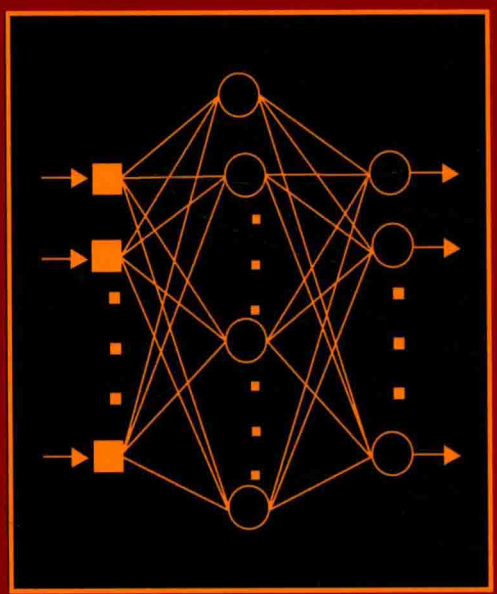
# Advances in COMPUTERS

## Volume 93



Edited by
**ATIF MEMON**

Series Editors
**Ali Hurson and Atif Memon**

VOLUME NINETY THREE

# ADVANCES IN COMPUTERS

Edited by

**ATIF MEMON**
*University of Maryland*
*4115 A.V. Williams Building*
*College Park, MD 20742, USA*
*Email: atif@cs.umd.edu*

ELSEVIER

Amsterdam • Boston • Heidelberg • London
New York • Oxford • Paris • San Diego
San Francisco • Singapore • Sydney • Tokyo
Academic Press is an imprint of Elsevier

Academic Press is an imprint of Elsevier
225 Wyman Street, Waltham, MA 02451, USA
525 B Street, Suite 1800, San Diego, CA 92101-4495, USA
The Boulevard, Langford Lane, Kidlington, Oxford, OX51GB, UK
32, Jamestown Road, London NW1 7BY, UK
Radarweg 29, PO Box 211, 1000 AE Amsterdam, The Netherlands

First edition 2014



**Library of Congress Cataloging-in-Publication Data**
A catalog record for this book is available from the Library of Congress

**British Library Cataloguing-in-Publication Data**
A catalogue record for this book is available from the British Library

ISBN: 978-0-12-800162-2



Transferred to Digital Printing 2014

# PREFACE

This volume of Advances in Computers is the 93rd in this series. This series, which has been continuously published since 1960, presents in each volume four to seven chapters describing new developments in software, hardware, or uses of computers.

Even though software has been around for a number of decades, its quality continues to elude computer engineers and scientists alike. This is largely due to the multi-faceted nature of software quality. Consider, for example, web applications that have become pervasive only in the last two decades. They have needed new techniques to assess their quality because they involve dynamic code creation and interpretation. Moreover, as software has become larger in size, its quality assurance imposes prohibitive performance overhead that has prompted the need for new, more efficient solutions. As software interconnectivity has become pervasive, security has quickly grown to be a dominant aspect of its quality, requiring novel techniques. Finally, as software changes throughout its lifetime, quality assurance approaches need to adapt with change. In this volume, we touch upon all these issues.

This volume is a compilation of a set of five chapters that study issues of software security, quality, and evolution. The authors of these chapters are world leaders in their fields of expertise. Together their chapters provide a view into the state-of-the-art in their respective fields of expertise.

Chapter 1, entitled "Recent Advances in Web Testing," provides a comprehensive overview of the research carried out in the last ten years to support web testing with automated tools. The chapter categorize the works available in the literature according to the specific web testing phase that they address. In particular, it first considers the works aimed at building a navigation model of the web application under test. Such a model is often the starting point for test case derivation. Then, it considers the problem of input generation, because the traversal of a selected navigation path requires that appropriate input data be identified and submitted to the server during test execution. Metrics are introduced and used to assess the adequacy of the test cases constructed from the model. The last part of the chapter is devoted to very recent advancements in the area, focused on rich client web applications, which demand a specific approach to modeling and to test case derivation.

