122	12100
		M	10	54	1290
W30	http://www.infobil.bilkent.edu.tr	C	29	41	17726
		M	13	35	3444