UNIVERSITÀ DEGLI STUDI DI PISA

DIPARTIMENTO DI INFORMATICA
DOTTORATO DI RICERCA IN INFORMATICA

PH.D. THESIS

An Integrated Environment For Automated Benchmarking And Validation Of XML-Based Applications

Jinghua Gao

SUPERVISORS	REFEREES
Antonia Bertolino	Franco Turini
Eda Marchetti	Giorgio Ghelli

November 18, 2009

Acknowledgments

I would like to express my gratitude to all those who helped me during the writing of this thesis. Without their encouragement and assistance, this thesis would not have been completed.

My deepest gratitude goes first and foremost to Antonia Bertolino, my supervisor. She gave me much inspiration and encouragement during my academic studies. In this thesis year, she gave much time, and provided valuable guidance in every stage of thesis writing. Without her honest support, impressive kindness and patience, I could not have completed my thesis. Her hard-working attitude and optimistic spirit inspired me not only in this thesis but also for my future life and study.

My great gratitude also goes to Eda Marchetti, who is my minor advisor, and Andrea Polini. They offered great assistance in my research. When I doubted myself, they always brought confidence and great ideas to me. The completion of my research and this thesis can not be separated from their kindly supports.

I would like to thank Professor Pierpaolo Degano and Andrea Maggiolo Schettini who gave me a great and kindly help when I had a difficult time finishing the thesis. I wish to thank also Professor Giorgio Ghelli and his student Luca Pardini, who provided me with an useful case study. I also want to thank my dear colleagues in the Software Engineering lab, Antonino Sabetta, Francesca Lonetti, Guglielmo De Angelis, Cesare Bertolini, Daniela Mulas and Alberto Ribolini. They are my good friends; our lab was like a warm family. They always gave me a hand when I needed help, not only in academics, but also in my life. I enjoyed very much my time with them.

I thank my husband Yan He, he always supports me with his love, understanding and tolerance. Without his encouragement, I probably would never have reached this achievement.

Finally I am grateful to my parents, for their deeply love and continuous support.

Abstract

Testing is the dominant software verification technique used in industry; it is a critical and most expensive process during software development. Along with the increase in software complexity, the costs of testing are increasing rapidly. Faced with this problem, many researchers are working on automated testing, attempting to find methods that execute the processes of testing automatically and cut down the cost of testing.

Today, software systems are becoming complicated. Some of them are composed of several different components. Some projects even required different systems to work together and support each other. The XML have been developed to facilitate data exchange and enhance interoperability among software systems. Along with the development of XML technologies, XML-based systems are used widely in many domains. In this thesis we will present a methodology for testing XML-based applications automatically.

In this thesis we present a methodology called XPT (XML-based Partition Testing) which is defined as deriving XML Instances from XML Schema automatically and systematically. XPT methodology is inspired from the Category-partition method, which is a well-known approach to Black-box Test generation. We follow a similar idea of applying partitioning to an XML Schema in order to generate a suite of conforming instances; in addition, since the number of generated instances soon becomes unmanageable, we also introduce a set of heuristics for reducing the suite; while optimizing the XML Schema coverage. The aim of our research is not only to invent a technical method, but also to attempt to apply XPT methodology in real applications. We have created a proof-of-concept tool, TAXI, which is the implementation of XPT. This tool has a graphic user interface that can guide and help testers to use it easily. TAXI can also be customized for specific applications to build the test environment and automate the whole processes of testing.

The details of TAXI design and the case studies using TAXI in different domains are presented in this thesis. The case studies cover three test purposes. The first one is for functional correctness, specifically we apply the methodology to do the XSLT Testing, which uses TAXI to build an automatic environment for testing the XSLT transformation; the second is for robustness testing, we did the XML database mapping test which tests the data transformation tool for mapping and populate the data from XML Document to XML database; and the third one is for the performance testing, we show XML benchmark that uses TAXI to do the benchmarking of the XML-based applications.

Contents

1 Introduction **11**
- 1.1 Motivations and Objectives 12
- 1.2 Outline of The Thesis 15
- 1.3 Publication List 16

I Background and Related Work **19**

2 Background **21**
- 2.1 Software Testing And Related Concepts 21
  - 2.1.1 White-box Testing and Black-box Testing 23
- 2.2 Category-Partition Methodology 26
- 2.3 Pair-wise Testing 27
- 2.4 XML Schema Fundamentals 28
  - 2.4.1 Brief Introduction of XML Language 28
  - 2.4.2 Document Type Definitions 30
  - 2.4.3 XML Schema 31
  - 2.4.4 Elements and Constraints of XML Schema 32
  - 2.4.5 XML Document Validation 33
- 2.5 Summary 34

3 Related Work **37**
- 3.1 Related XML Testing 37
  - 3.1.1 XML and XML Schema Document Testing 38
  - 3.1.2 Conformance Testing 38
  - 3.1.3 Using Derived XML Document or XML Schema 40
  - 3.1.4 Adopting Perturbation Testing 41
- 3.2 Syntax-based Testing 41
- 3.3 XML Benchmarks 42
- 3.4 Summary 44

II XML-based Partition Testing Methodology and Implementation 45

4 XML-based Partition Testing 47
4.1 Main Methodology of XPT 47
4.1.1 XML Schema Analysis and Rewriting 50
4.1.2 Subschema Derivation 52
4.1.3 Element Identification 54
4.1.4 Element Value Determination 54
4.1.5 Constraint Determination 55
4.1.6 Intermediate Instance Generation 55
4.1.7 Test Case Generation 59
4.2 XPT Test Strategy Selection 61
4.2.1 Weight Assignment 61
4.2.2 Test Strategies 62
4.3 Summary 64

5 TAXI - The Implementation Of XPT 65
5.1 The Overview Of TAXI 65
5.2 TAXI Components 67
5.3 User interface 68
5.3.1 XML Schema Input and Weight Assignment 68
5.3.2 Database Population 70
5.3.3 Instance Browsing 71
5.4 Implementation of XSA 71
5.4.1 Preprocessor 73
5.4.2 Subschema Generation 79
5.4.3 Occurrence Analysis 81
5.4.4 Intermediate Instance Derivation 83
5.4.5 Final Instance Derivation 84
5.5 Implementation of TSS 86
5.5.1 Application of Weights 86
5.5.2 Strategy Selection 87
5.6 Summary 89

6 Cover Set Of TAXI 91
6.1 Test Cases of <element> 91
6.1.1 <element> 92
6.1.2 <element> With “default” Attribute 93
6.1.3 <element> With “fixed” Attribute 93
6.2 Test Cases of <SimpleType> 94
6.2.1 <simpleType> With Child <restriction> 94
6.2.2 <simpleType> With Child <union> 95

6.2.3 <simpleType> With Child “list” 95
6.3 Test Cases of <complexType> 96
6.3.1 <complexType> With Child <simpleContent> 97
6.3.2 <complexType> With Child <complexContent> 98
6.3.3 Test Cases of <sequence> 99
6.3.4 Test Cases of <all> Element 103
6.3.5 Test Cases of <choice> Element 105
6.4 Test Cases Of <Redefine> 109
6.4.1 Redefine The <ComplexType> Element 109
6.4.2 Redefine The <SimpleType> Element 110
6.5 Test Cases Of <any> and <anyAttribute> 110
6.5.1 Test Case Of <any> Element 110
6.5.2 <anyAttribute> Element 110
6.6 Summary 110

III XPT Applications 115

7 XSLT Transformation Testing 117
7.1 Introduction 117
7.2 Automatic Validation of XSLT Stylesheet 118
7.3 Case Study 120
7.3.1 Marc21 and Dublin Core 120
7.3.2 Driving the Generation Process 123
7.3.3 Comparison of TAXI With Other Instance Generators 127
7.4 Summary 128

8 XML Database Mapping Test 129
8.1 Introduction 129
8.2 Black-Box Testing For XML-database Mapping 130
8.3 Case Study 131
8.3.1 MySQL Database 131
8.3.2 XML-database Mapper(myXDM) 131
8.3.3 Testing myXDM Tool 134
8.4 Summary 137

9 The Application Of TAXI to XML Benchmarks 139
9.1 Introduction 139
9.2 Benchmarks for XML-based Applications 140
9.3 Case Study 141
9.3.1 Background 141
9.3.2 XML Schema Validation Benchmark 142

9.4 Advantages and Disadvantages of TAXI In Meeting The Requirements of Benchmarks 146
9.4.1 How TAXI Meets the Requirements of XML Benchmark 146
9.4.2 Limitations of the TAXI tool for Benchmarking 148
9.5 Summary 148

10 Conclusion and Future Work 149

Bibliography 151

Chapter 1

Introduction

The computer is one of the greatest scientific inventions of the twentieth century. At the beginning, the computer was just used as a calculator to solve complex figures, and decipher code. As its use has developed, computer technologies have become more and more important in our lives. They are changing our daily life, the electric appliances running depend on computer technologies bring a lot of convenience and joy to our life. Advanced computer technologies are changing the way of people communicate. With development of the Internet, People can see each other with video cameras even though they are separated by thousands of miles. Computer technologies are also changing the ways we obtain information. Instead of book, now many people prefer to obtain information from the Internet or digital libraries, using one or more key words, search engines will list all related entries; it takes only seconds.

A reliable computer system requires not only the well-designed hardware; a well-functioning and robust software system is also indispensable. Along with widespread use of computer technologies, the concept of software engineering which is the application of a systematic, disciplined, quantifiable approach to development, operation, and maintenance of software [IEE90] has evolved steadily, and becomes more and more significant during computer system development.

In this thesis we concentrate on Software Testing, which is a process, or a series of processes, designed to make sure that computer code does what it was designed to do and that it does not do anything unintended [Mye04]. The activities of testing could run through the whole life cycle of software, and it is the most expensive part of software development. As experimentation, fifty percent or even more of development resources are spent on software testing [Bei90a]. The challenge of testing software adequately with minimal resources, as quickly and thoroughly as possible has become the goal and dream of testing. It has also become the drive of many researchers [Ber07]. To accomplish the goal of testing, people are turning to automated testing, which is "the management and performance of test activities, that include the development and execution of test scripts so as to verify test requirement, using an automated test tool" [ED00].

1.1 Motivations and Objectives

As already presented, during development process Software Testing is the most expensive part. But a system without testing is even more expensive. The further along in the development process an error is found, the more costly it is to correct it.

Unfortunately, the fact is that even with ample and proper testing, we can never ensure that a software product is perfect, even if it has a simple structure. Along with development of computer science, instead of traditional manual testing, Automated Software Testing has become a new hot spot of research. People hope to reduce the costs and increase reliability by performing the analysis by computer, which in generally much quicker and more powerful than the human brain. This desire is good, but if Automated Testing is not developed in a proper manner, it brings higher risks, and becomes even more expensive than manual testing.

There are different methodologies of applying automated software testing. With the White-box Testing strategy, model-checking is a widely-used way for applying automated software testing; for Black-box Testing, deriving test cases from the system specifications is a useful and common way for automated testing. Formal standard organizations such as W3C, OMG and so on are developing standard software specifications; also there is much research focused on the formal description of system specifications.

Since software systems are turning out to be "more and more complicated with components developed by different vendors and using different techniques in different programming languages and even run on different platforms", a concern that becomes increasingly crucial is interoperability. In the pursuit of working interoperability among independently developed systems, industry is increasingly adopting open specifications and binding such specifications to standardised technologies. Such binding technologies must be open in nature, to allow for a wide range of diverse platforms, languages, and tools to be used, and still create compliant applications and content. The eXtensible Markup Language (XML) [Har99] is today the predominant format for data representation and is generally recognized as the standard way to exchange information between remote systems and to bind the specifications [W3C05b]. In few years this language has established itself as the de facto standard form for specifying and exchanging data and documents between almost any digital or web application. Now XML applications are in widespread use for many areas, such as data and information preservation, data interchange, database population and for web applications. Therefore XML testing has become an interesting topic for many organizations and researchers. We are also enthusiastic about testing software systems based on XML. Considering the activities of automated testing, we would like to find a way for carrying out the testing of XML-based applications automatically.

Before starting methodology development, we first reviewed in depth the available literature, and found there are some systems taking XML Documents as input. Sometimes these documents must specially conform to specific rules.

The rules are usually defined by XML Schemas, which could be DTD [DTD96], RELAX NG [dV04], XML Schema, or other schemas. Among these schemas, currently XML Schema is the most powerful and flexible one. Along with the development of

XML, XML Schema has sprung up as a notation for formally describing what constitutes an agreed valid XML Document within an application domain. Compared with other schemas, XML Schema is more widely used by XML developers [Sri]. Also XML Schema actually conforms to the grammar of XML, so it is a well-structured language. XML Schema expresses the basic rules and constraints on data and parameters that diverse classes of systems and web applications exchange; thus it provides an accurate and formalized representation of the input domain in a format suitable for automated processing, which clearly has big potential for test automation.

Based on the characters of XML and XML Schema, we have attempted to find a method that can automatically test XML-based systems. For systems that take XML documents as input, the corresponding XML Schema can be considered as the specification of system input, or the grammar of system required data. Therefore, if we can get a set of XML documents that includes all possible or at least a portion of important derived XML documents from the XML Schema, these documents can be used as test cases to do the Black-box Testing to validate the behavior of XML-based systems.

The first aim of our methodology is "**Automation**". With the increase in software complexity, testing needs more time to design and execute. But with the pressure of market competition, the cycle of software development is shortening. In this case, testing needs to be more efficient and execute within a shorter time frame.

To apply the Automation, first we need to know which activities of software testing those could be applied automatically. According to the activities of normal software testing [Ber04], there are some test activities that could be automated. Those activities become the main activities of automated software testing [MF94].

The activities of automated testing include:

- **Test Condition Identification**: This activity determines the object that needs to be tested, and the conditions of testing according to the test specification. It is the basis of the next steps of the testing.
- **Test Case Design**: Test cases are designed according to the test specification, by determining test cases which comprise specific input values, expected outcomes, and any other information needed for testing.
- **Test Case Generation**: Test case generation builds the test cases by the conditions and constraints of test case design; the test cases should cover the system input and output domains, or the paths on which the system could be run. To cover all values of these domains is difficult, but representative values should be included in the test cases;
- **Test Case Execution**: Test case execution is to run the software under testing with the test cases;
- **Test Result Analysis**: Test result analysis is to check if the actual test outcomes conform to the expected outcomes. The system passes the testing only if all the outcomes are consistent with the expected outcomes.

Based on these five activities of automated testing, the first step is “test condition identification”, we have already identified the XML-based applications as the test object; meanwhile the input domain of the application should be defined by an XML Schema. Then the main work is design and generate the test cases.

For testing purposes, only “Automation” is not enough; the derived test cases should be organized and must cover the representative possible inputs of the system, otherwise the result of testing is not reliable. Therefore the second aim of our methodology is “**Systematization**”.

Systematic testing is executed with a special purpose, it is ordered, planned, and testing designed to be purposefully methodical in its approach. Systematic testing is useful in initial examinations, where testing time is relatively brief, or when learning general software behavior. Systematic testing requires an in-depth analysis of the application and the application’s components at a very granular level. Some systematic testing approaches include equivalence class partitioning, boundary value analysis, combinatorial analysis, state transition testing, basis path testing, etc.

There are several tools that can generate XML Instances automatically, such as Sun XML Generator [XML99], XMLSpy [XML05b] and Stylus Studio [Cor08], but the XML Instances derived from these tools are not systematic. Sun XML Generator generates random instances, XMLSpy and Stylus Studio generate only the fix numbers of instances. It could be sufficient for some kind of testings, and for a lot of testing applications, random generations are not suitable and the test cases are not qualified enough.

There are tools such as [XML04], [tox05] that can derive XML documents from XML Schema systematically, but the generation is based on instructions that are designed and written by the user. The creation of instructions requires the user to have sufficient knowledge of specific XML Documents and the XML Schemas; it also depends greatly on the experience of the user. Our aim is to create a method that can analyse structure of the XML Schema and guide the generation totally automatically and systematically.

After the study, we found that XML Schema is well-structured with clear levels, since it is based on XML language. So it lends itself quite naturally to the application of partition testing, in which a system is tested at its I/O interface by identifying relevant classes of input values, and by systematically choosing some representative test input values for each identified class. The basic assumption behind partition testing is that the input domain can be divided into subdomains, such that, for testing purposes, within each of them the program “behaves the same” (and then for every point within a subdomain the program either succeeds or fails). The subdivision of the input domain into subdomains, according to the basic principle of partition testing, can be done automatically by analyzing the XML Schema elements. By means of the elements, attributes and constraints of the XML Schema, we first divide the schema into a set of sub-schemas; we then gradually divide the sub-schema into sets of the elements that have the same properties of value and other constraints. The test cases actually are the XML Documents generated by combining the values from these sets. If the values in the derived XML Documents are from the sets defined by the XML Schema, then these documents conform to the XML Schema, and we call them valid documents; otherwise the derived documents are invalid.

The automation of test case execution and result analysis are not the same for different applications. Test execution takes the derived XML Documents as input to the system under testing; it is usually easily applied by some simple automatic file reading programs. The automatic tool for test result analysis must be designed specifically by the output of the applications. In our applications, which will be presented in Part III, we will show different ways for analysing the test result automatically according to the application.

As well as automatic testing, we have another goal for our approach, which is "**Usability**". The method should be easily used in actual projects. To do this, we need not only to consider the technical problem of the methodology, but also to think about how to optimize the method and make it suitable for use in testing real software systems.

As presented before, we would like to apply partition testing to the XML Schema, and traditional partition testing usually creates a huge number of test cases. Along with the growth of system parameters, the test case number increases geometrically. For real applications, the number of test cases should match the requirements of the testing, otherwise it spends unnecessary time and resources, and increases the expense of the testing. In order to overcome this problem, we have developed the combinational method to reduce the number of derived test cases, and provide several strategies to make the generation more flexible.

Traditional partition testing does not allow the tester to focus on particular parts of the system, but in the real life testing, it is important to reduce the costs and time of testing. Sometimes selection of relevant test cases is done by the testers manually, and mainly depends on their intuition and experience. If they make the wrong decision, the testing will not get satisfactory results, and additional costs will be incurred for redoing the testing. Our goal is to provides some strategies that help the user to choose the critical parts for testing, and select a suitable set of test cases.

In this thesis we not only provide the theory of the methodology, but also the implementation of the methodology; moreover we focus a lot on practical applications of the methodology. These applications make our method not only a concept, but also a tool that can readily be used to solve real problems.

1.2 Outline of The Thesis

This thesis is divided into three self-contained parts. The first part is the general introduction of background and related works; the second part focuses on the description of the approach; the third part gives case studies for our methodology; and the last part is a description of future work.

- **Part I**: In the first part, we present basic information regarding Software Testing, XML Schema and others information used in our research. We want give the reader the basic information necessary for comprehending the approach presented in the thesis. We also outline related works in this part.

  - *Chapter 2* presents a brief introduction of Software Testing from different points of views, it also includs XML Schema fundamentals and the algorithms Category-partition and Pairwise Testing, which are used in this thesis.
  - *Chapter 3* presents related works from two aspects: first is the related XML based Testings, we classify these testing methods into different aspects. The seond is syntax based testing in which some works are similar to our research. In this chapter we introduce these methodologies and compare them with our approach briefly.

- **Part II**: This part is the description of our algorithm XML-based Partition Testing in (XPT) and the implementation of a proof-of-concept tool TAXI (Testing by Automatically generated XML Instances).
  - *Chapter 4* proposes a practical and automatic approach to XPT, It applies the Category-partition method to the XML Schema, and generates XML Instances automatically and systematically.
  - *Chapter 5* presents the implementation of XPT. To verify the method XPT, we develop a proof-of-concept tool TAXI that implements the XPT method. TAXI has a graphic user interface, that allows the user to generate instances by different strategies. In this chapter we give the process of TAXI implementation.
  - *Chapter 6* presents the cover set of the TAXI tool. The cover set is a part of test cases of TAXI.