Chapter 2, entitled "Exploiting Hardware Monitoring in Software Engineering," discusses advances in program monitoring, a key component of many software engineering tasks. Traditionally, instrumentation has been used to complete such tasks. However, instrumentation can prohibitively increase the time and especially the memory overhead of an application. As an alternative to instrumentation, hardware monitoring has been shown to aid in developing more efficient techniques. The chapter examines efforts in applying hardware monitoring to a number of software engineering tasks including profiling, dynamic optimization, and software testing. It presents improvements in using instrumentation for monitoring, how hardware mechanisms can provide an alternative, and the success that has been revealed in software engineering research when applying hardware monitoring approaches.

Sound methodologies for constructing security-critical systems are extremely important in order to confront the increasingly varied security threats. As a response to this need, Model-Driven Security has emerged as a specialized Model-Driven Engineering approach for supporting the development of security-critical systems. Chapter 3, entitled "Advances in Model-Driven Security" summarizes the most important developments of Model-Driven Security during the past decade. The chapter starts by building a taxonomy of the most important concepts of this domain, which it uses to describe and evaluate a set of representative and influential Model-Driven Security approaches in the literature. The chapter focuses on the concepts shared by Model-Driven Engineering and Model-Driven Security, allowing the identification of the advantages, disadvantages and open issues when applying Model-Driven Engineering to the Information Security domain.

Chapter 4 is entitled "Adapting Multi-Criteria Decision Analysis for Assessing the Quality of Software Products. Current Approaches and Future Perspectives." Our great reliance on software-based systems and services nowadays requires software products of the highest quality. An essential prerequisite for developing software of guaranteed quality in a predictable way is the ability to model and objectively assess its quality throughout the project lifecycle. A potential approach must handle the abstract and multidimensional character of quality. This chapter leverages the analogies between software quality assessment (SQA) and Multi-Criteria Decision Analysis (MCDA) to investigate how MCDA methods can support SQA. The chapter (1) identifies the most relevant requirements for an SQA method, (2) reviews existing SQA methods regarding potential

benefits from using MCDA methods, and (3) assesses some popular MCDA methods regarding their applicability for SQA. Although a number of SQA methods proposed in recent years already adapt MCDA methods, the exact rationales for selecting a particular method are usually unclear or arbitrary. Usually, neither the goals nor the constraints of SQA are explicitly considered. Existing SQA methods do not meet the relevant requirements and mostly share the same weaknesses independent of whether they employ MCDA or not. In many cases, popular MCDA techniques are unsuitable for SQA because they do not meet its basic constraints, such as handling situations where data are scarce.

Software constantly changes during its life cycle. This phenomenon is particularly prominent in modern software, whose complexity keeps growing and changes rapidly in response to market pressures and user demands. At the same time, developers must assure the quality of this software in a timely manner. Therefore, it is of critical importance to provide developers with effective tools and techniques to analyze, test, and validate their software as it evolves. While techniques for supporting software evolution abound, a conceptual foundation for understanding, analyzing, comparing, and developing new techniques is also necessary for the continuous growth of this field. A key challenge for many of these techniques is to accurately model and compute the effects of changes on the behavior of software systems. Such a model helps understand, compare, and further advance important activities such as change-impact analysis, regression testing, test-suite augmentation, and program merging. Chapter 5, entitled, "Change-Effects Analysis for Evolving Software," describe progress in developing and studying a foundational approach called change-effects analysis. This kind of analysis computes all the differences that a change in the code of a program can cause on every element (e.g., statement) of that program. These differences include not only which program elements are affected by a change, but also how exactly their behavior (i.e., execution frequency and states) is affected.

I hope that you find these articles of interest. If you have any suggestions of topics for future chapters, or if you wish to be considered as an author for a chapter, I can be reached at atif@cs.umd.edu.