- **Part III**: In this part we present three applications of XPT methodology. For each application we give a case study and the comparison with other tools.
  - *Chapter 7* presents the environment for automatically testing XML Document transformation by XSLT.
  - *Chapter 8* shows the application of TAXI for doing Black-box Testing, specifically, we focus on XML database mapping and population.
  - *Chapter 9* presents the application and a case study using TAXI to do XML Benchmarking to evaluate the performance of XML-based systems.

Finally *Chapter 10* presents the conclusion of the thesis, and the ongoing work of this topic in the future.

1.3 Publication List

- Antonia Bertolino, Jinghua Gao, Eda Marchetti, "XML Every-Flavor Testing", Proc. Web Information Systems and Technologies WEBIST 2006, Setubal, Portugal, April 11-13, 2006. This article is a survey of XML Testing. We summarized the

existing XML Tests from different aspects, and organize them into a well-organized structure.

- Antonia Bertolino, Jinghua Gao, Eda Marchetti, Andrea Polini, "Systematic Generation of XML Instances to Test Complex Software Applications", Proc. Rapid Integration in Software Engineering (RISE 2006) Geneve, Switzerland, September 13-15, 2006. This ariticle is the first publication about the XPT method. We described the idea of XPT method, and evaluate the possiblity of using it for testing complex software applications.

- Antonia Bertolino, Jinghua Gao, Eda Marchetti, Andrea Polini, "XModel-Based Testing of XSLT Application", Proc. Web Information Systems and Technologies, WEBIST 2007, Barcelona, Spain 3-6 March, 2007. In this article, we presented how to apply the XPT method for testing XSLT transformations.

- Antonia Bertolino, Jinghua Gao, Eda Marchetti, Andrea Polini, "TAXI - A Tool for XML-based Testing", Proc. 29th International Conference on Software Engineering, ICSE 2007, Minneapolis, USA 20-26 May, 2007. This was a demonstration presented in the demo section of ICSE conference. It was the first time that we showed TAXI tool.

- Antonia Bertolino, Jinghua Gao, Eda Marchetti, Andrea Polini, "Automatic Test Data Generation for XML Schema-based Partition Testing", Proc. 29th International Conference on Software Engineering, Second International workshop on Automation of Software Testing (AST'07) at 29th International conference on Software Engineering (ICSE'07), Minneapolis, USA 26 May 2007. This article presented the details of the proof-of-concept tool TAXI, its possible application domains, and the comparison between TAXI and other tools.

Part I

Background and Related Work

Chapter 2

Background

This chapter provides the background information used in this Thesis. At the beginning of the chapter is a comprehensive overview of software testing with basic concepts and the related methodologies that are used in the thesis. Since our research is focused on automatic testing of XML-based system, this chapter also includes a brief introduction of XML and XML Schema.

The chapter is structured as follows: Section 2.1 presents the concept of software testing and a brief description of software testing classifications. Section 2.2 and Section 2.3 introduce Category-partition methodology and Pair-wise Testing respectively. Section 2.4 describes fundamentals of XML and XML Schema.

2.1 Software Testing And Related Concepts

"Software Testing consists of the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the specified expected behavior." [Ber04].

In the software life cycle, testing is an important and critical process. In software engineering opinion, testing should start as early as possible, and continue throughout the entire software life cycle, but due to the limitation of time, cost and experience, much software is developed without testing, or is tested after the whole process of development is complete. This carries a big risk, because the earlier bugs are found in the system, the easier and cheaper it is to fix them. The costs to fix bugs rise logarithmically; they increase tenfold as time increases.

The activities of testing depend on the process of the software under testing. According to [Ber04], software testing includes seven main activities:

- **Planning**: As stated earlier Software Testing is a process or series of processes; it must be planned and scheduled before being applied. The most important aspect of the test plan is the project management of testing, which includes planning for

equipment, managing personnel, setting the undesirable outcomes, and establishing the required costs in terms of time and effort to ensure the test environment and process run property.

- **Test Case Generation**: Test cases are generated by a particular test strategy; moreover they should be appropriate to the test level and particular testing techniques. The derived test cases should include the expected results for testing.

- **Test Environment Development**: The test environment includes test equipment, the control of test cases, analysis of results, scripts, and other materials for testing. It needs to be organized or created properly in preparation for future activities.

- **Execution**: In this active test cases are in the test environment. All actions during testing should be documented clearly to guarantee the testing process is reproducible. If other people repeat testing, they should get the same result. This is essential for future defect correction and test result evaluation.

- **Test Result Evaluation**: The test result determines whether the test was successful. It also indicates if performance of the software product conforms to the expected design, without any undesired outcomes.

- **Problem Reporting/Test Log**: The testing log should contain basic information regarding the test, which includes the time of test execution, software environment configuration, the tester who performs the testing, and other related data. It should also include the record of unexpected outcomes during testing and the final result.

- **Defect Tracking**: The defects that are caught during testing should be analysed to find what errors in the software caused those defects.

Test Planning should start at the early stages of a software project. Testing not only targets the code and the final product, but also analyzes the requirements, and the system design. Testing at the design stage can correct unsuitable parts of a project before they are implemented. This may save a lot of time in the future, since it avoids unnecessary implementation and saves the costs of finding bugs. During the code phase, tester and developer must cooperate properly to ensure that the behaviour of each release module strictly conforms to the specifications and is strong enough. Also the inter-operation between modules must execute correctly. In the software development process, the test phase sometimes is the most expensive part. The costs are mainly for:

- the design of the test plan and test cases. This must be done very carefully since the correctness of the test plan and the quality of test cases are directly affect the results of testing;

- the execution of test cases. This also needs a considerable amount of time, especially for big systems;

- the test result evaluation. This needs not only time but also sufficiently experienced of testers; and
- bug correction. This is costly because it requires not only the correction but also needs to ensure that the modification corrects the bugs without causing other problems.

Whatever testing does, the final goal is to find and fix the bugs in the software as early as possible and assure the effectiveness of software systems. Testers always want to do thorough testing, and find and fix all bugs to make the software perfect. Unfortunately this is only a dream. People can not fully test even very simple programs. Usually the set of inputs and outputs of software is very large or even unlimited; the number of possible paths through the software is very large, too. A full test suite would have to be huge enough to cover all cases that would occur during the software execution, but for most software, this test case set is unlimited. By current technology, it is impossible to generate and execute unlimited test cases. So one key mission of testing is finding a way to reduce the huge number of possible test cases, and decide which test cases are important, and which are not. According to the software definition that is presented at the beginning of this section, software testing needs to select a finite set of test cases from the usually infinite executions domain, and use dynamic verification to check the behavior of a program against the expected behaviors.

Since software can not be tested completely, and testers can only use finite test cases to find as many bugs as possible, how can it be done? There are two most classical test strategies that will be explained in the next section.

2.1.1 White-box Testing and Black-box Testing

There are different ways to classify software testing, Considering how to approach the testing for software, testing can be divided into Black-box Testing and White-box Testing. They are also the most popular strategies for testing.

White-box Testing

A software system can be regarded as a box. In White-box Testing (sometimes called clear-box testing) [Bei90b] [Mye04], the tester can see inside the box. Generally testing is done by accessing the program's code to identify all paths through the software. Based on the paths of the software, the tester needs to choose test case inputs that ensure they can exercise paths through the code, and to determine what outputs are acceptable.

There are a lot of coverage standards in White-box Testing. These include:

- *statement coverage*, which requires each statement to be executed at least once;
- *decision coverage*, which requires each decision branch to be executed at least one time;
- *condition coverage*, which requires each condition of each decision in the software to take all possible values;

- *decision/condition coverage*, which needs to satisfy both the requirements of decision coverage condition coverage. In this case the decision/condition coverage needs to execute each decision branch at least once, and all conditions of each decision branch must cover all possible values;
- *condition combination coverage*, which requires in each decision that all combinations of the each condition must execute at least one time; and
- *path coverage*, which requires execution of all possible paths in the software.

White-box Testing strategy has lots of advantages. Firstly, with the knowledge of the internal software structure, it is easier to find out the types of input that can help in testing the application effectively. Also White-box Testing can check whether, within the software, the code executes according to specification, and it can help to optimize the code and removing extra lines to avoid bringing in hidden defects. However there are still some risks to use White-box Testing.

As presented in the previous phase, White-box Testing guarantees all independent paths within a module have been exercised at least once; it exercises all logical decisions on the true and false sides; it executes all loops at their boundaries and within their operational bounds; and exercises internal data structures to ensure their validity.

White-box Testing however, requires a considerable amount of time and effort. Another factor that causes White-box Testing to become expensive is that the number of independent logic paths in software is often huge. To carry out test cases through each path is usually costly. However, even if all paths pass testing, it does not ensure that they are verified, because exhaustive path testing can not verify whether the code conforms to the software specification. Also it is nearly impossible to look into every bit of code, so some paths might be omitted and faults cause by these paths may not be found. Finally exhaustive path testing may not be able to find some bugs that are related to the data. Because of its characteristics, White-box Testing is mainly used in fields that require high reliability, such as war industry software, aerospace and industrial control systems.

Black-box Testing

In Black-box Testing [Mye04] [Bei95], the structure of the software is not shown to the tester, nor is it considered during testing. The tester only knows how the software is supposed to behave, he/she can not look inside the box to see how the software operates. The only thing the tester can do is type in certain inputs, and check if the output complies with the expected output. The tester doesn't know how or why to get the output, they are only concerned with the results of comparison. Nevertheless, in order to implement Black-box Testing, the tester needs to read through the software specification, and clearly understand the expected behavior of the system.

The main methods of Black-box Testing includes: the *equivalence class division method*, the *boundary value analysis method*, the *wrong to speculate method*, the *cause-effect graph method*, and the *comprehensive strategy method*.

Equivalence Class Division [RML05] [Bei95]

Equivalence Class Division is a classical method of Black-box Testing. Equivalence

Classes indicate a subset of an input domain in which each input value has an equal effect on debugging, using a select a representative value from each subset as a test case. This method can effectively reduce the quantity of test cases and reduce the costs of testing. When partitioning the classes, care must be taken to ensure that there are two kinds of equivalence classes: a *valid equivalence class*, which refers to sets that consist of valid and acceptable values of software; and an *invalid equivalence class*, which denotes sets composed of invalid or insignificant values of the system. Usually there will be one or more valid, and at least one invalid equivalence classes.

Boundary Value Analysis Method [Coe08] [Rei97]

The Boundary Value Analysis method takes boundary values from the input domain, and input values that can cause the software to issue boundary output values as test cases. Long-term experience indicates that a high number of faults occur at the boundary values of inputs/outputs, so that test cases aimed at the boundary values can catch more bugs. Values that are equal or slightly greater or smaller than the boundary are suitable to be selected as test cases. The boundary Value Analysis method is often used as complementary to the Equivalence Class Division method.

Wrong to Speculate Method [Mye04]

The Wrong to Speculate method depends on the use of experience and intuition to anticipate possible errors that could occur in the software system, and uses these supposed errors to make the test suite. The basic idea of the Wrong to Speculate method is first to list all situations that might cause errors in the software. The test cases should be designed specifically for each of these situations. The method is very dependent on people's experience and could be an efficient testing method if the test group is well organized, and gathers enough error speculations.

Cause-effect Graph Method [Elm97] [AP97]

The Cause-effect Graph method is a supplement to the Equivalence Class Division method and the Boundary Value Analysis method, for overcoming the weakness of these two methods, which do not analyse combinations of software input conditions, so test cases for more input situations might be omitted. The adoption of the Cause-effect Graph method can help the tester to choose an efficient test suite by some specified steps. The Cause-effect Graph is a directed graph that maps a set of causes to a set of effects. The tester must draw a cause-effect graph according to the software specification, then add the constraints and conditions for the causes and effects to the graph. He/she then transforms the cause-effect graph to a determinant form, and transforms the rows of the determinant form to test cases.

Random Method [Ham94] [KC00] [SB04]

The Random method uses random data as test cases. This data can be generated automatically to reduce the cost of the testing. Also customers will have more confidence if the system passes the testing with sufficient random data. However, this is a high risk method, as it is difficult to determine what constitute an appropriate number of test cases that is sufficient but not too many; it is also not easy to guarantee that the random data generated accurately represents probable actual inputs. Random testing may omit some parts of the system, or test some parts repeatedly.

Black-box Testing is often used to find functional errors, the bugs of interface, data structure, system performance, and the fault of system initialization and termination. However, Black-box Testing is not a substitute for White-box Testing, although it can be used together with White-box Testing as an assistant to discover omitted faults.

The advantage of Black-box Testing is high automatization. The process of testing can be done automatically; for some software the results can even be analysed by Oracle automatically. For large software systems, it is more effective. Because the structure of code is not considered during testing, the tester tests the software from a user's point of view; he/she does not need to have knowledge of implementation, and can be independent from the developer. But Black-box Testing depends highly on the software specification; if the specification is not clear and concise, the test cases can not be well designed. It is also difficult to test directly the specific parts of code that are very complex and have a higher possibility of containing errors. Another disadvantage is that Black-box Testing may leave many program paths untested, since test cases are not generated according to the structure of code. The final problem of Black-box Testing is bug shooting. If the software does not have well-designed exceptions, the causes of a fault are not easy to confirm.

While Black-box and White-box Testing are still in popular use, in order to overcome the disadvantages of these two strategies, people have attempted to combine them, and have invented the "gray-box testing". Gray-box Testing is similar to Black-box Testing, which tests the software from outside, but the internal information of software is not strictly "off limits". The tester has some knowledge of the software's internal structure, and he/she is required to design a limited number of test cases aimed at to the internal structures. The internal information can also helps the tester to choose appropriate test cases from the input domain.

Our methodology is inspired from famous method Category-partition [OB88], this method is based on Equivalence Class Division method in Black-box testing.

2.2 Category-Partition Methodology

Equivalence Class Division method is one of the most popular methods in Black-box testing. Category-partition (CP)[OB88] is a well-known method for applying the Equivalence Class Division.

Category-partition is used to create functional test suites. In this method a test engineer analyzes the system specification, writes a series of formal test specifications, and

then uses a generator tool to produce test descriptions from which test scripts are written. It provides a formal way to partition the software specifications, and a method to generated a test suite from the specification automatically and systematically. The advantages of this method are that the tester can easily modify the test specification when necessary, and can control the complexity and number of tests by annotating the test specifications with constraints. The Category Partition method has seven steps, the terms and the purpose of each step are explained below.

1. Analyze the specifications and identify the *functional units* (for instance, according to design decomposition). The functional units are the subsystems that can be tested independently.

2. Partition the functional specifications of a unit into *categories*: The categories are the environment conditions and parameters that are relevant for testing purposes.

3. Partition the categories into *choices*:[1] these represent the significant values for each category from the tester's viewpoint. Choices are sets of values; that the values in each set have the same influence on the testing result.

4. Determine constraints among choices (either properties or special conditions), this prevents the construction of redundant, meaningless or or even contradictory combinations of choices.

5. Derive the test specification: categories, choices and constraints form a Test Specification, suitable for automatic processing. It is not yet a list of test cases, but contains all the necessary information for substantiating them by unfolding the constraints.

6. Derive and evaluate the test frames: from the test specification, a set of test frames is derived by taking every allowable combination of categories, choices and constraints.

7. Generate the test scripts, i.e. the sequences of executable test cases.

The Category-partition method provides a great method for generating test cases from the system systematically and automatically. Since the test cases are generated by combining the categories, the number of test cases are closely related to the number of categories. With this method, when the system is complex, the user often gets a huge number of test cases. Sometimes there are too many to be executed. To solve this problem, combinational methods are developed. Next section will present one of these methods, Pair-wise Testing, which is used in our methodology for test cases selection.

2.3 Pair-wise Testing

Testing approaches can be divided into two vast groups: stochastic testing (random testing) and combinatorial testing. Stochastic testing does not execute all test cases, but picks some of them randomly, The test case may locate a bug only by luck. Combinatorial testing generates all possible combinations of test data. Usually this set is too huge or even infinite, so that it can not be used to test software.

[1]Note the usage of the same term "choice" both in XML schema syntax (written as <choice>) and in the CP method (written as *choice*). This is purely coincidental.

What the tester needs is a set of finite test data that includes the selected test cases, and is of a suitable size; the method to get this set of test data is "test case selection". Pairwise testing is a well-known and important method for selecting the test cases. It is a widely popular approach to the combinatorial software testing method which for each pair of input parameters to a system, tests all possible discrete combinations of those parameters [THCS01]. As presented before, whether we do Black-box Testing or White-box Testing, the combinations of system parameters are usually very huge even for a simple system. For instance suppose a system have 3 parameters and each of them has 10 values, to do exhaustive testing all the values in the different parameters should be combined, and result in 3^{10} combinations. If the system is complicated, it is impossible to do exhaustive testing because the number of test cases will be too huge to execute. In the history of software testing, people have found that in a software system, most of the errors are caused by the interactions of one or two parameters. Bugs involving interactions between three or more parameters are much less common, and it is very rare that an error is caused by the interaction of all parameters [YL02]. Therefore if we apply pair-wise testing to test the interaction of all pairs of parameters, we should catch most of the bugs at a greatly reduced cost.

2.4 XML Schema Fundamentals

As presented in Chapter 1, currently software is becomeing more and more complicated. Communication among different software components and different systems is becoming more and more frequent. The interoperability of software needs a standard format for data. XML [Har99] is the most popular and general standard for storing and exchanging data. It have been widely accepted for several years. We will focus on systems based on XML.

XML Schema is itself represented in XML, and can be easily extended. It defines the rules of the elements, attributes and other items of XML Documents.

In this section some basic concepts of XML and XML Schema are provided.

2.4.1 Brief Introduction of XML Language

"*Extensible Markup Language (XML) is a simple, very flexible text format derived from SGML (ISO 8879). Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere.*" [XML96]

XML is a set of rules for defining semantic tags that break a document into parts and identify the different parts of the document. It is a meta-markup language that defines a syntax used to define other domain-specific, semantic, structured markup languages [Har99].

XML language is the W3C endorsed standard for document mark-up; in simple terms, it defines a generic syntax used to mark up data with intuitive, human-readable tags. XML

is a cross-platform, text-based and extensible language. As a meta-markup language, there are not any predefined tags in XML language; users make up the tags as required. Figure 2.1 shows a simple example of XML. As shown in the example, an XML Document must have one root element, the content of root element could be text-only or other elements. The XML Document in Figure 2.1 presents the structure and content of an email. All tags in the example are defined by the user. The content of the tags must appear between the start tag that is written as <tag name> and close tag that is written as </tag name>.

```
<?xml version="1.0" encoding="UTF-8"?>
<email>
<to>Annie</to>
<from>Jasemi</from>
<title>Greeting</title>
<body>Hi, how are you? I miss you!</body>
</email>
```

Figure 2.1: A simple XML

XML can store and organize any kind of information in textual form. The information is represented as an hierarchy of elements with names, optional attributes and contents, which can be defined by the developers. XML Documents typically contain a structure that includes text, attributes, and other elements that, in turn, may include elements, text and attributes. This structure can be described as a tree, in which every item in the XML Document holds a position.

XML provides a standard format for storing and exchanging documents between computer applications (and not only), and it brings big advantages to the developers. XML language gives the possibility to the user to design the specific markup languages for different domains of the system, so that the developers can focus on the part they care about, and reduce the complexity of the system description during communication among developers. XML is a self-describing language; it can be written totally in ASCII text that is easily understood by computers, and is also easily read by a person, even if he/she does not have professional knowledge of XML. Since it is easy to read and write and it is non-proprietary, XML is an excellent format to interchange data among different applications. The data of XML Document is well structured, so it is ideal for large and complex documents.

With Unicode as its standard character set, the XML supports many kinds of writing systems and symbols, and it can be edited with any kind of editor, from a standard text editor to a visual development environment. Besides, XML is easily combined with style sheets to create formatted documents in any desired style. Because of its many benefits, XML is fast becoming the most widely used format for data interchange between a system's components, and on the Web.

2.4.2 Document Type Definitions

Besides XML Schema, Document Type Definition (DTD) is an other well-known XML Schema Language. Before XML Schema was developed, it was the most popular schema language. It is not XML based schema language; it specifies the tags used to define legal elements, attributes and constraints. Fig 2.2 shows a simple DTD example.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT people_information (person*)>
<!ELEMENT person (name, birthdate?, address?, IDnumber?)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT birthdate (#PCDATA)>
<!ELEMENT address (#PCDATA)>
<!ELEMENT IDnumber (#PCDATA)>
```

Figure 2.2: A simple example of DTD “People_information”

By the specification of DTD in Figure 2.2, in the conformed XML Document, “people_list” is a valid element, and it can contain any number of elements named “person”, because “person” is followed with *; and “person” is a valid element name, because it contains one element “name”, followed by three optional elements “birthday”, “address” and “IDnumber”. The “?” indicates these elements are optional. All these three optional elements should contain the parsed character data since they are followed with **#PCDATA**. In Figure 2.3 we show an XML Document that conforms to the specification of DTD People_information.

```
<?xml version="1.0" encoding="UTF-8"?>
<people_information>
  <person>
    <name>Sam Brook</name>
    <birthday>15/12/1970</birthday>
    <address>23 Ester Ave NY</address>
  </person>
</people_information>
```

Figure 2.3: An XML Document conforms to DTD “People_information”

In recent years, DTD has becomes very popular schema language in XML applications, but it does not support communication between different modules. Also it has very basic content models that do not have enough capability to give a very clear description of XML Document structure, the element, attribute and constraints of data types. So in May of 2001, W3C recommended XML Schema as the standard mode of XML, and since then XML Schema has started to replace DTD gradually.

2.4.3 XML Schema

An XML Schema language is a formalization of the constraints, expressed as rules or a model of structure, that applies to a class of XML Documents [vdV02]. Like other XML Schema languages (such as DTD), XML Schema expresses the rules that can determine whether an XML Document conforms to these rule. If it conforms, then it can be considered as "valid" to that schema. When an XML Document has the structures or elements that are not defined in the XML Schema, then it does not conform and is invalid against the XML Schema.

When an XML document is validated against a schema (a process known as assessment), the schema to be used for validation can either be supplied as a parameter to the validation engine, or it can be referenced directly from the instance document using two special attributes; xsi:schemaLocation and xsi:noNamespaceSchemaLocation. (The latter mechanism requires the client invoking validation to trust the document sufficiently to know that it is being validated against the correct schema.)

XML Schema offers a powerful set of tools for defining acceptable structures and content of XML Documents. Technically, an XML Schema is a collection of abstract metadata that resides within the XML format.

Schema documents are organized by namespace: all the named schema components belong to a target namespace, and the target namespace is a property of the schema document as a whole. A schema document may include other schema documents with the same namespace, and may import schema documents with a different namespace.

XML Schema is based on the format of XML, and could extend and reuse the elements and types both inside the schema and also in the other namespaces. XML Schema is becoming more widely-used than other schemas because of its advantages:

First, XML Schema uses XML as the language to define the schema, so users who want to use XML Schema do not need to learn a new language. Also, it inherits the properties of XML, which is well structured and easily read both by humans and by computers.

Further, XML Schema defined more than 44 built-in datatypes. and each of these datatypes can be further refined for fine-grained validation of the character data in XML.

The cardinality of the elements can be defined in a fine-grained manner using the minOccurs and maxOccurs attributes. This is a big improvement: DTD can not define the exact number of element occurrence. XML Schema makes the element definition more clear and conformed.

XML Schema also has other new features. It supports modularity and re-usability by extension, restriction, import, include, and redefine constructs. This brings great flexibility in defining elements, and can also save time and avoid repeated definition of elements, types and attributes.

XML Schema supports identity constraints to ensure the uniqueness of a value in an XML Document, in the specified set, and it has an Abstract Data Model and therefore is not bound to XML representation only. While schemas are powerful, that power comes with substantial complexity.

In many ways, XML Schemas act as design tools to XML Documents. There are many uses of XML Schemas in the world of XML, but the most important and common usage is validation.

There are two significants uses of the XML validation. One is to check if the XML Document has the correct syntax conforming to the grammar of the XML specification; this validation is called "XML well formed". Being well-formed is a basic requirement for an XML Document. Another validation is to check if the XML Document conforms to a DTD or an XML Schema. When the XML Document conforms to the criterion a defined in the XML Schema/DTD, then we call this XML Document valid for that schema; otherwise it is invalid. The validation against schemas takes place at several levels: The structural level checks if the XML element and attribute structures conform to the specification of the schema. Data level validation makes certain that the contents of the elements and attributes meet the rules made by the schema. In Section 2.4.5 we will give more details about XML validation against an XML Schema.

Besides validation, XML Schemas also are frequently used to do the documentation of XML. XML Schema provides a formal description of the structure and content of the XML Documents; usually when a new XML vocabulary is published, there will be XML Schemas attached to it. The XML Schema now become a part of the XML Documentation. Because it is easily understood by people and machines, it is helpful in understanding the structure of the XML Documents.

2.4.4 Elements and Constraints of XML Schema

XML Schema is the description of a type of XML Document. It makes the rules for the syntax constraints of these XML Documents, and defines the elements, attributes that can appear in an XML Document, and their data types. Also it specifies the valid structure of the XML Document. This specification includes:

* The parent elements and their child elements;
* The sequence in which the elements should appear;
* The occurrence times of the elements, and the appearance rules for attributes;
* The context of the elements;
* The predefined values for the elements and attributes, specifically the fixed and default values.

For this purpose, XML Schema specifies a set of elements and constraints. We list the elements that can affect and are significant to our work in Table 2.1, and give a simple explanation of each of them. This table is taken from [W3S05]. In Chapter 6 there are more explanations for each of the elements with a simple XML Schema as test case for TAXI tool.

In order to define acceptable values for elements and attributes, XML Schema also defines a set of value restrictions. In XML Schema the restrictions for elements are called Facets. We list the Facets in Table 2.2 [W3S05].

2.4.5 XML Document Validation

As already presented in the previous section, XML validation is the most widely-used XML Schema application; also in our research it is an important concept. As presented before, XML Schema files define the structure of a class of XML Documents. The process for checking whether an XML Document matches the specification of a schema is called XML Validation. This validation is not the same as XML's core concept of syntactic well-formedness. When an XML Document relates to a schema, it is considered "valid" when it is well-formed and conforms to the associated schema.

When XML Documents attach all of the specifications of XML Schema, we call the XML Documents conform to this XML Schema, and they are "valid" documents of this schema. The requirements typically include such constraints as:

Element and attribute, and time they are permitted to occur: XML Schema defines the element and attribute by giving the name, structures, and rules for how they can be included in the XML Document. There are two attributes that restrict the element occurrences in XML Document: "minOccurs" defines the minimum occurrence, and "maxOccurs" defines the maximum occurrence. When the "minOccurs" of an element equals "0", it means this element can be absent from the XML Document; otherwise it must be included in the XML Documents. Also the time that an element occurs must between the values of "minOccurs" and "maxOccurs".

In contrast to elements, attributes can not appear more than once in an element, so in the schema, attributes have an attribute "use" to define the occurrences. This "use" has only three values: *optional*, *required* and *prohibited*. When the value equals *optional*, the attribute is not obliged to appear in XML Document; *Required* means the attribute must occur in XML Documents; while *prohibited* denotes that the attribute is forbidden to appear in XML Document.

The legal structure: The structure of elements and attributes is specified by a regular expression syntax in specification of XML Schema. It defines the acceptable parent, child of each specific element, and their allowable attributes and constraints. All elements in the XML Document must be constructed according the rules, otherwise the element and the whole XML Document are considered to be "invalid".

The data types of the elements and attributes: XML Schema defines the data types for each of its elements and attributes. In the associated XML Documents the element data type must be interpreted strictly according to the XML Schema.

Along with the wide-used of XML Schema, there are many tools that can valid ate an XML Document against a schema automatically. For instance XMLSpy [XML05b] is a very well-known tool, that it offers a set of tools for XML applications; Stylus Studio [Cor08] is a powerful XML integrated development environment; also the Apache Xerces

Java XML Parser library provides a highly standards-conformance validator [Jav06]; it offers very good performance for validating.

2.5 Summary

In this chapter we presented the background for this thesis. We introduced some information about Software Testing, especially Black-box Testing and White-box Testing. After that we included a brief introduction to the Category-partition method, which forms the basis method of our methodology and Pair-wise Testing which is also used in our approach for selecting the test cases. Simple introductions to XML and XML Schema foundations are also presented in this chapter as they are leading actors in our research. With the content of this chapter, it will be easier to follow and understand this thesis.

Element	Explanation
all	Specifies that the child elements can appear in any order. Each child element can occur 0 or 1 time
any	Enables the author to extend the XML Document with elements not specified by the schema
anyAttribute	Enables the author to extend the XML Document with attributes not specified by the schema
attribute	Defines an attribute
attributeGroup	Defines an attribute group to be used in complex type definitions
choice	Allows only one of the elements contained in the <choice> declaration to be present within the containing element
complexContent	Defines extensions or restrictions on a complex type that contains mixed content or elements only
complexType	Defines a complex type element
element	Defines an element
extension	Extends an existing simpleType or complexType element
field	Specifies an XPath expression that specifies the value used to define an identity constraint
group	Defines a group of elements to be used in complex type definitions
import	Adds multiple schemas with different target namespace to a document
include	Adds multiple schemas with the same target namespace to a document
key	Specifies an attribute or element value as a key within the containing element in an instance document
keyref	Specifies that an attribute or element value correspond to those of the specified key or unique element
list	Defines a simple type element as a list of values
redefine	Redefines simple and complex types, groups, and attribute groups from an external schema restriction
restriction	Defines restrictions on a simpleType, simpleContent, or a complexContent
schema	Defines the root element of a schema
selector	Specifies an XPath expression that selects a set of elements for an identity constraint
sequence	Specifies that the child elements must appear in a sequence. Each child element can occur from 0 to any number of times
simpleType	Defines a simple type and specifies the constraints and information about the values of attributes or text-only elements
simpleContent	Contains extensions or restrictions on a text-only complex type or on a simple type as content and contains no elements
union	Defines a simple type as a collection (union) of values from specified simple data types

Table 2.1: XML Schema elements and attributes

Constraint	**Description**
enumeration	Defines a list of acceptable values
fractionDigits	Specifies the maximum number of decimal places allowed. Must be equal to or greater than zero
length	Specifies the exact number of characters or list items allowed. Must be equal to or greater than zero
maxExclusive	Specifies the upper bounds for numeric values (the value must be less than this value)
maxInclusive	Specifies the upper bounds for numeric values (the value must be less than or equal to this value)
maxLength	Specifies the maximum number of characters or list items allowed. Must be equal to or greater than zero
minExclusive	Specifies the lower bounds for numeric values (the value must be greater than this value)
minInclusive	Specifies the minimum number of characters or list items allowed. Must be equal to or greater than zero
pattern	Defines the exact sequence of characters that are acceptable
totalDigits	Specifies the exact number of digits allowed. Must be greater than zero
field	Specifies an XPath expression that specifies the value used to define an identity constraint
whiteSpace	Specifies how white space (line feeds, tabs, spaces, and carriage returns) is handled

Table 2.2: XML Schema restrictions/facets

Chapter 3

Related Work

Software testing is so important that there are many groups and organizations working on it, trying to find methods to reduce costs and improve the veracity of the testing. This chapter covers the research that is related to our work. During the process of our research we studied extensive literature, and conclude that the related works can be grouped into two aspects. One is from the view of XML based testing methods which will be presented in section 3.1. The other aspect is from the view of automatic XML generation methods, which will be presented in section 3.2.

3.1 Related XML Testing

During our research, we not only focused on our methodology, but also studied a wide area of literature about XML Testing. XML Testing can refer to verifying the adequacy of an XML Document with respect to the user's exigencies; verifying the adequacy of an XML Instance with respect to a specific schema (DTD or XML Schema); verifying the wellformedness of a schema structure; or even for defining methodologies for merging or matching diverse XML Schemas.

As presented in Section 2, the DTDs and XML Schemas have largely evolved paired with XML diffusion. DTDs and XML Schemas are used for expressing basic structural rules and complex restrictions of the diverse data and parameters that units/components exchange with each other; both of them can be considered as the structuring schema of XML Documents.

The introduction of XML and XML Schemas paved the way for many tools and techniques devoted to checking the most varied aspects and concerns of the produced documents. In next sections we will provide an overview of existing approaches, distinguishing between approaches that are applied to XML Documents, and approaches that start from the XML Schema. In this vast context of diverse interpretations, we have attempted to classify the various testing approaches and structure those methodologies, and to compare them with our methodology.

3.1.1 XML and XML Schema Document Testing

XML and XML Schema Document testing is to verify if the elements and values of XML or XML Schema documents are conformed to their specification; this testing can also be called well-formedness. Over the years, XML format flexibility and its possibility to be adapted to any kind of situation have increased the possibility of using it in diverse customized domains, as well as for data interchange for web-sites, graphics, remote and real time applications.

With the aim of developing successful applications which can correctly interoperate each other, verifying the correctness and adequacy of XML or Schema data becomes extremely important.

Due to its flexible schema, XML gives its users a certain freedom in writing their specific documents. With the aim of interoperability, the W3C XML Core Working Group[W3C05b] provides a sort of core infrastructure that can be used for verifying the XML Document. This is represented by a set of XML-based guidelines that provide metrics for determining the conformance to the W3C recommendations [W3C05b]

Well-formedness is an essential test and a basic requirement for an XML or Schema file. If this property is not satisfied, the tested file cannot even be classified as an XML or XML Schema. Well-formedness can be easily verified: a simple browser generally provides conformance validation features for thid type kind of validation. However, well-formedness does not guarantee the quality of an XML or Schema file; it just ensures elements and structure in these files conform to the specification of W3C.

Using as a basis this type of testing indications, different sets of test suites have been implemented, for example [W3C05a],[Con03]. Each of them is represented by a test file (including up to thousands of diverse tests) generally associated with a test report, which contains all the background information for verifying a specific aspect of the conformance of the XML Document to the basic recommendations.

Several XML Schema validators for checking the syntax and the structure of the W3C XML Schema document are available. Among them, most widely used are SQC (Schema Quality Checker) [SQC01], XSV (XML Schema Validator) [W3C01], and XML Spy5 [XML05b]. Recently an interesting approach has been proposed by [LM05], which detects semantic errors in XML Schemas by using mutation analysis.

XML and XML Schema document testing are very basic testing for XML. In Chapter 5 we will present our proof-of-concept tool, TAXI. This tool is able to check both the well-formedness of the input XML Schema and also the generated XML Document automatically. When the XML Schema or the XML Document is not well-formed, user will get the warning message.

3.1.2 Conformance Testing

The object under test by conformance testing is usually an XML Document. In order to successfully complete the information exchange and transition between the different systems, XML Documents need to conform the requirements coming from different do-

mains. The requirements can derive from a standard or they can consist of some specific rules defined by the developer's community, which can include the specifications of the input domain or other requirements of the system.

Basically the targets of the compliance test are document verification and validation. Document verification is used to verify that the messages generated by the system conform to the input standard. There are some tools that verify XML Documents automatically. The standard can be in the format created by the user, such as [RTTnd], which developed strategies for automated production of requested XML Documents for posting, to facilitate the testing of web services, and components, to facilitate the validation of the data content of the XML Documents. The widespread diffusion of XML Schema has increased the proliferation of many tools and methodologies for deriving XML Instances, which represent the allowed naming and structure of data for component interaction and for service requests. XML Instances can be generated from DTD as well. Depending on the schema, XML Instances can be manually generated. This could be complicated and a huge amount of work when schemas are complex, so tools for automated XML Instance generation based on DTD or XML Schema appeared. Some of these generate the XML Instance directly, such as [Sun03], [XML99], [XML04] and [TBSK03]. Some tools generate instances in other notations, like java files [EJB03], [Jav03], C++ classes [XOM02], or .NET language such as C♯ and VB.NET [Obj04]. However, most of these generate instances randomly. The disadvantage of random generation tools is that instances cannot cover all possibilities of the schema. In [DK07] a tool that generates instances based on DTDs, users can make their own test driver with a test script wizard, and get an XML-based class instance. There are also integrate a graphic tools such as [Cor08] and [XML05b], which support the multi-functions of XML testing, including well-formed XML /XML Schema and also validate an XML Instance against a schema. Each time they generate one conformed XML Instance from the specific XML Schema.

Document validation is used to check the conformance of the content and structure of documents, and format the documents to the requirements. XML validation means checking the conformance of structure and data in an XML Document against different specifications or protocol models. For automatic verification and validation a tool has been implemented, [XML03], which enables unit testing of XML, and compares a control XML Document to a test document to do the validation; [XML02] and [XML05a], use different methodologies for large, complex systems. Different XML Documents can use different schemas (DTD or XML Schemas) to specify valid (what allowable can do) and invalid (not allowed) documents. In this case an XML Document can be considered valid only if everything in the document conforms to the declarations in schema. In this case usually a schema should include all the elements, attributes and entities that can be used in the document as well. There exist several tools to verify the XML Documents against its DTD or XML Schema. For instance, [BdR04], [xmlndb], [xmlnda] and [easnd].

Validation is an important application of XML Schema for XML-based systems. In our methodology and application, it is also important. At the end of XML Document generation, we must validate and ensure the derived document conforms with the target XML Schema. Also, for testing applications, XML Validation can often be used as an

oracle to analyse the correctness of system output.

3.1.3 Using Derived XML Document or XML Schema

Many programs set their specific requirements on an XML file, and can work only if the files conform to their specifications. Sometimes a vast amount of information must be manipulated and organized, so methods and tools are developed using as test cases generated XML Schemas or common structures from XML Documents and generated XML Instances from common structures or XML Schemas.

Research on how to derive a common structure from XML Instances is going on actively and interesting results have been produced [Lev99], [STH+99], [Wid99], [GW97]. Recent applications of this technology are [ST02], [WYW00], which extract schemas using graphs according to the frequency of element occurrence in XML Documents and [HHS04], in which the authors represent, by using the XML markup, a text type schema definition of the structure in a scientific paper.

In parallel with these approaches, another field of research, also called instance-level matching [RB01], tries to derive a common structure by dynamically analyzing the diverse XML elements and extracting from time to time the proper schema structure. For implementing such analysis, diverse proposals have been adopted, such as rules, neural networks, and machine learning techniques [BM01], [DDL00], [DDH01], [LC94], [LC00], [LCL00].

Currently there are two solutions for using generated XML Documents to do testing, that are widely-used for this dilemma. One is document type definition (DTD); another is XML Schema, which became an official W3C recommendation in May 2001. Establishing a formalized agreement on the format of data exchange supports the application of testing strategies for checking local data structures and the interfaces used by different components. Using a DTD or XSD allows the recipient of an XML Instance to determine whether that instance conforms to the control document by testing the XML Instances for conformance against control documents. DTDs and XSDs have been the first step towards testing for the conformance of content and applications.