**Prof. Atif M Memon, Ph.D. College Park, MD, USA**

# CONTENTS

*Preface* *vii*

**1. Recent Advances in Web Testing** **1**

Paolo Tonella, Filippo Ricca, and Alessandro Marchetto

1. Introduction 2
2. Model Construction 6
3. Input Generation 18
4. Metrics 27
5. Rich Client 36
6. Conclusion 46

References 48

About the Authors 50

**2. Exploiting Hardware Monitoring in Software Engineering** **53**

Kristen R. Walcott-Justice

1. Introduction 54
2. Reducing the Overhead of Program Monitoring 57
3. Foundations in Hardware Monitoring 58
4. Hardware Monitoring in Software Engineering 67
5. Discussion and Future Directions 91
6. Conclusions 96

References 96

About the Author 101

**3. Advances in Model-Driven Security** **103**

Levi Lúcio, Qin Zhang, Phu H. Nguyen, Moussa Amrani, Jacques Klein, Hans Vangheluwe, and Yves Le Traon

1. Introduction 104
2. Model-Driven Engineering 105
3. Model-Driven Security 112
4. Evaluation of Current Model-Driven Security Approaches 118
5. Discussion 136
6. Related Work 141
7. Conclusion 142

List of Acronyms 143

Acknowledgments 144
References 144
About the Authors 151

**4. Adapting Multi-Criteria Decision Analysis for Assessing the Quality of Software Products. Current Approaches and Future Perspectives 153**

Adam Trendowicz and Sylwia Kopczyńska

1. Introduction 155
2. Background 158
3. Study Design and Execution 169
4. Analysis of Existing SQA Methods 187
5. Analysis of Popular MCDA Methods 196
6. Discussion 210
7. Summary and Conclusions 218

Acknowledgments 219
Appendix 220
References 221
About the Authors 225

**5. Change-Effects Analysis for Evolving Software 227**

Raul Santelices, Yiji Zhang, Haipeng Cai, and Siyuan Jiang

1. Introduction 229
2. Background 230
3. Related Approaches 237
4. Foundations of Change-Effects Analysis 242
5. Application: Test-Suite Augmentation Requirements 254
6. Application: Test-Suite Augmentation Requirements 255
7. Application: Demand-Driven Test-Suite Augmentation 261
8. Application: Quantitative Change-Impact Analysis 269
9. Conclusion 277
10. Future Directions 278

References 279
About the Authors 285

*Author Index 287*
*Subject Index 297*
*Contents of Volumes in this Series 307*

CHAPTER ONE

# Recent Advances in Web Testing

**Paolo Tonella*, Filippo Ricca†, and Alessandro Marchetto***

*Fondazione Bruno Kessler, Trento, Italy
†University of Genova, Italy

## Contents

1. Introduction 2
   1.1 Running Example 4
   1.2 Key Problems in Web Testing 5
   1.3 Structure of the Chapter 6
2. Model Construction 6
   2.1 UML Models 8
   2.2 FSM Models 12
   2.3 Identification of Logical Web Pages 15
3. Input Generation 18
   3.1 Manual Techniques 19
   3.2 (Semi-)automatic Techniques 21
      *3.2.1 Hidden Web Crawlers 21*
      *3.2.2 Automating Equivalence Partitioning and Boundary Value Analysis 22*
      *3.2.3 Fuzz Testing 23*
      *3.2.4 User-Session-Based Techniques 24*
      *3.2.5 Symbolic Execution and Constraint Solving Techniques 26*
4. Metrics 27
   4.1 Adequacy Metrics 29
   4.2 Crawlability Metrics 32
5. Rich Client 36
   5.1 Dynamic Analysis 40
   5.2 Model Mining 42
   5.3 RIA Testing 45
6. Conclusion 46
References 48
About the Authors 50

## Abstract

Web applications have become key assets of our society, which depends on web applications for sectors like business, health-care, and public administration. Testing is the most widely used and effective approach to ensure quality and dependability of the software, including web applications. However, web applications are special as

*Advances in Computers*, Volume 93
ISSN 0065-2458, http://dx.doi.org/10.1016/B978-0-12-800162-2.00001-4

compared to traditional software, because they involve dynamic code creation and interpretation and because they implement a specific interaction mode, based on the navigation structure of the web application.

Researchers have investigated approaches and techniques to automate web testing, dealing with the special features of web applications. This chapter contains a comprehensive overview of the research carried out in the last 10 years to support web testing with automated tools. We categorize the works available in the literature according to the specific web testing phase that they address. In particular, we first of all consider the works aiming at building a navigation model of the web application under test. In fact, such a model is often the starting point for test case derivation. Then, we consider the problem of input generation, because the traversal of a selected navigation path requires that appropriate input data are identified and submitted to the server during test execution. Metrics are introduced and used to assess the adequacy of the test cases constructed from the model. The last part of the chapter is devoted to very recent advancements in the area, focused on rich client web applications, which demand a specific approach to modeling and to test case derivation.

## 1. INTRODUCTION

Web applications represent a key, strategic asset of our society. Many of the processes that affect our daily lives are mediated by web applications. Nowadays, people expect to perform tasks related to their work, money, health, public administration, and entertainment on the web. Online banking, e-commerce, e-government, e-health are all terms that refer to the vast application domain and the multitude of application scenarios that involve web applications. As a consequence, web applications must be dependable. Being a critical, strategic asset, their quality and reliability must achieve adequate standards.

While software quality has been investigated for a long time and a huge amount of research works and practical tools are available to help developers deliver the required quality level for traditional software, the same is not true for web applications, because the web technology is quite recent, it differs substantially from that of traditional software and it is rapidly changing. With traditional software, testing is the most prominent approach to ensure that adequate quality standards are met. Testing of traditional software usually involves modeling the system under test, using the model to generate the test cases and to evaluate their adequacy, and defining the oracles that specify whether the behavior observed during testing is compliant with the one expected by the end user. With web applications the key phases of testing remain the same, but the reference model and the related adequacy criteria differ substantially.

For traditional software, a straightforward model of the system under test is provided by the source code and more specifically the control flow programmed in the source code. The control flow graph models the execution flow inside each procedure while the call graph models the invocation of procedures. Such models—and several others built upon them—determine how to generate the test cases and how to evaluate their adequacy. As a simple example, whitebox coverage criteria such as statement/branch coverage demand that all nodes/edges in the control flow graph of the system under test are exercised in at least one test case. This provides a clear guidance for the creation of test cases and for the assessment of their adequacy with respect to the target coverage goal (e.g., 100% branch coverage).

While web applications are still constructed as collections of source code modules, they introduce more dynamism, associated with runtime code generation, and an alternative view of the execution flow, which is represented by the navigation graph. Since they are organized as client-server programs, web applications reside on the server, but the code that is run on the server performs only part of the required computation. Another, relevant part is delegated to the client and the code to be executed on the client is generated dynamically on the server. Such dynamism may have different degrees of complexity. In the simplest case, the code running on the server just retrieves an HTML page which is sent to the client and client side execution is limited to page rendering. In more complex cases, fragments of script code is also sent to the client, for client-side execution. This might involve, for instance, the client-side validation of user input or the creation of graphical effects and advanced interactions on the GUI. An extreme case of dynamism is represented by modern, rich-client web applications. In such a case, the client page is continuously modified by client-side code which interacts asynchronously with the server to obtain data and to execute services. A continuous flow of data and code between client and server occurs dynamically, at runtime, making the notion of source code no longer a static notion, which can be subjected to control flow modeling, as with traditional software.

The navigation view offered by a web application to the end user is crucial to testing. It is complementary to the control flow view, so it does not replace it, but it introduces a different perspective for both test case generation and adequacy assessment. The navigation graph, which describes how the end user can possibly move from one page or one GUI state to another one, introduces another, different coverage dimension, which is equally important as the control flow one for quality assurance. Ensuring that the test cases cover all statements/branches may be insufficient to ensure that all relevant navigation paths have been adequately exercised. A novel modeling approach

is required for web applications, so as to focus on the navigations that are possible, starting from the client-side web page the user interacts with. The navigation model of a web application is the guiding principle and the key reference to derive test cases and to assess whether the web application has been tested enough.