The tool [tox05] first needs to write the XML test instruction based on the XML Schema and the expectation of the derived instances, then generate XML based on the test instruction. Recently there is a new strategy that is presented in [dlR07]; it applies the Category-partition method and uses standard Xpath as the query language to classify the XML Schema, and give the construction of constraints for the valid XML Instances.

Compared with similar tools, our approach and method has some advantages. First our tool does not need extra effort for test case generation; it is simple to use and understand, and generates test cases from XML Schema directly. However, tools like Toxgene and XMLGen first need to rewrite XML Schema into generation instructions manually; they require deep understanding of XML Schema, otherwise the transformation may not be totally correct. If the schema is big and complicated, then rewriting becomes hard work and needs a lot of effort. Our method avoid random testing in the methodology, while provide a sufficient quantity of test cases. Also our method and tool support both value

selection and random value generation; this function is omitted by a lot of other tools, such as [dlR07] which does not generate value for elements, and Toxgene which can not generate random values for elements. Finally, using our method, users can obtain different sizes of test cases from various of XML Schemas easily.

3.1.4 Adopting Perturbation Testing

In the area of using commonly adopted testing strategies for guiding the XML based testing, another interesting work is [OX04], which presents a new approach to testing Web services. The authors, taking as a basis the approach in [SCL01] which presents a technique for using mutation analysis to test the semantic correctness for XML-based component interactions, consider the communication infrastructure of web services, typically XML and SOAP, and develop a new approach to testing them based on data perturbation.

3.2 Syntax-based Testing

Among the process of software development, testing is very important and indispensable. It is also the most expensive part of work. Sometimes testing takes up 50 percent of the development costs. In order to reduce spending, people tend to find methods that can apply a test process totally or partially automatically. The most difficult facet of testing is the deriving of test cases, so there are a lot of methods that focus on how to automatically generate test cases, especially methodologies that are independent of the given software systems. That is also what our research aims to do. Partition testing provide the possibility of classifying the specification of systems formally to derive test cases automatically. For instance Category-partition (CP)[OB88] is a classical and very well-known method in that area of automatic software testing. In software literature there are a lot of methodologies applying partition testing to generated test cases automatically. The methodology that is relevant to our research is Syntax-based Testing, or sometimes called Grammar-based Testing. Every input domain of a software system has a syntax. The syntax could be defined in formal or natural language; it also might not be documented or even not specified, but it does exist. Grammars are used to define syntax, exchange formats and other things. Grammars are omnipresent in software development [Läm01]. The grammar can be defined in various ways. Actually, XML Schema can be considered as the grammar of the input domain of XML-based systems. In the literature studied we found many works about grammar based testing. When the grammar is defined in a formal language, it provides the possibility for the automatic test data generation.

For example [JO06] describes a mutation method that, from the given grammar description of a software artifact, uses mutation operators to create alternate versions of artifacts. According to the grammar, these alternate versions can be valid or invalid, and they can be created directly from the grammar or by modifying a ground string. As presented in [Coh06], there are a lot of XML documents that can be used for testing, but they do not conform to the specific schema. A new method of XML generation is presented

in [Coh06]. Since classes of count-constraints and DTDs can identify the satisfiability problem, these classes can be used to generate XML document that can satisfy both the DTD and count-constraints if this document exists. With this method, the documents that satisfy global constraints can be generated, and can be used for testing purpose. Also [Coh08] presents a framework that uses the target DTD and an example of an XML Document (called a GxBE Document) to generate structural XML documents automatically. The GxBE document is an XML Document with embedded-count constraints. These constraints allow the user to control the global properties of the document. So far the derived documents only have structure, without values for the elements. [Kab08] presents a methodology that receives a DTD as an input, relying on its associate elements' attributed grammars to automatically generate web form XML data. Korat [DM03] [AM07] is a tool to enumerate all valid inputs within a certain size systematically, then evaluate the quality of the test suite by some measurements.

These syntax-based Testing methods or tools also do the XML-based testing. They do not take XML Schema, or only XML Schema, but also the syntaxes or grammars for deriving XML data. These grammars could be constraints, attributes, DTDs, and so on. Actually, XML Schema is also a grammar for a specific class of XML Documents. Studying the related Syntax-based Testing methods opened our eyes, and inspired our further research.

3.3 XML Benchmarks

Finally we mention XML Benchmarks, since benchmarks are one of the application domains of our research. The XML Benchmark guarantees the consistency, recoverability, performance and the usability of the XML systems, especially XML database transaction. Along with the increasing use of XML technology, the requirements of the XML Benchmark are extensive in many industry domains, including government, finance, banking, health care, insurance and so on, although to date, a recognized industrial XML database benchmark standard is still not available. However there are many tools implemented for XML Benchmarks that can be found in the literature and on the internet, such as: [XMa], a XML Benchmark tool that generates well-formed, valid and meaningful XML data from DTDs for XML database benchmarks; [Fra05], an XPath benchmark for XMark, which consists of a Functional Test (XPath-FT) and a Performance Test (XPath-PT) for XPath 1.0 language; and [tox05] that uses TSL language to transform XML Schema to a generation instruction. Valid XML data, the number of the XML data and the value of the XML element can be defined in the instruction. [MLL] is an XML version of the OO7 benchmark [MJC93] enriched with relational, document and navigational queries that are specific and critical for XML databases. The XOO7 benchmark meets the four criteria: relevance, portability, scalability and simplicity.

Comparing between TAXI and other XML Benchmark tools, such as XMarch-1 [TB01], XMark [XMa], XPathMark [Fra05], XBench [XBe], XOO7 [SB01] and Toxgene [tox05], we can see the advantages and disadvantages of TAXI.

Firstly, we compare the schema types that the tools support. XMarch-1, XPathMark and X007 support only DTD; XBench supports both DTD and XML Schema; while Toxgene, XMark and TAXI support only XML Schema.

For the format of schemas that these tools support, some of the tools accept only one or several specific files. For example XMark, Xbench, X007 can accept only one fixed XML Schema or DTD. XMarch-1 can accept a set of fixed DTDs. In contrast to those tools, TAXI and Toxgene can accept common XML Schemas, so that they have much more flexibility during XML Benchmarking, and suit to more applications. The different between Toxgene and TAXI for dealing with the schema is that Toxgene needs to write the instruction for each input schema, and generate instances based on that instruction, while TAXI analyze the schema automatically, and does not need user to know exactly the XML Schema before XML data generation.

With regard to the size of the documents that these tools can generate, the ranges are quite different. XMarch-1 generates mainly small documents, range from 2KB to 100KB; the size of documents generated from XPathMark and XMark could range from 10MB to 10GB; and the size range from XBench and X007 are 1KB to 10 GB and 4MB to 1GB; respectively documents generated from TAXI and Toxgene do not have exact size ranges, since they can generate XML data from various of XML Schemas; the range could be must wider than the others. The smallest documents we used to generate from TAXI is 1KB, and the biggest one is 12GB, but it is not the upper boundary.

The requirement of using multiple namespaces in XML Schemas is common in real world applications; it is very common to use the namespace of XML Schemas. The storing, indexing and query processing of namespaces is subject to performance optimization and cannot be ignored.Among the tools that we are comparing here, only TAXI and XPathMark support namespace.

The final aspect for comparison is schema validation, which is important to assure the correctness of the XML data derivation. Of the tools that we mention in this chapter, only Toxgene has a validation process, but it does not validate XML data against XML Schema; the object of validation is the TSL template. This can save the cost of the time, but not provide 100% assurance that the derived XML Documents are conformed to the XML Schema. Also XMarch-1 could be optionally used to do the validation. On the other hand, TAXI validates each XML Document against the XML Schema when they are generated; it is a required process.

Based on this comparison, TAXI is more flexible as a benchmarking tool for XML data generation, and supports various XML Schema inputs. Also it is not necessary to have a lot of knowledge about the input schema; the analysis and the generation are totally automatic. Documents generated from TAXI have sufficient variety and complexity. Other outstanding features which are better than other tools include the supporting of namespace and the automatic validation process for each generated XML data.

3.4 Summary

In this chapter we presented works related to our research from three aspects: existing approaches to XML-based testing, automatic syntax-based testing, finally we list some relevant works of XML Benchmarks. After the study we found there are some strategies or tools does something similar as our research and methodology, but compare with them, our methodology is able to generate more systematic test cases without predefined instructions; it is adaptable for different applications, and it gives more flexible to the user. Our methodology have the advantages in XML Document generation, and in generation control aspects.

Part II

XML-based Partition Testing Methodology and Implementation

Chapter 4

XML-based Partition Testing

This chapter presents our methodology for XML-based Partition Testing (XPT). XPT is able to analyze an XML Schema and generate XML Instances from the schema automatically and systematically. This methodology is presented in two main parts. One is the implementation of instance generation. The other is composed of functions used to optimize the generation to make XPT methodology more efficient and more flexible.

The chapter is organized to follow the process of our research. Firstly in Section 4.1 presents the details of the methodology by mapping from the CP method to XPT. The component for improving XPT method is shown in Section 4.2.

4.1 Main Methodology of XPT

As presented in Chapter 2, today XML Language has established itself as the de facto standard form for specifying and exchanging data and documents between almost any digital or web applications. On the heels of this, the XML Schema [XML98] has become the primary notation for formally describing what constitutes an agreed valid XML Document within an application domain. We are presenting an XML-based Partition Testing (XPT) approach, which is a proposal to leverage the great potential of the XML Schema in describing input data in an open and standard form, to forward push the automation of test data generation.

From a tester's point of view, in fact, the XML Schema expresses the basic rules and constraints on data and parameters that diverse classes of systems and web applications exchange; thus it provides an accurate and formalized representation of the input domain in a format suitable for automated processing, which clearly offers great potential for test automation. The aim of XML-based Partition Testing (XPT) approach is to leverage this potential.

The XPT method draws its inspiration from Category-partition (CP) [OB88]. As presented in Chapter 2, CP method uses the system specifications to do the analysis, and generates test cases from it. The main steps of the CP method can be found also in Chapter 2. XML Schema is very suitable for applying CP method because of its natural

structure. As a XML-based language, XML Schema is well structured; it offers a great convenience during the partitioning of the categories. Today XML Schema is already became a part of XML Documentation. An XML-based system usually will provide a set of XML Schemas to explain the structure of the XML Documents in the system. We have targeted a system that takes XML Documents as input, and uses the XML Schema to define the input documents, which are actually the specifications of the system input domain. The idea of the XPT method is to try to find a finite number of XML Documents that contain all possible valid structures defined by the XML Schema, This means that those XML Documents cover most possibilities of the system input domain, so that we can use those XML Document as test cases to apply Black-box Testing.

As presented in Chapter 2, there are several basic methodologies to do Black-box Testing. The most popular method is *equivalence class division*; this is usually used with the method *boundary value analysis method*. Category-partition [OB88] is a classic method for equivalence class division. Also we have found it is suitable for applying to an XML Schema. According to the properties of the XML Schema and the CP method, XML Schema is partitioned equally into structures by its elements and attributes; each structure can generate at least one XML Document. The CP method gives seven steps for doing the Category-partition. XPT use the similar steps to partition XML Schemas. Figure 4.1 shows the XPT applies CP method to XML Schema, and the map from CP steps to XPT steps.

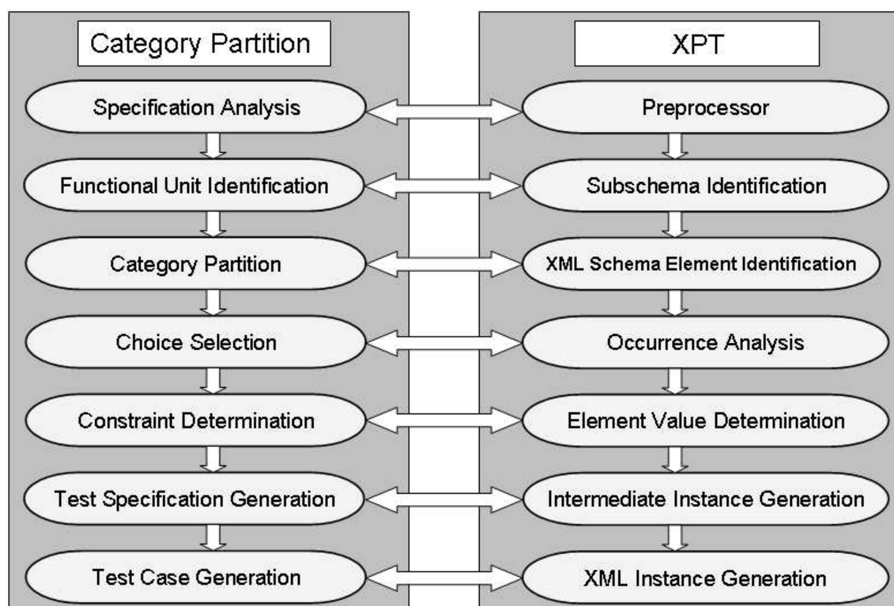

Figure 4.1: The map from Category-partition method to XPT.

Now we will explain Figure 4.1 briefly, step by step. More details will follow in the next sections.

Preprocessor

The first step of CP method is Specification Analysis, which analyzes the specifications of the system. The relevant step in XPT is named Preprocessor, this step is to analyse the structure of the XML Schema, convert the complex schema structures to simple structures and prepare for future activities. The details of the Preprocessor will be presented in Section 4.1.1.

Functional Unit Identification

The next step of CP method is "Functional Unit Identification", which divides the system into a set of *Functional Units*; these *Functional Units* must equal the original system. A *Functional unit* is a subsystem that can be tested independently. The *Functional Units* mapped to XML Schema are the subschemas that can be sub-divided within the original schema. The whole set of subschemas should be equivalent to the mother schema. This step in XPT method is called "subschema identification", which will be explained in Section 4.1.2.

Category partition

The step "Category Partition" of the CP method corresponds to "element identification" of the XPT method. In the CP method this step partitions the *Categories* that have the significant characteristics of parameter and environment that are relevant for testing purposes for each *Functional Unit*. Because of the properties of the XML Schema structure, the elements, attributes and occurrence constraints are considered as *Categories*, because they are the main properties of the schema. Actually, in XPT this step is not required to do anything since the elements, attributes and occurrence constraints are naturally structured.

Choice Identification

After Category Partition, the next step of the CP method is partitioning the *Categories* into *Choices*. *Choices* are the sets of values that have the same influence on the testing result. The *Categories* of an XML Schema are the elements, attributes and occurrence constraints; obviously the *Choices* are their value sets. The *Choices* can be divided into the types and constraints of the elements and attribute, and "minOccurs" and "maxOccurs" attributes. This will be presented in Section 4.1.4.

Constraints Determination

The following step of the CP method is "constraints determination", which gives constraints to each *Choice*, and generates the test cases by combining the *Choices* of each category. In like manner, in XPT method there are only two constraints for *Choices*: "valid" denotes that the *choice* contains values that satisfy the rule of the XML Schema; and "invalid" indicates that the *choice* contains invalid values for the XML Schema.

Test Specification Generation and Test Case Generation

In contrast to the CP method, during the test specification generation, XPT does not combine all values in the *Choices*. In order to reduce the number of final instances, we only combine the values of occurrence constraints; the values of the elements and attributes will be selected randomly during test cast generation. This may lose some combinations of *Choice* values, but we have to give them up to select the most representative test cases, otherwise the number of test cases could be so large that most computers would not have enough capacity to handle it; also it would take too long time to generate and execute the testing. The details of test specification generation and test case generation will be

presented in Section 4.1.6 and Section 4.1.7. Figure 4.2 shows the work process of XPT method. XPT takes an XML Schema, analyzes it and generates a set of XML Documents that are conformed to the schema. The frames at the right side of the figure with gears represent the main steps in XPT.

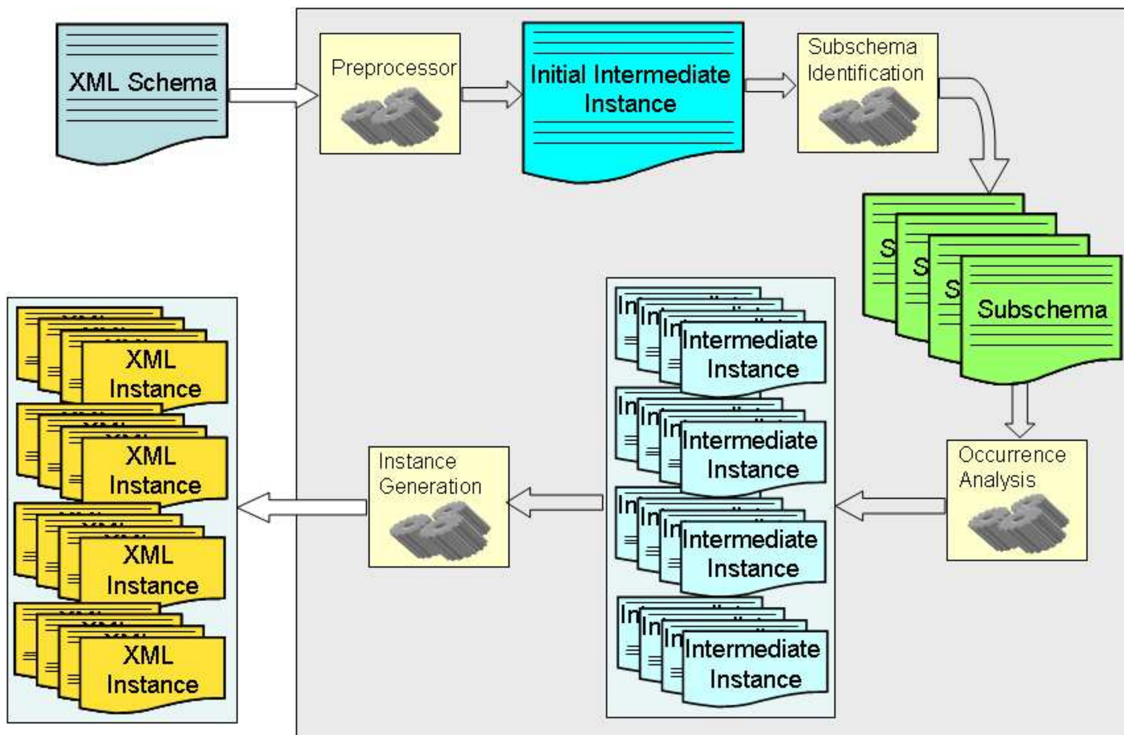

Figure 4.2: The work process of XPT method.

4.1.1 XML Schema Analysis and Rewriting

As described in Chapter 2, an XML Schema file may include not only simple elements but also complex elements. As well both the elements and the definition of the types can be referred from other elements. This property sometimes causes the XML Schema file to have a very complex structure, especially when there are many layers reference. The referred elements bring confusion to the analysis, because using only the type of the element, it is impossible to know its really structure if the type is not a basic XML Schema type, but refers to another user-defined type.

So the first step of XPT is to simplify the structure of the XML Schema file, extend and rewrite the reference structure of the XML Schema, and change the XML Schema to an XML that has the same semantics but a different syntax. This step is corresponds to the step *Specification Analysis* in CP method.

The element in the XML Schema will be transferred into the corresponding XML element structure, and all elements will tend to rewrite to simple elements or <sequence>

elements. The only exceptions are the <choice> element and the <all> element, which are kept as before, only the ID number for each child of the <choice> element is added by Preprocessor. The methods of rewriting are different depending on the types of elements. XPT classifies the elements of XML Schema into *Basic Element*, which include simple elements, <choice>, <all> and <sequence> elements; and referred elements in which the type or the element itself is referred to other type definition or element. The task of schema rewriting is to rewrite the XML Schema format to the corresponding XML format, and transfer the referred elements to basic elements.

Rewriting of Basic Elements

There are two kinds of *Basic Elements* in XPT. One is a simple element, in which the only structure that can conform to the element is an element with a specific type of value. In the different XML Documents the element value could be different, but the structure is always the same. Another one is the basic complex types, include <sequence>, <choice> and <all>. <sequence> specifies an element with children that have a fixed sequence of appearance in an XML Document. If a <sequence> element is without any attributes, then the accepted XML structure is actually unique. Although <choice> and <all> define more than one possibilities of XML Document structures, they are also the basic indicators of XML Schema, therefore their structures will not be solved in this phase.

A *Basic Element* in the schema will change only from an XML Schema structure to an XML structure. To describe it clearly, in Figure 4.3 we show a very simple example. A simple XML Schema element is shown in Figure 4.3; it is an element named "Number", and the type of the element is "xs:integer". After the Preprocessor the XML Schema element is converted to a XML element, but it still has the same name and type as the original one.

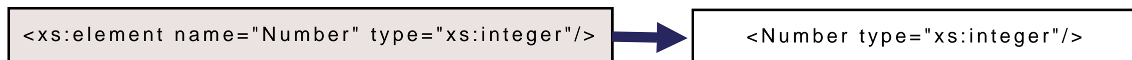

Figure 4.3: The transformation of *Basic Element* by *Preprocessor*.

Rewriting of Referred Elements

Referred Elements need to be rewritten; otherwise they will cause confusion in the future processes as presented before. XPT needs to partition the XML Schema by the elements, attributes, and element occurrences. For *Referred Elements*, it is difficult to understand the whole structure without unfolding the referred parts. But the elements included the referred part will be omitted. The way to resolve this to find the referred part in the XML Schema, and rewrite the referred element by replacing the referenced part with its real content. Figure 4.4 gives a sketch map of this activity.

The rewritten *Referred Elements* need to guarantee equivalence between the original and new structures. After rescription, the XML Schema file is converted into a

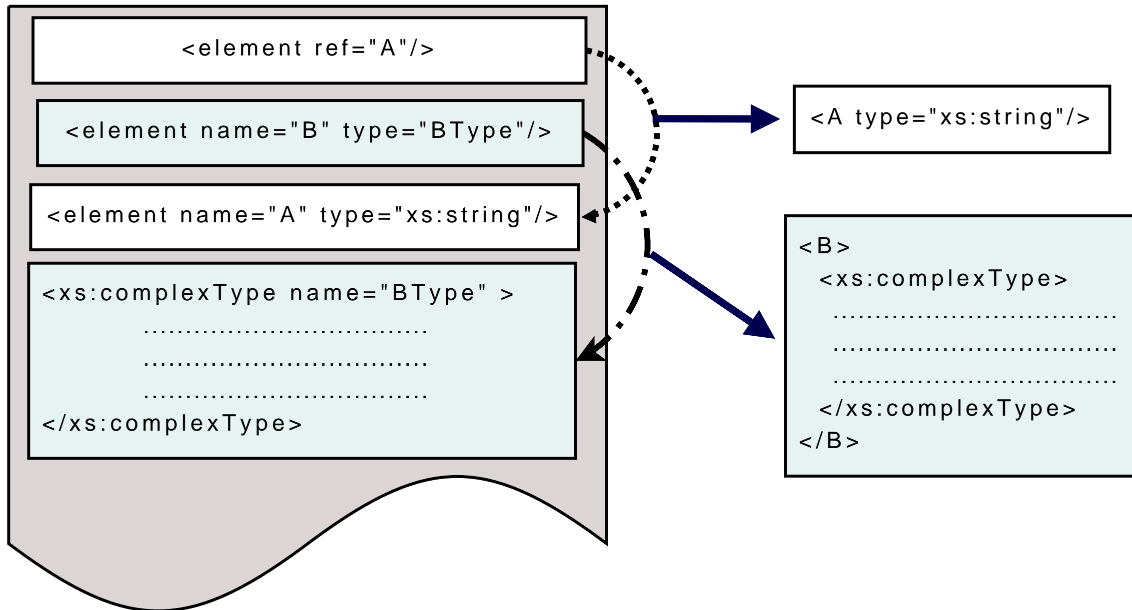

Figure 4.4: The sketch map of referred element rewriting.

XML structure, and the syntax is changed, but the semantic remains the same as original schema. This XML structure is called an Initial Intermediate Instance.

4.1.2 Subschema Derivation

The second step of XPT is Sub-schema Identification (or Choice Analysis); this corresponding to the second step of the CP method Functional Units Identification. In this step the XML Schema is divided equally into a set of subschemas. In the CP method, subschemas can be considered as functional units, which are sub-specifications that can be tested independently.

In XML Schema the most suitable element for deriving the independent sub-schemas is <choice>. In the specification of XML Schema, <choice> defines an element for which in the conformed XML Document, only one child element of <choice> can appear. This means that in the XML Document, these child elements are not closely related; that may even exclude each other. Suppose an XML Schema has an <choice> element with three children; let's call the <choice> element "elementA" . The structures that can be derived are equal to three schemas, each of which has a distinct child element of "elementA". If we divide the XML Schema into subschemas by the child elements of <choice>, each subschema is independent and the set of them has the a structure equivalent to the original schema. We use a simple diagram in Figure 4.5 to show the derivation.

Thus the XML Schema is partitioned into distinct sets. Consequently, the functional units can be derived by combining the children of the various <choice> elements. As

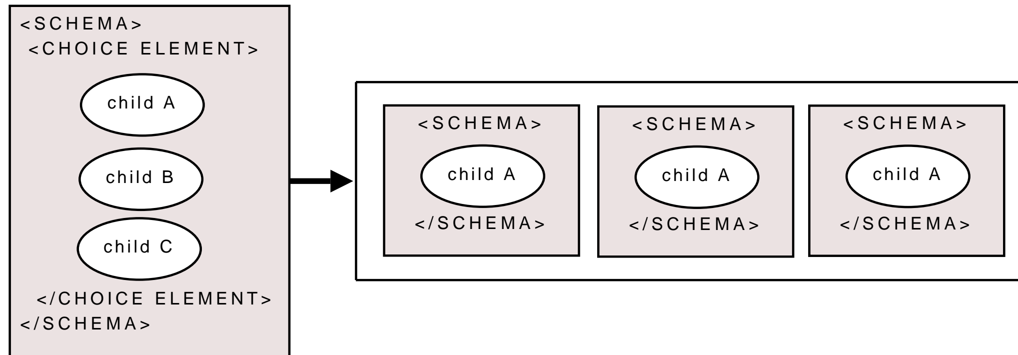

Figure 4.5: Subschema derivation.

a result, each functional unit is a subschema containing, for each <choice>, only one child, (chosen from those representing the family of <choice>) and is different from the other functional units for the chosen combination of <choice> children. We use the method based on Depth-First Search algorithm [THCS01] called DFS-XPT, shown in the following example, to do the combination. The derivation is based on the "initial intermediate XML Instance" that includes only the *Basic Elements*; here we call it briefly as "IIX-Tree".

Let C is the set of child elements of <choice> n; V is the tree for subschema.
$c \in C$; v, v',v" $\in$ V; i,j are the number of child elements.