## 1.1 Running Example

In this chapter, we use a running example to illustrate the various techniques and approaches. We consider the same web application introduced in previous works on the subject [13], an e-commerce application for online shopping of products, more specifically, books. This web application provides the typical functionalities involved in e-commerce:

- **[Authentication]** Users are required to have an account to be able to buy a book. This involves the authentication of users through username and password. The credential management system must ensure standard levels of security to the application users.
- **[Product selection]** Users are allowed to search for books and to browse books, according to different criteria (top selling, recommended, etc.). Advanced search involves multiple filters and search constraints (e.g., price threshold).
- **[Cart management]** Selected books are deposited into a virtual cart, stored persistently for each user account and managed according to the user's choices.
- **[Payment and shipping]** Book payment and delivery involves the collection of the information (credit card number, home address, etc.) necessary to perform these actions. Methods for secure payment are employed, so as to ensure that the transaction will take place safely for the end user.

Let us assume that the implementation of this web application contains a bug. In particular, let us assume that the *advanced search* functionality is faulty and ignores the price threshold search criterion. This means that all books satisfying the other search criteria are returned, regardless of their price. To expose this fault, the testing phase for this web application should include the following steps: (1) a navigation model is constructed, which includes the navigation path associated with the *advanced search* functionality; (2) test scenarios are derived from the model and in particular one test scenario will be produced to exercise the *advanced search* functionality, if we apply the test adequacy criterion requesting that all functionalities reachable in the navigation graph must be exercised in at least one test scenario; (3) input data are provided for the selected test scenarios and in particular, the advanced

search fields must be filled in, including the price threshold; (4) test cases are executed and the fault related to advanced search is identified when the returned book list includes also books that exceed the price threshold set by the test case.

The cart management functionality of our running example might be implemented using rich client technologies, such as Ajax. Technically, this means that whenever a book is added to or removed from the cart, a client side script (e.g., written in Javascript) will be executed, so as to modify the Document Object Model (DOM) of the cart web page. This results in the web page being updated to the new state of the cart. However, this is achieved without requesting a new web page from the browser. Notification of the cart update to the browser is instead carried out asynchronously, using the Ajax client-server asynchronous communication facilities.

Since no transition from a page to another page occurs when a rich client modifies the DOM, the navigation model for a rich client web application will not be enough to thoroughly test it. In our running example, changing the state of the cart does not correspond to any navigation path in the navigation graph, since edges in this graph are exclusively associated with page requests sent to the server and HTML response pages provided by the server to the browser. As a consequence, to test the cart management functionality a different modeling approach must be taken. Instead of focusing on the navigation graph, in the case of a rich client web application, the modeling focus shall be on the DOM state. Events that alter the DOM state will be transitions in the rich client DOM model.

A model for the Ajax cart management may consist of a set of states, each characterized by the number of items in the cart (e.g., 0 items, 1 items > 1 items). GUI events that trigger DOM changes (e.g., *addToCart* and *removeFromCart* events) are modeled as transitions between states. Testing the cart functionalities can be turned into the problem of exercising the event sequences in the rich client model of the cart functionality. In this way, relevant behaviors of this functionality, that would go unnoticed in the navigation graph, are exercised during testing.

## 1.2 Key Problems in Web Testing

Based on the example described above, we can categorize the key problems associated with web testing as follows:

- **[Model construction]** A navigation graph of the application under test must be constructed, so as to ensure that all relevant navigation paths are exercised during testing.

- **[Input generation]** Once a test scenario has been derived from the navigation model of the web application, concrete input values must be supplied to make it an actually executable test case.
- **[Metrics]** Adequacy metrics are computed to determine if the set of test cases produced so far provides enough coverage of the web application model or if additional test cases must be produced. Metrics are also useful to characterize the part of a web application that are more difficult to be explored by automated tools, hence requiring manual intervention during testing (these are called *crawlability* metrics).
- **[Rich client]** When scripts running on the client modify the DOM directly and communicate asynchronously with the server, the navigation graph is no longer appropriate as a model for test case derivation and a different model, representing the DOM states and transitions, should be adopted.