```
DFS-XPT-Visit(c, v)
1.  c_i ∈ C i ∈ [1, number of child[n]]
2.      v' ⟵ Copy(v)
3.      Add(c_i, v')
4.      if hasChild(c_i) = TRUE
5.          then for each s ∈ Child[c_i]
6.              Add(s, v')
7.              if type[s] = CHOICE
8.                  c'_j ∈ Child[s] j∈[1, number of Child[s]]
9.                      v'' ⟵ Copy[v']
10.                   Add(c'_j, v'')
11.                   DFS-XPT-Visit(c'_j, v'')
12.                   Add(c'_0, v')
13.                   DFS-XPT-Visit(c'_0, v')
14.              else
15.                   for each c' ∈ Child[s]
16.                       Add(c', v')
17.                       DFS-XPT-Visit(c', v')
18.  Add (c_0, v)
19.  DFS-XPT-Visit(c_0, v)
```

G is the set of nodes in IIX-Tree; s is the root element of IIX-tree.

```
DFS-XPT (G)
1.  if ∃(s∈ G & type[s]= CHOICE)
2.  New (v) ∈ V
3.  ∀n ∈ Child[s]
4.      if type[n] ≠ CHOICE
5.          Add(s, v)
6.      else
7.          Add(s, v)
8.          ∀ c ∈ Child[s]
9.              DFS-XPT-Visit(s, v)
```

This subschema generation is based on the "initial intermediate XML Instance" (IIX); the algorithm will be triggered if there is a <choice> node in the IIX-tree. In the DFS-XPT algorithm, G is the node set of IIX-tree. s is the root element of IIX-tree; usually s is the "schema" element. V is the set of trees for derived subschemas.

The first thing the algorithm does check if the <choice> element exists in the IIX-tree. If the result is true, and it includes a choice element, the algorithm then creates a new subschema tree, and does a depth-first search of the tree. For each child node the root element, if child is not a <choice> node, the child node is to added subschema, and the DFS-XPT-Visit. Otherwise it copies the subschema tree, Then this subschema tree is considered as the basic tree, and other subschema trees need to copy and generate based on it. For each child element of <choice> the child nodes are added into different subschema tree. We leave the first node, and do DFS to it at the end, because we would like this node to use the basic tree. Finally we get a set of subtrees, each subtree representing a subschema, and including different child element of <choice>. Let's define the original tree as T, and the subtrees as s. i is the number of subtrees. The relationship of those subschema and the original schema is: $\sum s_0,..., s_i \equiv T$

4.1.3 Element Identification

Category-partition in the CP method is defined as: "To identify the environment conditions and the parameters that are relevant for testing purposes" [OB88]. The Category is:"A sub set of parameter values that determines a particular behavior or output, and whose values are not included in any other category" [OB88]. The properties of an XML Schema gives natural categories, which are the types of XML elements that are defined in the specification of the XML Schema. These types define the structures, contents and the value of the conforming elements in the XML Instances. Each element defines its own types and range of values, and even if the elements have the same type, they are defined independently.

4.1.4 Element Value Determination

A choice in Category-partition is a specific test value for a category (The choice in the CP method is not the same as the <choice> element). The values in a choice have the same influence on the system behavior. The *choice* concept in CP method is similar to

the *equivalence class*, that all values in a choice test the same thing in the system; they can catch the same bug, and also there may be bugs that no values in a choice can not find. In an XML Schema there are two possibilities for identifying a choice: data that can influence the value of element and data that affect the structure of the final instances. The former are derived by analyzing the element attributes (such as *type*, *fixed* value and *default* value) and their restrictions (such as minInclusive, *minLength* and so on). The latter are derived by analyzing the *minOccurs*, and *maxOccurs*. Occurrence is an important attribute that specifies the number of times an element can occur in the parent element. *minOccurs* specifies the minimum occurrence, and *maxOccurs* specifies the maximum occurrence, these attribute values are the boundaries of the choice. The values between the boundaries and outside the range are considered as different choices. Similarly, for the element value, if there is a value restriction, the choice is divided by the value of the restriction; otherwise the choice is divided by the restriction of element data type.

The Category and the Choices of the XML Schema defined by XPT are presented in Figure 4.6

Category	Choice
Type of the element	**Occurrence(minOccurs, maxOccurs)**
	Fix value (fix, default)
	Constraints of the element values (minInclusive, maxInclusive,minExclusive, maxExclusive, length,minLength, maxLength, totalDigits)

Figure 4.6: Category and Choices in XML Schema.

4.1.5 Constraint Determination

In XPT there are only two types of *constraint*: "valid" and "invalid". They are associated with the definition of a set of valid or invalid instance respectively. Thus a "valid" constraint indicates that the values in this *choice* conforms to the specification of the XML Schema, while an "invalid" constraint denotes that the values in the *choice* does not conform to the declaration of XML Schema.

Figure 4.7 shows the constraints with in the category and choices of XML Schema.

4.1.6 Intermediate Instance Generation

The test specification in XPT is called *Intermediate Instance*, which is generated by combing the values from each *choice* occurrence. With respect to the CP method, we added a further refinement to the test specification generation: the application of the Boundary Condition [Pat05] [CK99] approach to the occurrences of each element. Based on our experiences, software is susceptible to bugs at the boundaries; if we choose boundary data it is easy to find the bugs in a system.

Category	Choice	Constraints
Types of the element Define by XML Schema	**Occurrence:** maxOccurs minOccurs	**Valid:** minOccurs<=value<=maxOccurs; **Invalid:** value<minOccurs; value>maxOccurs
	Fix value: fix, default	**Valid:** value = fix; defalutValue=default; **Invalid:** value!=fix; defalutValue!=default;

	Constraints of the element values:
	minInclusive
	maxInclusive
	minExclusive
	maxExclusive
	length
	minLength
	maxLength
	totalDigits
		Valid: minInclusive<=value<=maxInclusive;
		minExclusive<value<maxExclusive;
		valueLength = length;
		valueLength>=minLength;
		valueLength<=maxLength;
		numberOfDigits=totalDigits;
		Invalid: value<minInclusive||<=minExclusive;
		value>maxInclusive||>=maxExclusive;
		valueLength != length;
		valueLength <minLength||>maxLength
		numberOfDigits!=totalDigits

Figure 4.7: Categories, choices and constraints of XML Schema.

In XPT, combining all the possible values of *minOccurs* and *maxOccurs* defined for each element is, in fact, unfeasible. The number of Intermediate Instances could be extremely large or even infinite; for example, the *maxOccurs* could be defined as "unbounded". To address this problem we could associated specific boundary values taken from the input domains with the occurrences of each element. In this case, we would take the extreme value for *minOccurs* and *maxOccurs* wherever specified and assign fixed values in the other cases. When the value of *maxOccurs* is defined as "unbounded", it could be assigned an integer M by XPT. Besides the values of *minOccurs* and *maxOccurs*, the middle values could be generated by XPT automatically by calculating and obtaining the integer midway between the boundaries.

For instance, Figure 4.8 gives an XML Schema that has occurrence attributes. From that figure we can see that in an XML Schema file, there is a sequence element "item" with three child elements, and all of these three elements have occurrence attributes.[1] The occurrences of these child elements are:

- The element "itemA" has a *minOccurs* attribute that equals "0", this means "itemA" is an optional element, that could be absent from the sequence element "item". The maximum occurrence of "itemA" is not specified, so by default it is equal to "1". As presented before, to apply boundary conditions, the boundary values of the occurrence and one value between the boundaries will be selected for further test

[1]By default, the *maxOccurs* and *minOccurs* are equal to 1 when the occurrence attribute is not specified for the element.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="items">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="itemA" type="xs:string" minOccurs="0"/>
        <xs:element name="itemB" type="xs:string" maxOccurs="unbounded"/>
        <xs:element name="itemC" type="xs:string" minOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Boundary Values of Occurrence Attributes		
itemA	itemB	itemC
minA(0)	minB(1)	minC(1)
midA(1)	midB(2)	midC(2)
maxA(1)	maxB(3)	maxC(3)

Figure 4.8: Schema “items” and the values of its element occurrences.

case generation. Therefor the values of minimum occurrence “0”, and maximum occurrence “1” are selected. Then we need to choose a value between the minimum occurrence and maximum occurrence (usually we take the value at the middle of the boundaries). In this example, the value between “0” and “1” is “0.5”, but the value of occurrence must be an integer. To solve this problem, XPT always takes the next biggest integer if the middle value is a decimal; therefore the middle value of “itemA” is assigned a value of “1”.

- The element “itemB” has a *maxOccurs* attribute that equals “unbounded”; this means there is no boundary for the occurrence time of “itemB”. Of course, it is impossible to take a value that is unbounded. In this case, XPT will assign a default integer M as the value of maximum occurrence. In the example, let us set M as “3”. The *minOccurs* is “1” by default, since it is not specified in the schema. The middle occurrence is equal to the value between *minOccurs* and *maxOccurs*, which is “2”.
- The element “itemC” has the same occurrence attributes as “itemB”, so the values of the occurrences are same as “itemB”.

In Figure 4.8, under this schema we list the boundary values of element occurrences. In this figure the *maxOccurs* is written as “max”, *minOccurs* as “min” and the middle occurrence is written as “mid”. The letter following the occurrence represents the element that the occurrence attribute belongs to, and the number in brackets is the value of occurrence. For instance, “maxA(1)” means the *maxOccurs* of “itemA” is equal to “1”; and “midB(2)” means the middle occurrence value of “itemB” is equal to “2”.

The set of Intermediate Instances is defined by combining the assigned boundary values. The Intermediate Instances have the same structure as the final instances, but the

element values remain implicit. There are two types of combinations provide by XPT: all-combination and n-wise combination.

All-combination exhausts all possible combinations of the values. For the same schema and the boundary values of occurrences in Figure 4.8, the result of all-combination is shown in Figure 4.9.

N	itemA	itemB	itemC
1	minA(0)	midB(2)	maxC(3)
2	midA(1)	maxB(3)	midC(2)
3	maxA(1)	minB(1)	minC(1)
4	minA(0)	maxB(3)	minC(1)
5	midA(1)	minB(1)	maxC(3)
6	maxA(1)	midB(2)	midC(2)
7	minA(0)	minB(1)	midC(2)
8	midA(1)	midB(2)	minC(1)
9	maxA(1)	maxB(3)	maxC(3)
10	minA(0)	minB(1)	maxC(3)
11	midA(1)	minB(1),	midC(2)
12	maxA(1)	minB(1)	minC(1)
13	minA(0)	midB(2),	midC(2)
14	midA(1)	midB(2)	minC(1)
15	maxA(1)	midB(2)	maxC(3)
16	minA(0)	maxB(3)	minC(1)
17	midA(1)	maxB(3)	maxC(3)
18	maxA(1)	maxB(3)	midC(2)
19	minA(0)	midB(2)	minC(1)
20	midA(1)	minB(1),	minC(1)
21	maxA(1)	maxB(3)	minC(1)
22	minA(0)	minB(1),	midC(2)
23	midA(1)	maxB(3)	midC(2)
24	maxA(1)	midB(2)	midC(2)
25	minA(0)	maxB(3)	maxC(3)
26	midA(1)	midB(2)	maxC(3)
27	maxA(1)	minB(1)	maxC(3)

Figure 4.9: All-combinations of occurrence boundary values for schema “items”.

In this example there are three elements, each of them having three values of occurrence. Using the exhaustive method, we get 27 combinations (see Figure 4.9), which include all combinations of the values from each occurrence. If the number of elements having an occurrence attribute is n, then the number of all occurrence combinations is 3^n. Therefore when there are more occurrence attributes in an XML Schema, the number of combinations is increase geometrically. For instance, if in a schema there are 20 elements that have occurrence attributes, as presented before, each of these attributes will have three boundary values for use in combination. With all-combination, there are 3^{20} =3,486,784,401 possible combinations.

Usually tests do not need billions or even millions of test cases; also, excessive numbers of test cases need more time to derive and execute. To solve this problem, we must find a way to select a limited number of instances. We cannot not simply stop the process of the exhaustive combination when the number of combinations is big enough, or just select the combinations randomly, because random strategies may select test cases that are not relevant to the test purpose, and omit other important ones. Based on experience in testing, most of the bugs in a system are caused by one or two features; it is very rare that a bug is caused by all features together; so if we combine two or three features only, then these combinations should find most of the bugs. Pairwise testing [AWW96], [YL02] is the most well-known and popular method for 2-way combinations. The introduction of pairwise testing can be found in Chapter 2.

For the same schema and values of occurrences shown in Figure 4.8, if we apply pairwise testing, we can get 9 combinations, which are shown in Figure 4.10.

Pairwise testing is triggered when the expected number of test cases is smaller than the the number of test cases that can be generated by the all-combination method. It can

No.	itemA	itemB	itemC
1	minA(0)	midB(2)	minC(1)
2	midA(1)	maxB(3)	minC(1)
3	maxA(1)	minB(1)	minC(1)
4	minA(0)	maxB(3)	midC(2)
5	midA(1)	minB(1)	midC(2)
6	maxA(1)	midB(2)	midC(2)
7	minA(0)	minB(1)	maxC(3)
8	midA(1)	midB(2)	maxC(3)
9	maxA(1)	maxB(3)	maxC(3)

Figure 4.10: Pairwise combinations of occurrence boundary values for schema "items".

reduce the number of combinations, and ensure that each pair of occurrence values is combined. With pairwise testing, a large number of combinations can be cut from the all-combinations. As presented in [DMC97], the number of pairwise combinations supposes that the test model had 13 parameters and each has three values. Corresponding to the XPT, this would be an XML Schema with 13 elements that have occurrence attributes. By the exhaustive method, there are 3^{13}= 1,594,323 possible combinations; with pairwise testing, there are 15 combinations generated that cover all pair combinations of the occurrence values.

The Intermediate Instance is generated from the Initial Intermediate Instance by giving each element an exact number of occurrences. The value of each occurrence is taken from the value combinations, and assigned to the corresponding elements. For instance, with the sample schema in Figure 4.8, we try to generate an Intermediate Instance by the results of all-combination (see the combinations in Figure 4.9). First, we take the first combination (see the first combination in Figure 4.9). According to the values in the first combination, the occurrence value of element "itemA" is equal to "0", because in the first combination, its occurrence value corresponds to "minA(0)", which means the value equals the minimum boundary (which is "0"), so "itemA" will not present in this Intermediate Instance; the occurrence value of element "itemB" is "midB(2)" which means the middle occurrence value of B is equal to "2", therefore, in the Intermediate Instance "itemB" will appear twice; in this combination, the occurrence value of element "itemC" is "maxC(3)", so in Intermediate Instance "itemC" it will occur three times.

4.1.7 Test Case Generation

This is the step for generating the final XML Instances. The instances are derived based on the Intermediate Instances that were presented in the previous section, by applying these occurrence values to the elements, and giving the proper value to each element.

These values could be obtained from the predefined value sets, or by using random val-

ues generated automatically. Each Intermediate Instance will basically generate one XML Instance, unless the required number of instances is larger than the number of Intermediate Instances that can be derived from the XML Schema. If there are no predefined values for the elements, XPT will generate the values randomly according to the types and the constraints of the elements. Otherwise, if there are the predefined element values, those values for different elements should be combined by means of the Category-partition. However in XPT method, as presented in the previous section, even the number of Intermediate Instances is easy to reaches a very large number, the combination of element values could lead to the generation of an unwieldy number of instances. In order to reduce the number of derived XML Instances, XPT will forego element value combination, but select predefined values randomly and assign them to the corresponding elements.

Figure 4.11 shows the XML Instances generated from the sample schema by the combinations of pairwise testing. There are nine instances derived by XPT. To show the difference in value selection, we predefine the element values for “itemA” as “dress, coat, jacket, suit, skirt” and “itemB” as “book, paper, magazine, letter, press”. We have not defined any values for “itemC”, but have just let it use random values generated by XPT.