### 1.3 Structure of the Chapter

This chapter complements a previously published chapter on the same topic [17]. We share with the previous publication [17] the same high level view of the testing process, consisting of: (1) model construction; (2) test case generation from the model; (3) adequacy metrics and criteria, to decide on the thoroughness of testing. However, we analyze in depth a complementary set of problems: while the previous work [17] is focused on modeling notations (including statistical Markov models, object-oriented models and regular expressions) and portability analysis, we consider the problems of automated model construction, automated input generation and metrics. Moreover, we include also recent works on modeling and testing of rich client web applications.

The rest of this chapter is organized along the four problems identified above. Specifically, Section 2 presents the main approaches for model construction; Section 3 describes the most important input generation techniques; Section 4 deals with metrics; Section 5 provides details on how to model rich client web applications and how to test them.

## 2. MODEL CONSTRUCTION

In web testing, a model-based approach is often adopted to derive navigation sequences that are successively turned into executable test cases [2, 16, 15, 40, 41, 46]. Hence, the problem is how to define a web application model and how to construct it. Since the focus is on navigation sequences, a

web model can be abstractly regarded as a *navigation graph*, whose paths correspond to the various possible navigation sequences. While such a navigation graph is the backbone of almost all proposed test models for web applications, we can distinguish two major families of web models adopted for testing:

- **[UML models]** These models extend Conallen's UML model [11] with test-specific information.
- **[FSM models]** These models adopt a finite state machine-based approach, which incorporates states, transitions and guards.

Since both types of models are based on the same underlying navigation graph, it is possible to define a mapping between them, with limited loss of information.

The problem of constructing a web application model to be used during testing can be approached manually, automatically, or semi-automatically. Automated or semi-automated model construction can be viewed as a reverse engineering problem. Based on static and dynamic information collected for the web application to be modeled, a model is synthesized, sometimes with the help of the tester or requiring the tester to perform some post-construction refinement.

Static analysis alone is not adequate for the reverse engineering of web application models. In fact, the information available statically is severely limited and incomplete. Dynamically generated pages are obtained by concatenating fixed HTML fragments with additional content which is computed at run time, possibly depending on the user input. This means that the actual HTML code of a web page is not known entirely until the web page is actually navigated at run time. Moreover, the web page may include client side scripts (e.g., Javascript code), which are synthesized dynamically at the server side and which might affect the web page structure at run time. In fact, client side scripts have access to the Document Object Model (DOM) of the web page both in read and write mode. This means that at run time the structure of a web page may change due to client side script execution. Additional dynamism may come from late binding to external services invoked by the web application and by dynamic component loading on the server side. Reflection on the server side may introduce even further dynamism. For all these reasons, all approaches for the reverse engineering of a web model take advantage of both static and dynamic information [3, 40, 41, 46].

Each static or dynamically constructed web page is a node in the navigation graph of the web application. In UML models this is represented as a class in the class diagram. In FSM models it is a state. Whatever is the representation, a common problem is that during web navigation

(i.e., dynamic analysis), pages that have different structure and content may indeed represent the same *logical page* in the model. For instance, the page that shows the information of a particular web application user will be different from that shown for another user. However, such pages should not be treated as different nodes of the model. They are in fact the same logical page. The problem of identifying Logical Web Pages (LWP) from the actually navigated pages is a key problem in model construction for web applications and different authors have come out with different solutions to the problem.

In the following, we describe the UML models and the FSM models that have been proposed in the context of model construction for web testing. Then, we analyze in depth the LWP identification problem.