```
<?xml version="1.0" encoding="UTF-8"?>
<items xmlns:xsi="http://www.w3.org/2001/
   XMLSchema-instance">
   <itemB>book</itemB>
   <itemB>magazine</itemB>
   <itemC>utuLogir</itemC>
 </items>
```
1

```
<?xml version="1.0" encoding="UTF-8"?>
<items xmlns:xsi="http://www.w3.org/2001/
 XMLSchema-instance">
   <itemA>coat</itemA>
   <itemB>paper</itemB>
   <itemB>book</itemB>
   <itemB>letter<itemB>
   <itemC>OlflFIgie</itemC>
</items>
```
2

```
<?xml version="1.0" encoding="UTF-8"?>
<items xmlns:xsi="http://www.w3.org/2001/
   XMLSchema-instance">
   <itemA>skirt</itemA>
   <itemB>letter</itemB>
   <itemC>lefjslsp</itemC>
</items>
```
3

```
<?xml version="1.0" encoding="UTF-8"?>
<items xmlns:xsi="http://www.w3.org/2001/
 XMLSchema-instance">
   <itemB>book</itemB>
   <itemB>book</itemB>
   <itemB>paper</itemB>
   <itemC>l09Ldls</itemC>
   <itemC>7kfdidl3f</itemC>
</items>
```
4

```
<?xml version="1.0" encoding="UTF-8"?>
<items xmlns:xsi="http://www.w3.org/2001/
   XMLSchema-instance">
   <itemA>coat</itemA>
   <itemB>paper</itemB>
   <itemC>opoi5fie</itemC>
   <itemC>mdHy</itemC>
</items>
```
5

```
<?xml version="1.0" encoding="UTF-8"?>
<items xmlns:xsi="http://www.w3.org/2001/
   XMLSchema-instance">
   <itemA>dress</itemA>
   <itemB>paper</itemB>
   <itemB>press</itemB>
   <itemC>uer8flHT</itemC>
   <itemC>yGFue</itemC>
</items>
```
6

```
<?xml version="1.0" encoding="UTF-8"?>
<items xmlns:xsi="http://www.w3.org/2001/
   XMLSchema-instance">
   <itemB>press</itemB>
   <itemC>wfdDFdf</itemC>
   <itemC>8je4sf3</itemC>
   <itemC>h8diW2isf</itemC>
</items>
```
7

```
<?xml version="1.0" encoding="UTF-8"?>
<items xmlns:xsi="http://www.w3.org/2001/
   XMLSchema-instance">
   <itemA>jacket</itemA>
   <itemB>letter</itemB>
   <itemB>press</itemB>
   <itemC>jld</itemC>
   <itemC>utYlti6sfs</itemC>
   <itemC>iueR8d3</itemC>
</items>
```
8

```
<?xml version="1.0" encoding="UTF-8"?>
<items xmlns:xsi="http://www.w3.org/2001/
   XMLSchema-instance">
   <itemA>jacket</itemA>
   <itemB>press</itemB>
   <itemB>press</itemB>
   <itemB>book</itemB>
   <itemC>iurlE7</itemC>
   <itemC>3Nfkgk</itemC>
   <itemC>xdfiLhie</itemC>
</items>
```
9

Figure 4.11: XML Instances derived by XPT.

4.2 XPT Test Strategy Selection

As presented before, the XPT method performs the basic function of XML automatic generation, but the process is not controlled. The derived XML Instances depend totally on the structure of the XML Schema and the process of generation is blind.

The XML Schema represents in a clear way the overall structure of the input domain. Apart from the advantages mentioned in the previous subsection, this is an enormous benefit for test planning. The testing phase is an expensive but essential part of development, which must be well-organized and defined.

Generally, it is not easy to decide on which parts the testing effort should be concentrated and the number of test cases to dedicate to each of them. Wrong decisions could increase the overall cost and the completion time of the testing phase. Considering to the XML Schema representation, it is possible to implement an integrated, practical and automatic strategy, planning a suitable set of instances.

In XPT methodology we use Testing Strategy Selection to control the test case selection. Testing Strategy Selection includes Weight Assignment, which assigns weights to the children of the <choice> elements to denoting the parts that is more significant for testing; and Test Strategies for instructing the methods of test case selection, controlling the quantity of the derived XML Documents.

4.2.1 Weight Assignment

Weight Assignment is applied at the beginning of the XPT. The idea underlying the *Weight Assignment* activity is that children of same <choice> may have not the same importance for instances derivation [2]. There could be options rarely used or others having critical impact on the final instance derivation. Because according to the definition of <choice> element, only one child per time can appear in the set of final instances, from the user's point of view the possibility of selecting those most important would be very attractive. He/She can guide the automatic instance derivation, forcing it to derive more instances including the most critical <choice> options.

An XML Schema doesn't provide the possibility of explicitly declaring the criticality of the diverse options; this information is often implicitly left to the sensibility and expertise of the people deriving the instances. The basic idea is to ask the XML Schema expert or user to make this knowledge explicit. For this we provide them with a systematic strategy in order to use such information for test planning. In particular XPT explicitly requests the user to annotate each child of a <choice> element with a value, belonging to the [0,1] interval, representing its relative "importance" with respect to the other children of the same <choice>. This value, called the *Weight*, must be assigned in such a manner that the sum of the weights associated with all the children of the same <choice> element is equal to 1. The more critical a node, the greater its weight. Several criteria

[2]Of course if the original XML Schema did not include <choice> at this point only one structure is available having 1 as subtree weights.

for assigning importance factors could be adopted. Obviously this aspect in the proposed approach remains highly subjective, but here we are not going to provide a quick solution for how numbers should be assigned. We only suggest expressing in quantitative terms the intuitions and information about the peculiarity and importance of the different options, considering that such weights will correspondingly affect the testing stage.

For instance there is an XML Schema, let's call it "Example_schema", which has a <choice> with three child elements, "childA", "childB", and "childC". By default the weights of these child elements will be assigned by XPT as the same weight "0.333" since the total weight should equal to "1" (here $1/3$ is indivisible number, so XPT use the approximate value). Weight Assignment allows the modification of the element weights. Suppose we change the weight of: "childA" to $w1$, "childB" to $w2$, and "childC" to $w3$, and ensure $w1 + w2 + w3 = 1$. After the Subschema Derivation we get three subschemas; the weight of each subschema is equal to the weight of the child element it includes. The details of the weight recalculation of the subschemas can be found in Chapter 5.

4.2.2 Test Strategies

Following the steps described so far, each set of substructures has been defined and a specific subtree weight assigned to each of them. Now it is necessary to determine a test strategy to adopt for test case derivation. For this we consider four different situations: either a certain number of instance to be derived is fixed, or the percentage of functional coverage is chosen, or the method consider with both the instance number and the functional coverage, or all possible instances are generated. The first is the case in which a fixed number of instances must be derived from a specific XML Schema.

A practical application of these strategies is shown as below:

- **Applying XPT with a fixed number of instances**
  If a number NI of final instances is fixed (this could be in practice a case in which a finite set of test cases must be developed) XPT strategy can be used to develop NI final instances out of the many that could be conceived starting from the original XML Schema. Using the subtree weights associated with each substructure, the number of instances that will be automatically derived from each of them is calculated as NI times the subtree weight.

  Let's use the same example as in the previous section. The XML Schema "Example_schema" has a <choice> element with three child elements: "childA" with weight $w1$, "childB" weighted as $w2$, and "childC" weighted as $w3$. So there are three subschemas generated from the "Example_schema"; we call them "subschemaA", "subschemaB" and "subschemaC" according to the different <choice> child elements they include. By means of this test strategy the required numbers of instances from the subschemas are:
  the instance number from subschemaA = $NI * w1$;
  the instance number from subschemaB = $NI * w2$; and
  the instance number from subschemaC = $NI * w3$.

- **Applying XPT with a fixed functional coverage**
  As presented before, the weights of the <choice> children represent their importance for testing, the heavier ones should be paid more attention. Here the coverage means the total weight that is relevant for testing. Of course, the sum of the weight, or here we call it the test coverage is "100%", but this test strategy provides the possibility that the coverage could be less. If a certain percentage of functional test coverage C is established as an exit criterion for testing, and it is less than "100%". XPT will start the subschema selection by ordering their weights in a decreasing manner, multiplying them times 100, and starting the selection from the heaviest ones. The weight of the selected subschemas are added together until the sum of the weights is greater than or equal to C. Then XML Instances are generated only from the selected subschemas. By this test strategy we can specifically focus on certain functions of the testing.

  Using the same example used in the description of the first test strategy, suppose $w1 > w3 > w2$, and $w1 + w3 > C$, then in this case, only "subschemaA" and "subschemaC" are selected and used for generating the XML Instances, and "subschemaB" is ignored.

- **Applying XPT with a fixed functional coverage and number of instances**
  In this case the above mentioned strategies are combined. XPT first selects the proper substructures useful for reaching a certain percentage of functional coverage (as described above). Then XPT considers the subtree weights of these selected subschemas and normalizes them so that their sum is still equal to 1. The newly derived subtree weights are finally used for distributing among the selected substructure, the fixed number of instances to be automatically derived.

  Still using the same example that has already been used in the previous strategies, if the required number is NI, and the test coverage is C, $w1 > w3 > w2$, and $w1 + w3 > C$, therefore "subschemaA" and "subschemaC" are selected. Then XPT redistribute the weight $w1$ and $w3$ to $w1'$ and $w3'$, which $w1' + w3' = 1$. It then distributes the number of instances to the selected subschemas as: the instance number from subschemaA = $NI * w1'$; and the instance number from subschemaC = $NI * w3'$.

- **Applying XPT with all possible combinations**
  By this test strategy XPT will generate all possible XML Instances. In other words, XPT will apply all-combination of the occurrence values to all subschemas. For an XML Schema, if the number of derived subschemas is I, and the number of elements that have the occurrence attribute in the subschemas is $OV_i (i \in (0, I])$, then the number of all possible combinations is: $\Sigma_{i \in [1,I]} 3^{OV_i}$

4.3 Summary

In this chapter, we have described the methodology of XML-based Partition Testing, which is a partition testing methodology of XML Schema.

In additional, we have improved the basic methodology by using a set of test strategies that gives the user the possibility of guiding the process of generation, and distributing the number of instances among the subschemas. This methodology was inspired by the Category-partition method, and improved to fit the characters of the XML Schema.

Chapter 5

TAXI - The Implementation Of XPT

In order to prove the XPT methodology, we have implemented a proof-of-concept tool called TAXI (Testing by Automatically generated XML Instances). TAXI is a java-based tool with a graphic user interface, with this tool a user can easily apply XML Instance generation, this tool also allows a user to configure and control the process of generation.

This chapter presents the details of implementation and the improved architecture adopted to increase the performance of the tool. Section 5.1 gives an overview of the TAXI tool, and Section 5.4 presents the implementation of the TAXI tool.

5.1 The Overview Of TAXI

TAXI is programmed by Java, and implemented according to the methodology of XPT; additionally, by means of the requirements of the application for real projects, TAXI provides functions that allow the user to focus on specific fragments of a schema, and generate more instances from the critical parts of the schema. Figure 5.1 shows the use case diagram of TAXI.

- **Input an XML Schema**: To select and input an XML Schema to TAXI, the schema could be a new one from a local folders, or a schema that has already been used for generating instances, because TAXI records the used XML Schemas. There are two places that perform the schema upload, from the buttons on the tool bar or from the icons on the information bar. Figure 5.4 shows the main interface of TAXI.
- **View the XML Schema**: As shown in Figure 5.4, in the main window of TAXI, an XML Schema file is shown with its tree structure; each element is presented as a node on the tree. The user can view the structure of the schema tree from the main interface of TAXI. The children of the nodes can be extended or retracted, so that the user can focus on the most important part of the schema.
- **Assign weights for the children of <choice>**: As presented in Chapter 4, weight is defined by XPT; it refers to the child elements of <choice>. If there is a <choice>

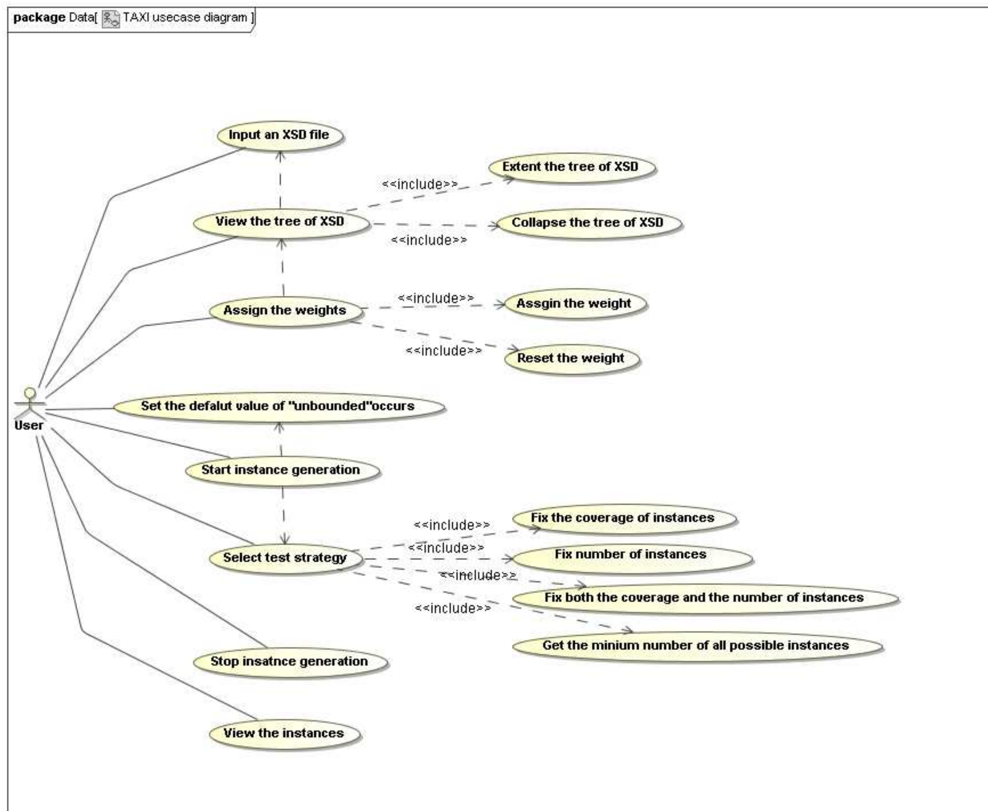

Figure 5.1: TAXI Use case diagram

element, initially the default (uniform) weight values for the corresponding elements are given by TAXI automatically. The user can modify the value of weights using the TAXI interface.

The value of the weight is shown at the right side of the TAXI main interface, as in Figure 5.4; the values of weight can be found under the "weight value" line.

- **Set the value of element occurrence**: In XML Schema, the occurrence of elements is defined by "minOccurs" and "maxOccurs". The value of "maxOccurs" could be "unbounded" when it does not have an exact boundary value. This does not make sense for instance generation, because we can not generate a schema with unlimited number of elements. Therefore, in this case we must give a number of our own. By default TAXI assigns an integer N as the value of unbounded "maxOccurs", for instance "3". However this is not fixed; the user can modify it before instance generation.

- **Select test strategy**: As described in Chapter 4, there are four test strategies that can be selected by the user. The window of test strategy selection is shown in Figure 5.22. The test strategy can control the number of instances, and distribute

the number to subschemas. Each test strategy can guide the different processes of test case derivation, so that the result instance sets are different.

- **Terminate the generation process**: The process of generation can be shut down even if it is not finished. The user can control the process of instance generation, to break it as needed for his/her purposes.
- **View XML Instances**: TAXI has a simple XML editor; the user can use it to view the generated XML Instances. Currently it does not support XML file modification.

5.2 TAXI Components

In this section we will present details of the TAXI tool, the structure and its implementation. The tool is structured by four components:

XML Schema analyzer (XSA) implements the XPT method. It is the core component of TAXI. All other components sever for XSA. It expands and preprocesses the XML Schema, prepares the Intermediate Instance frames and provides a set of final instances. The details of XSA implementation will be presented in Section 5.4.

Test Strategy Selector (TSS) implements a set of test strategies for selecting the final XML Instances. The test strategies use different methods for XML Instance selection, which gives the user the possibility of focusing on and obtain more XML Instances from the critical part of an XML Schema. By each test strategy, the user can get a different set of instances. This component also includes the management of element weight assignment.

Values Storage (VS) manages the values used in the element of final XML Instances. The values include the element value and the boundary value of the element occurrences. During the generation of final instances, the selection of the element value and the assignment of the occurrence value are also carried out by the VS component.

User Interface (UI) is the graphic interface, which handles user interaction, allows the user to load an XML Schema, and applies the activities of instance generation.

The relation between these components is shown in Figure 5.2.

Each component includes the set of activities shown in Figure 5.3. We use different types of frames to distinguish which component these activities belong to. From the figure we can see:

The activities with white background belong to the XSA component; they are: Preprocessor, which is used to rewrite the structure of the XML Schema into an equally simple XML, and prepare for the future generation; Choice Analysis, which divides the XML Schema into a set of subschemas; Occurrence Analysis, which assigns the value of occurrence to the corresponding element; Intermediate Instance Generation, which generates sets of Intermediate Instances from the subschemas; and Final Instance Derivation, which derives XML Instances from the intermediate instances.

The shaded activities belong to the TSS component. There are two activities: Weight Assignment allows the user to modify the weight value of elements; and Test Strategy Selection lets the user choose their preferred strategy for instance generation.

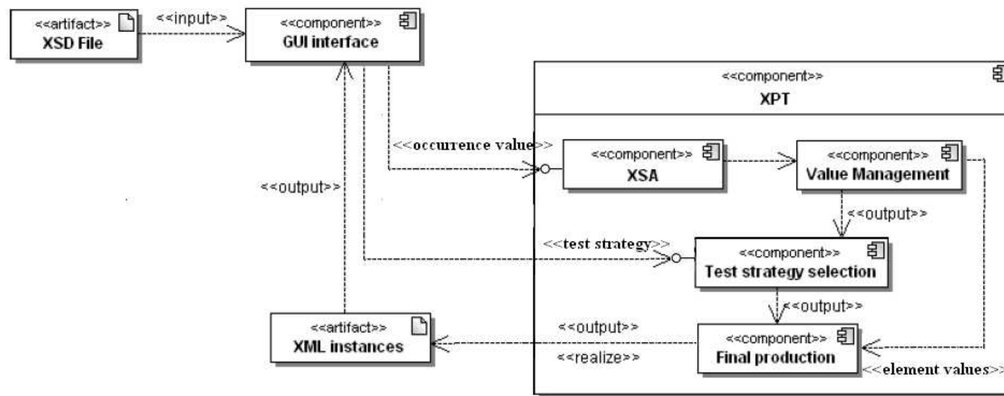

Figure 5.2: Component diagram of TAXI

Those activities with dashed borders facilitate the user interaction and value storage components. Reading Input and Instance Browsing help the user to input the XML Schema and view the derived instances; Database Population is used to populate the value database; and Value Management selects element values during the final instance generation.

5.3 User interface

TAXI has a graphic interface to communicate with the user, Figure 5.4 shows the main interface of TAXI. This interface includes the main window, tool bars, operation buttons and the information bar. The main window lays out the XML Schema file; tool bar and application buttons are used for browsing the schema, assigning the weight to the elements in the schema, and executing the instance generation.

In the activity diagram in Figure 5.3, the component of user interface includes XML Schema Input, Database Population and Instance Browsing. We also present Weight Assignment in this section, because it is a parallel activity with Database Population when the XML Schema is read by the user with the interface.

5.3.1 XML Schema Input and Weight Assignment

There are two ways to input the XML Schema, from a local folder, or by reloading a file that has already been used to generate instances by TAXI. The input schema will first be stored in the XML database automatically. In TAXI we use the open source database eXist [exi]. Next, TAXI will illustrate the XML Schema with a graphic DOM tree, and show it in the main window, so that the user can easily browse the XML Schema, and assign weights to the XML Schema elements. Figure 5.4 shows the main interface of TAXI. The tree of the XML Schema is shown on the left side of the main window. Initially TAXI assigns equal default weights automatically to the <choice> children, but this value can

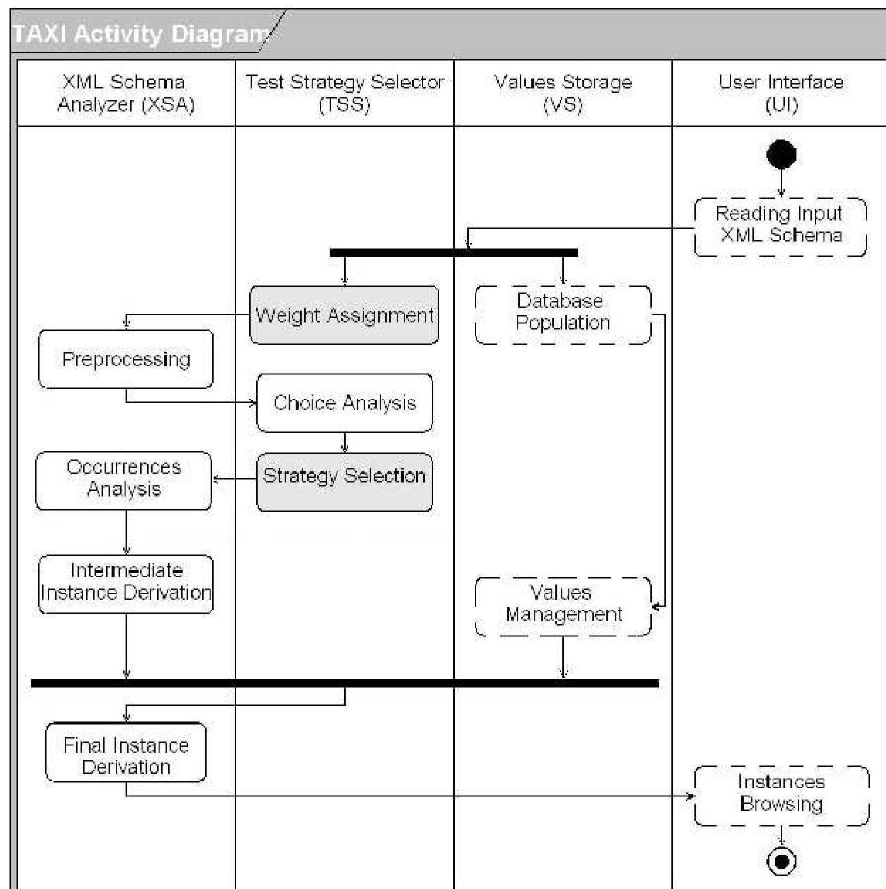

Figure 5.3: TAXI activities

be modified by the user. In Figure 5.4 the default values of the weights are shown on the right side of the <choice> child elements.

To modify the value of weights, the user just clicks the button "Start modifying", changes the weight values, and then clicks the button "Stop modifying" when the modification is finished. For each <choice> element, the total weight of its child elements must equal "1". If the user changes only some of the child elements, when they finish the assignment, TAXI will calculate automatically and make the sum of weights equal to "1". For example, if a <choice> element has three children, then at the beginning they have the same default weight '0.333" (1/3). If the user changes the weights of first two child elements to N and M (N+M $<$ 1), and leaves the last one as before, when he/she clicks the icon "stop modifying", TAXI will automatically assign (1-N-M) as the value of weight to the last child element, so that the sum weight of these three child elements still equal to "1". If the user wants to reset all weights, he/she must ensure the total weight of the new values is "1", otherwise TAXI will send a warning message, and refuse to accept those weight values. With the interface, the user can also see the value of "minOccurs" and "maxOccurs"; they are shown at the right side of the weight values. In the current

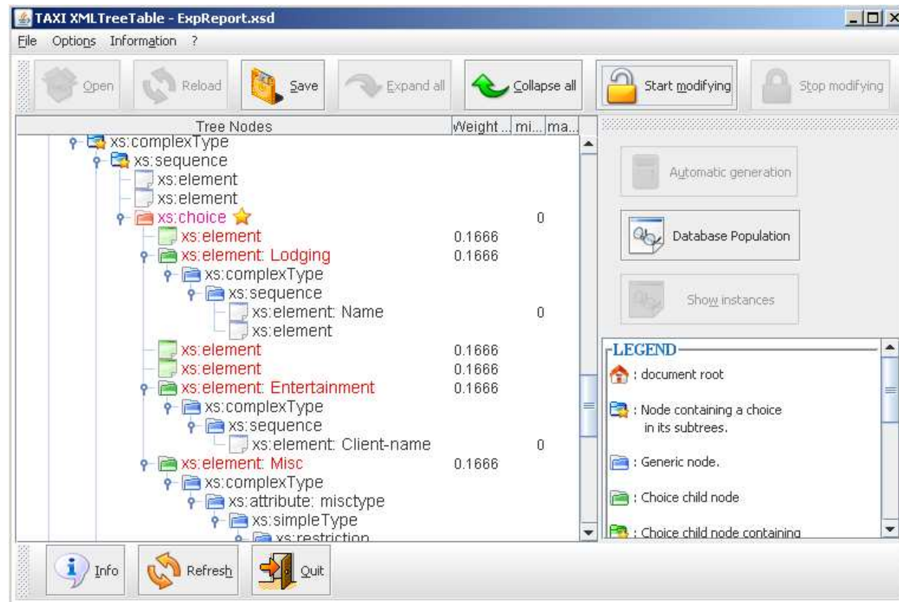

Figure 5.4: TAXI Main interface

version of TAXI, the occurrence of each element can not be changed individually.

5.3.2 Database Population

The user can assign values to the element by Database Population. There are three ways to obtain element values in TAXI: if there is a value definition in the schema, for example an <enumeration> element, values are automatically generated by TAXI in accordance with the definition; from user-defined values, the user must input values for the elements; lastly, random values can be generated by TAXI automatically. For each element, the last two ways can not occur together. By default, TAXI will first check if there is the predefined value for each element. If these exist in the database, these values will be used in the derived XML Instances; otherwise TAXI will generate random values according to the type and constraints of the element automatically, and use these random values in the XML Instance.

Database Population asks the user to insert predefined values of the elements. We use eXist database [exi] to store and manage the predefined values. eXist database is an open source database management system entirely built on XML technology. It stores XML data according to the XML data model and features efficient, index-based XQuery processing [exi], [Mei02]. TAXI performs value collection in the database automatically. For each element the user needs to create a *value file* with a specific format: first, the value file is XML Document; the name of the value file must the same as the related element. For example the value file of element “cat” must be named “cat.xml”; second, in a value file the root element must be the element <value>; the content of <value> is the string list of values, each value separated by a specific symbol, for instance the specific symbol

could be “,”, “;” or other symbol defined by user.

There are three options for Database Population:

Manually Insert Values: the user could create a value file with the same name as the element in the value collection. For example, if we want to set the value of an element “year” in an XML Schema, first we open the Value Collection in the database, and create an XML file named “year.xml”. Then create the element <value> in “year.xml”, and insert the values into the element. For example we use “,” as a separation symbol, and insert the years from 1990 to 2000. Then the content of the value file should look like:
<value>1990,1991,1992,1993,1994,1995,1996,1997,1998,1999,2000</value>

Specify The Source Of values: The user could also specify the source from which the data can be downloaded or the paths to files containing specific values; they could have a remote or local address, but the files must conform to the format presented before. The user must load these files to the value collection of the database.

Collect Schema-defined Values: Additionally, as stated before, TAXI can also collect values from the schema elements. The values could be obtained from the <enumeration> restriction or the “default” and “fixed” attributes of the XML Schema. As is well known, <enumeration> defines a list of acceptable values of an element. TAXI creates the value files for the elements that have the <enumeration> restriction automatically with the values listed inside <enumeration>. For elements with “default” or “fixed” attributes, TAXI does not create a value file, but records the value inside the initial Intermediate Instance, and attaches it to the corresponding element in the remaining instances.

5.3.3 Instance Browsing

There is a simple XML tool for browsing and visualizing generated instances. When the generation is finished, the user can click the button “Show Instances” at the right side of the interface to see the derived instances. Currently this XML tool does not support XML writing; the user can only read, but not modify, the XML Instance.

The following section we presents the component XSA, which is the main implementation of XPT.

5.4 Implementation of XSA

As presented before, TAXI is composed of four components, we will present TAXI implementation according these components.

XSA is the most important component; the XPT method is implemented in XSA. The main classes include:

- *Preprocessor* which analyses and rewrites the XML Schema into a file with simple structures; we call that file *Initial Intermediate Instance*;

- *Choice Solver* derives subschemas from the Initial Intermediate Instance. The subschemas are derived by the combination of <choice> children elements;
- *Occurrence Combination* combines the boundary values of the element occurrence;
- *Intermediate Instance Generation* generates the *Intermediate Instances* from the subschemas by assigning an exact occurrence value to the elements. Elements in the *Intermediate Instance* have a fixed number of occurrences, but they do not have specific element values;
- *Value Management* gets and assigns element values to the elements; the value could be selected from user-defined or schema-defined value sets or generate randomly according to the type and constraints of the elements;
- *Final Instance Generation* derives XML Documents from *Intermediate Instances*. The relationship of these classes is shown in Figure 5.5

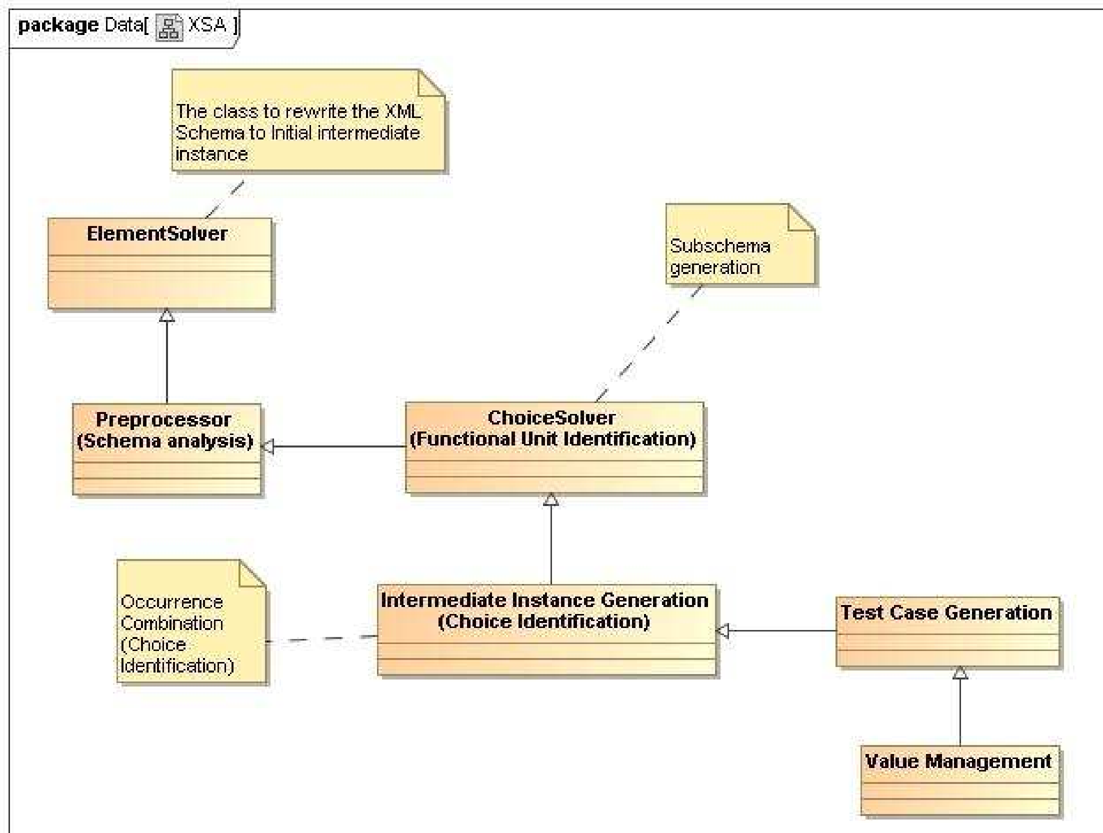

Figure 5.5: Class diagram of TAXI core

In the following sections, we will describe the implementation of each class in the order to the they occur in the XSA component.

5.4.1 Preprocessor

As presented in Chapter 4, we already know Preprocessor is relevant to the first step of CP method Specification Analysis. In XML Schema, an element or type can refers from other element or type definition, to clarify the XML Schema structure, we need to unfold these elements or types to know exactly of their structures. Preprocessor is implemented for this purpose, it changes only the syntax of the XML Schema, and keep its semantics as before. This step transforms the elements in the XML Schema into simple structures. The class diagram of Preprocessor is shown in Figure 5.6.

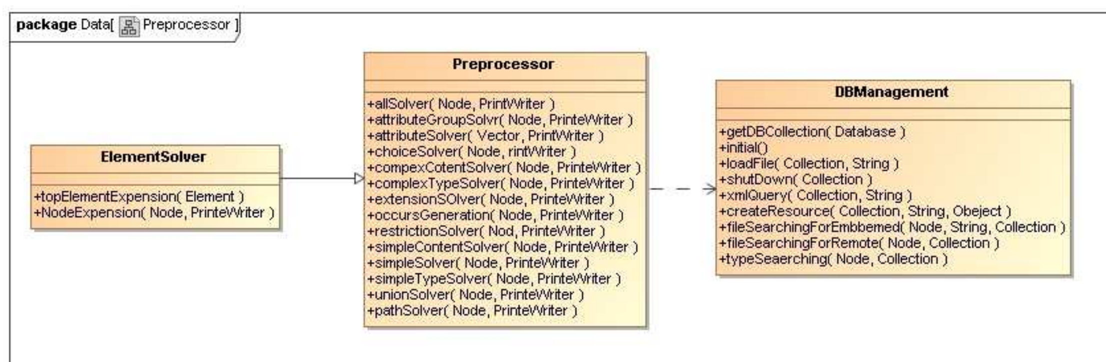

Figure 5.6: Class diagram of Preprocessor

In the class diagram, we can see Preprocessor has different functions for different types of elements in XML Schema. As already presented in Chapter 4, we classify the XML Schema elements into *Basic Element* which includes simple elements and <sequence>; "special element" which includes <all> and <choice>; and "referred element" which includes elements referred from other defined elements, and the type refers from a type definition. There are different methods for dealing with these elements.

For A Multiple Schemas Element

Namespace is one of the core standards of XML; since XML Schema is based on the XML, it also supports the namespace very well. Like any other XML Document, an XML Schema is constructed of elements and attributes, and all these elements and attributes must come from the namespace "http://www.w3.org/2001/XMLSchema". Also, there is a targetNamespace in the schema to specify that the elements, attributes and types defined in an XML Schema all belong to that target namespace. TAXI searches for the namespace that brings the elements and attributes out of the schema. If there is an imported namespace in the XML Schema, TAXI will first get the address provided in the schema, and then try to find the schema of the namespace. Actually, the address defined by the namespace is not always the real address of the XML Schema; therefore if TAXI does not find the schema, it will give a warning message. Otherwise TAXI will download the XML Schema of the imported namespace, and save it into the database with the corresponding namespace prefix.

Besides the imported namespace, the <include> element is also used to add multiple schemas, but it adds schemas with the same target namespace. With the <include> element, the schema could come from the local disc or from the Internet. TAXI first attempts to get the address of the schema from the <schemaLocation> attribute; if it is from the local disc it will be put into the database. Otherwise TAXI will first download the schema, then put it into the database.

For A Basic Element

A *Basic Element* does not need to use the Preprocessor, because in a *Basic Element* there is only one possible structure of the XML Document that can be derived. For theses elements, the only thing that needs to be done in the Preprocessor is rewriting them from XSD format to XML. If it is a <sequence> element, then at the beginning of element, a tag "ISTI_et_CNR_sequence" should be attached to annotate the type of the element, so that in the future process this tag will lead the generation to the correct function. Figure 5.7 shows the XML Schema with a <sequence> element, and the Initial Intermediate Instance after the Preprocessor.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="name">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="firstName" type="xs:string"/>
        <xs:element name="lastName" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
                XML Schema with <seuqnce>
```

```
<?xml version="1.0" encoding="UTF8"?>
<name xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <ISTI_et_CNR_sequence>
    <firstName type="xs:string">
    </firstName>
    <lastName type="xs:string">
    </lastName>
  </ISTI_et_CNR_sequence>
</name>
                Initial Intermediate Instance
```

Figure 5.7: Preprocessor of <sequence>

For A Special Element

As presented before, there are two special elements, <choice> and <all>. They are special because these elements contain more than one possible structure. For example, from element the <all>, the derived XML element could be different from the distinct child orders. To distinguish special elements, there are tags which need to be attached at the beginning of the element. For example, in Figure 5.8, in the schema after the Preprocessor element "petShop" is <all> element, so in the Initial Intermediate Instance, this element is tagged with "ISTI_et_CNR_all".