## 2.1 UML Models

Figure 1 shows an excerpt of the UML meta-model that defines Conallen's UML notation for web application models. A static HTML page is a class to which the ≪ClientPage≫ stereotype from the UML meta-model applies. Hyperlinks between pages are modeled as ≪Link≫ associations. A page may contain forms, used to collect user input to be submitted to the server. The ≪Form≫ stereotype applies to the classes that represent the HTML forms. Attributes of ≪Form≫ classes can be stereotyped as ≪InputElement≫, ≪SelectElement≫ or ≪TextAreaElement≫, depending on the kind of input that is collected from the user (respectively, a single text field, a selection among multiple constant values or a multi-line text field). The server script that is executed upon form submission is stereotyped as ≪ServerPage≫. The HTML page it constructs dynamically is obtained by following the

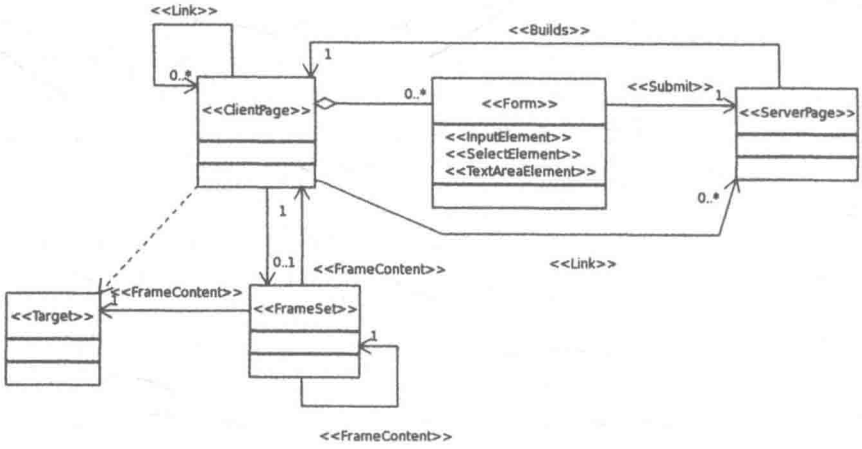



**Fig. 1.** Excerpt of the Conallen UML meta-model.

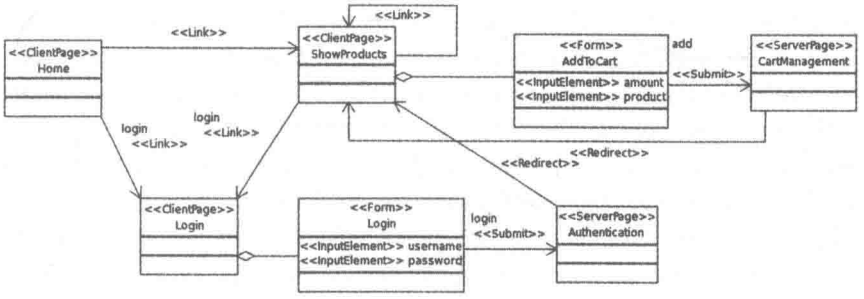



**Fig. 2.** Example of Conallen model.

≪Builds≫ association. Page redirection is modeled by the stereotype ≪Redirect≫ (not shown in the figure for lack of space).

Pages can be divided into frames and page loading can be directed to a specific target frame. This is modeled in Conallen's notation using stereotypes ≪FrameSet≫ and ≪Target≫.

Figure 2 shows an example of Conallen UML model. The web application being modeled is a typical e-commerce application, which includes user authentication, product browsing and selection, cart management, payment and checkout. For lack of space, Fig. 2 shows only a portion of the model, including authentication, product browsing and cart management.

Navigation starts from the static client page *Home*. From this page, users can either decide to authenticate themselves, by clicking on the *login* hyperlink, or to proceed with product browsing, by moving to the client page *ShowProducts*. The *Login* page contains a form with two attributes, *username* and *password*, both stereotyped as ≪InputElement≫. Such form can be submitted to the server. The server page that handles authentication is named *Authentication*.

Product browsing is performed inside the client page *ShowProducts*. Users can add products to their cart by entering the amount of products they want to add into the *amount* text field of the form and by submitting the form to the server page *CartManagement*, which in turn updates the page *ShowProducts* and stores the selected products into the database holding the user cart (not shown in the model).

Several approaches to web analysis and testing [16, 15, 39, 41, 46] are based on the Conallen's web modeling notation and use static and dynamic analysis to reverse engineer a Conallen model for a web application. Tools that implement these approaches are based on navigation in the target web application to extract its web pages and to build a Conallen model of the