On the other hand, with a <choice> element, not only does the element need to be tagged with "ISTI_et_CNR_choice", but also the children of the <choice> element must be attached with a tag that includes information on the node's weight and the ID number of the child nodes. The weight tag is written as "ISTI_et_CNR_pesi", which has two attributes; one is named "peso" for recording the value of the weight; another is named "modificato", which marks whether the weight of this node has already been modified

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="petShop">
  <xs:complexType>
    <xs:all>
        <xs:element name="Cat" type="xs:string"/>
        <xs:element name="Dog" type="xs:string"/>
        <xs:element name="Bird" type="xs:string"/>
    </xs:all>
  </xs:complexType>
</xs:element>
</xs:schema>
```

XML Schema with <all>

```
<?xml version="1.0" encoding="UTF-8"?>
<petShop xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <ISTI_et_CNR_all>
    <Cat type="xs:string">
    </Cat>
    <Dog type="xs:string">
    </Dog>
    <Bird type="xs:string">
    </Bird>
  </ISTI_et_CNR_all>
</petShop>
```

Initial Intermediate Instance

Figure 5.8: Preprocessor of `<all>`

by user or is the default value assigned by TAXI. Figure 5.9 shows a simple example of <choice> transformation by Preprocessor. In this figure, we can see that for the preparation of the subschema derivation, the <choice> element and its children are assigned numbers. In the sample schema, the weight does not change, so in the weight tag, the modification attributes are “false” and the values of the weight are the same and set by TAXI automatically.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/xmlSchema">
  <xs:element name="Animal">
    <xs:complexType>
      <xs:choice>
        <xs:element name="Cat" type="xs:string"/>
        <xs:element name="Dog" type="xs:string"/>
        <xs:element name="Bird" type="xs:string"/>
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

XML Schema with <choice>

```
<?xml version="1.0" encoding="UTF8"?>
<Animal xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >
  <ISTI_et_CNR_choice choiceID="0">
    <ISTI_et_CNR_choice_child childID="0">
      <ISTI_et_CNR_pesi modificato="false" peso="0.333"/>
      <Cat type="xs:string"></Cat>
    </ISTI_et_CNR_choice_child>
    <ISTI_et_CNR_choice_child childID="1">
      <ISTI_et_CNR_pesi modificato="false" peso="0.333"/>
      <Dog type="xs:string"></Dog>
    </ISTI_et_CNR_choice_child>
    <ISTI_et_CNR_choice_child childID="2">
      <ISTI_et_CNR_pesi modificato="false" peso="0.333"/>
      <Bird type="xs:string"></Bird>
    </ISTI_et_CNR_choice_child>
  </ISTI_et_CNR_choice>
</Animal>
```

Initial Intermediate Instance

Figure 5.9: Preprocessor of `<choice>`

For A Referred Element

A referred element means that an element or some parts of the element are referred from other definitions of the schema or namespace. In order to make the schema analyze more easily, we want to use this kind of element to fill the referred part with its real content. For referred elements, Preprocessor not only needs to rewrite them from XML Schema format to XML format, but also needs to find the referred part, and fill it into the schema properly. In Table 5.1 we show the referred elements of the XML Schema.

With referred elements, TAXI first checks if the referred element is from the same schema or namespace. If it is, then TAXI gets name and type of the referred element, attribute or type, and searches it from the database; otherwise if the referred element comes from another namespace, then the information for searching should also include

Element	Explanation
attributeGroup	Defines an attribute group to be used in complex type definitions
complexType	Defines a complex type element
simpleType	Defines a simple type and specifies the constraints and information about the values of attributes or text-only elements
simpleContent	Contains extensions or restrictions on a text-only complex type or on a simple type as content and contains no elements
complexContent	Defines extensions or restrictions on a complex type that contains mixed content or elements only
field	Specifies an XPath expression that specifies the value used to define an identity constraint
extension	Extends an existing simpleType or complexType element
group	Defines a group of elements to be used in complex type definitions
redefine	Redefines simple and complex types, groups, and attribute groups from an external schema restriction
selector	Specifies an XPath expression that selects a set of elements for an identity constraint
union	Defines a simple type as a collection (union) of values from specified simple data types

Table 5.1: Referred elements in XML Schema

the prefix of the namespace. If the referred item can not be found, the content will not be filled, and TAXI will give a warning at the end of the Preprocessor.

The rewrite methods for a referred element and a referred type are different. For an element, the result of database searching is an element that has the same name as the target element and is from the required namespace. The element found in the database will replace the original referred element. For example if elementA refers to elementB, then after Preprocessor, elementA is replaced by elementB. A simple example can be found in Figure 5.10.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="personInfo">
  <xs:complexType>
    <xs:sequence>
     <xs:element ref="firstName"/>
     <xs:element ref="lastName"/>
    <xs:element name="gender" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
 </xs:element>
 <xs:element name="firstName" type="xs:string"/>
 <xs:element name="lastName" type="xs:string"/>
</xs:schema>
```

XML Schem with <ref>

```
<?xml version="1.0" encoding="UTF-8"?>
<personInfo xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <ISTI_et_CNR_sequence>
    <firstName type="xs:string">
    </firstName>
    <lastName type="xs:string">
    </lastName>
    <gender type+"xs:string">
    </gender>
  </ISTI_et_CNR_sequence>
</personInfo>
```

Initial Intermediate Instance

Figure 5.10: Preprocessor of `<ref>`

In another case, the XML Schema defines an element, but its type may refer to a type definition. In this case the result of the database search is a type definition, which could be

<complexType> or <simpleType>. Only the content of the type will be used to replace the type of the element; the name of the element remains the same. For example the type of elementA refers to typeA, which is a <sequence> type; after Preprocessor, elementA becomes a <sequence> element that has the same children as typeA, The example is shown in Figure 5.11.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="addressType">
    <xs:sequence>
      <xs:element name="street" type="xs:string"/>
      <xs:element name="city" type="xs:string"/>
      <xs:element name="country" type="xs:string"/>
      <xs:element name="postcode" type="xs:token"/>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="address" type="addressType"/>
</xs:schema>
```

XML Schema with <complexType>

```
<address xmlns:xs="http://www.w3.org/2001/XMLSchema-instance">
  <ISTI_et_CNR_sequence>
    <street type="xs:string"/>
    <city type="xs:string"/>
    <country type="xs:string"/>
    <postcode type="xs:token"/>
  </ISTI_et_CNR_sequence>
</address>
```

Initial Intermediate Instance

Figure 5.11: Preprocessor of `<complexType>`

For Datatype Restrictions/Facets

In addition to the elements, there are also restrictions of the datatype in an XML Schema that need to be rewritten by Preprocessor. Restrictions are used to define acceptable values for XML elements or attributes. Restrictions on XML elements are called facets. The facets in an XML Schema are shown in table 5.2.

In Preprocessor, special tags are defined and attached to the elements in the initial intermediate instance. The tag is written as "ISTI_et_CNR_" + facet name; it records the type and value of the facet. Figure 5.12 shows a simple example of a Preprocessor of datatype facet.

```
<xs:element name="age">
  <xs:simpleType>
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="100"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Element with facet

```
<age type="xs:integer"
  ISTI_et_CNR_minInclusive="0"
  ISTI_et_CNR_maxInclusive="100">
</age>
```

Initial Intermediate Instance

Figure 5.12: Preprocessor of datatype facet

For An Occurrence Attribute

Occurrences of an element also need to be rewritten in Preprocessor. In an XML Schema, element occurrences are defined by "minOccurs" and "maxOccurs". For example, <xs:element

Constrains and restrictions	**Explanation**
enumeration	Defines a list of acceptable values
fractionDigits	Specifies the maximum number of decimal places allowed
length	Specifies the exact number of characters or list items allowed
maxExclusive	bounds for numeric values (the value must be less than this value)
maxInclusive	Specifies the upper bounds for numeric values (the value must be less than or equal to this value)
maxLength	Specifies the maximum number of characters or list items allowed
minExclusive	Specifies the lower bounds for numeric values (the value must be greater than this value)
minInclusive	Specifies the lower bounds for numeric values (the value must be greater than or equal to this value)
minLength	Specifies the minimum number of characters or list items allowed.
pattern	Defines the exact sequence of characters that are acceptable
totalDigits	Specifies the exact number of digits allowed
whiteSpace	Specifies how white space (line feeds, tabs, spaces, and carriage returns) is handled

Table 5.2: Restrictions/Facets of XML Schema datatype

name="address" type="xs:string" minOccurs="0" maxOccurs="unbounded">. The perfect solution is get all possible values between the "minOccurs" and "maxOccurs", so that we can cover all possibilities of the element occurrences. However, the "maxOccurs" value may be assigned as "unbounded"; this means there is no exact boundary of the maxOccurs value. In this case it is impossible to get all values. Furthermore, even when "maxOccurs" has an exact value, generating the instances which include all possible values sometimes may cost too much.

In experiments with testing, boundary values easily cause errors, and the values inside the boundary often have the same probability of causing errors. Therefore there is a method called Boundary Condition [Coe08], which takes the boundaries of the values as the main test cases. In TAXI, in order to reduce the number of derived instances, we apply the Boundary Condition method for selecting values for element occurrences. In Preprocessor, TAXI analyzes and obtains the boundaries for element occurrence. The boundaries include the maximum value and minimum value defined by the "minOccurs" and "maxOccurs" attributes. When "maxOccurs" is assigned with "unbounded", TAXI will give a integer value "N" as its default value, for example the value could be "3".

Additionally, TAXI also takes a middle value between boundaries, and generates two invalid occurrence values; one is a maximum value that is smaller than the minimum boundary, and another is a minimum value that is bigger than the maximum boundary. All these values will be included in the "ISTI_et_CNR_occurs" element, and written in the Initial Intermediate Instance. An example of occurrence rewriting is shown in Figure 5.13.

```
<xs:complexType name="Seq">
  <xs:sequence>
    <xs:element name="A" type="xs:string" minOccurs="0"/>
    <xs:element name="B" type="xs:string" maxOccurs="unbounded"/>
    <xs:element name="C" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

<seuqnce> element with occurrence attribute

```
<ISTI_et_CNR_sequence>
  <A type="xs:string">
    <ISTI_et_CNR_occurs occursID="0">
      <ISTI_et_CNR_min value="0"/>
      <ISTI_et_CNR_middle value="1"/>
      <ISTI_et_CNR_max value="1"/>
      <ISTI_et_CNR_max_invalid value="2"/>
    </ISTI_et_CNR_occurs>
    </A>
    <B type="xs:string">
      <ISTI_et_CNR_occurs occursID="1">
      <ISTI_et_CNR_min_invalid value="0"/>
      <ISTI_et_CNR_min value="1"/>
      <ISTI_et_CNR_middle value="2"/>
      <ISTI_et_CNR_max value="3"/>
    </ISTI_et_CNR_occurs>
  </B>
  <C type="xs:string">
  </C>
</ISTI_et_CNR_sequence>
```

Initial Intermediate Instance

Figure 5.13: Preprocessor of `<minOccurs> and <maxOccurs>`

After the Preprocessor, XML Schema is transformed to Initial Intermediate Instance, which is in XML format, and includes only basic and special elements. From then, all following operations are based on this Initial Intermediate Instance.

After the Preprocessor, the schema "Pets.xsd" become "Pets.xml"; we show this transformation in Figure 5.14. Since we changed the weight of the child element of "Pets", the attributes "modificato" are all equal to "true", and the value of "peso" equal the weight value that we assigned at the beginning.

5.4.2 Subschema Generation

Subschema Generation is the implementation of Functional Unit Derivation in Category-partition. As explained in Chapter 4, children of <choice> are independent from each other, because only one of them can occur in a particular XML Instance. Originally the probability of their occurrence in an XML Document is the same.

In particular, in the Functional Unit Derivation implementation, a set of subschemas are derived by selecting a different child from each <choice> element. When there are more than one <choice> elements presented in an XML Schema, a combination methodology of <choice> children is performed to ensure the set of subschemas represents all possible structures derivable from <choice>.

Subschema Generation takes the Initial Intermediate Instance as input, and does DFS first to find the <choice> elements in it, and then combines the children of all <choice>

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Pets">
    <xs:complexType>
      <xs:choice>
        <xs:element name="Cat" type="xs:string" maxOccurs="unbounded"/>
        <xs:element name="Bird" type="xs:string"/>
        <xs:element name="Dog" type="xs:string"/>
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

XML Schema "Pets.xsd"

```xml
<Pets xmlns:xsi="http:www.w3.org/2001/XMLSchema-instance">
 <ISTI_et_CNR_choice choiceID="0">
  <ISTI_et_CNR_choice_child childID="0">
   <ISTI_et_CNR_pesi modificato="true" peso="0.3"/>
   <Cat type="xs:string">
    <ISTI_et_CNR_occurs occursID="0">
     <ISTI_et_CNR_min_invalid value="0"/>
     <ISTI_et_CNR_min value="1"/>
     <ISTI_et_CNR_middle value="2"/>
     <ISTI_et_CNR_max value="3"/>
    </ISTI_et_CNR_occurs>
   </Cat>
  </ISTI_et_CNR_choice_child>
  <ISTI_et_CNR_choice_child childID="1">
   <ISTI_et_CNR_pesi modificato="true" peso="0.2"/>
   <Bird type="xs:string"></Bird>
  </ISTI_et_CNR_choice_child>
  <ISTI_et_CNR_choice_child childID="2">
   <ISTI_et_CNR_pesi modificato="true" peso="0.5"/>
   <Dog type="xs:string"><Dog>
  </ISTI_et_CNR_choice_child>
 </ISTI_et_CNR_choice>
</Pets>
```

Initial Intermediate Instance "Pets.xml"

Figure 5.14: Example “Pets.xsd” - From XML Schema to Initial Intermediate Instance

elements. The algorithm of combinations can be found in Chapter 4.

If there is no <choice> in the Initial Intermediate Instance, this step will be skipped, and the whole schema is considered as one large functional unit. The class diagram of <choice> analysis is shown in Figure 5.15.

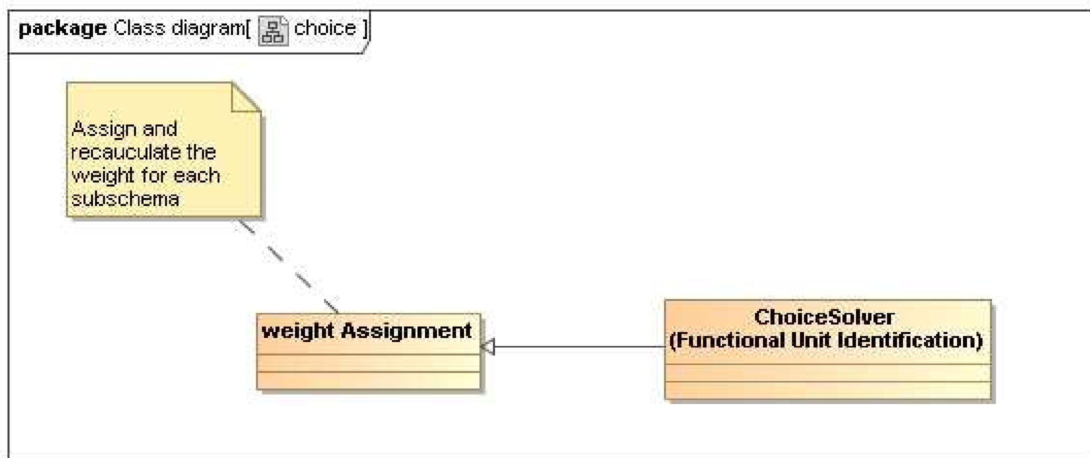

Figure 5.15: Class diagram of `<choice>` analysis

From the class diagram we can see that Choice Analysis class depends on the Weight Assignment class, because during the implementation, Weight Assignment needs to be completed in this step. *Weight* is defined in TAXI to denote the importance of each element. The idea behind this is that the children of the same <choice> might not have the same importance with respect to instances derivation, and therefore the most important ones from the tester’s point of view should be emphasized. TAXI explicitly requires

the user to annotate each child of a <choice> element with a value (called the *weight*), belonging to the [0,1] interval, representing its relative "importance" with respect to the other children of the same <choice>. (Clearly the sum of the weights associated to all the children of the same <choice> element must be equal to 1.) The default assignment is that all children have the same weight. For each option TAXI then derives the so-called "final weight", as a product of the weights of all nodes on the complete path from the root to this node. Such value expresses roughly the idea of how risky that child is and how much effort should be put into the derivation of instances containing it.

Going back to the example "Pets.xml", there are three children of the element "Pets", so three subschemas can be generated. The subschemas of "Pets.xml" are shown in Figure 5.16.

```
<Pets xmlns:xsi="http:www.w3.org/2001/XMLSchema-instance">
  <ISTI_et_CNR_choice choiceID="0">
    <ISTI_et_CNR_choice_child childID="0">
      <ISTI_et_CNR_pesi modificato="true" peso="0.3"/>
      <Cat type="xs:string">
        <ISTI_et_CNR_occurs occursID="0">
          <ISTI_et_CNR_min_invalid value="0"/>
          <ISTI_et_CNR_min value="1"/>
          <ISTI_et_CNR_middle value="2"/>
          <ISTI_et_CNR_max value="3"/>
        </ISTI_et_CNR_occurs>
      </Cat>
    </ISTI_et_CNR_choice_child>
    <ISTI_et_CNR_choice_child childID="1">
      <ISTI_et_CNR_pesi modificato="true" peso="0.2"/>
      <Bird type="xs:string"></Bird>
    </ISTI_et_CNR_choice_child>
    <ISTI_et_CNR_choice_child childID="2">
      <ISTI_et_CNR_pesi modificato="true" peso="0.5"/>
      <Dog type="xs:string"><Dog>
    </ISTI_et_CNR_choice_child>
  </ISTI_et_CNR_choice>
</Pets>
```

Initial Intermediate Instance "Pets.xml"

```
<Pets xmlns:xsi="http:www.w3.org/2001/XMLSchema-instance">
  <Cat type="xs:string">
    <ISTI_et_CNR_occurs occursID="0">
      <ISTI_et_CNR_min_invalid value="0"/>
      <ISTI_et_CNR_min value="1"/>
      <ISTI_et_CNR_middle value="2"/>
      <ISTI_et_CNR_max value="3"/>
    </ISTI_et_CNR_occurs>
  </Cat>
</Pets>

<Pets xmlns:xsi="http:www.w3.org/2001/XMLSchema-instance">
  <Bird type="xs:string"></Bird>
</Pets>

<Pets xmlns:xsi="http:www.w3.org/2001/XMLSchema-instance">
  <Dog type="xs:string"></Dog>
</Pets>
```

Subschemas of "Pets.xml"

Figure 5.16: Example "Pets.xsd" - From Initial Intermediate Instance to Subschema

5.4.3 Occurrence Analysis

This activity focuses on the occurrence manipulation and specifically on the definition of the boundary values of element occurrences.

The class diagram of Occurrence Analysis is shown in Figure 5.17

In Preprocessor, occurrences in XML Schema have already been rewritten to XML format with exact boundary values; a middle value between boundaries, and two invalid values. In the current version of TAXI, only valid instances can be generated automatically; the invalid values are generated in case they are necessary for the future development of TAXI.

Occurrence Analysis combines the values of each occurrence attribute, and gets value combinations that can be used for the Final Instance Generation. The method of combination has already been presented in Chapter 4. In a perfect situation, TAXI should

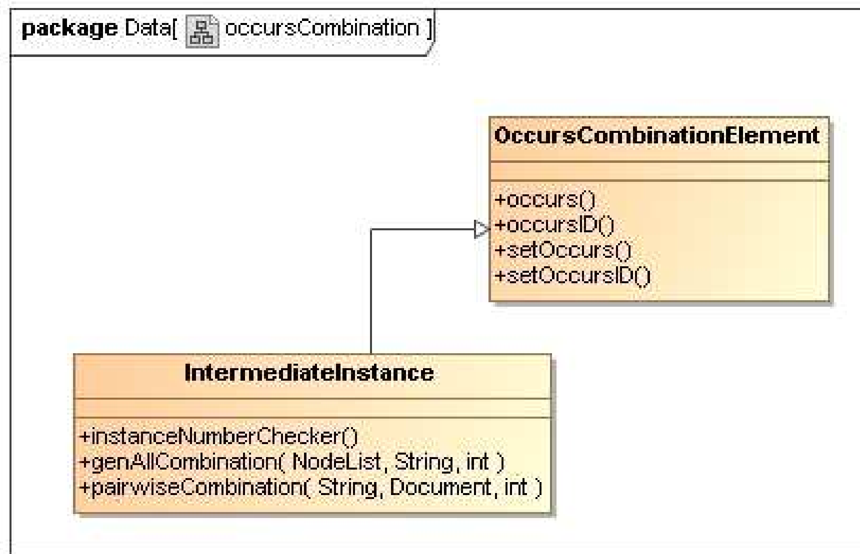

Figure 5.17: Class diagram of occurs analysis

generate XML Documents with all possible combinations of the occurrences, but in the most cases, this is exorbitant. When the XML Schema is large and complex, with a lot of elements, the number of Occurrence Combinations increases geometrically. It is very easy to reach a huge number, and this huge number will lead to a huge number of derived XML Documents. For testing purposes, when there are too many test cases, executing these test will be costly. To avoid this problem, TAXI applies a method to reduce the number of occurrence combinations.

According to our experience in software testing, we know most bugs only require one or two features to be used together, so first the *Pairwise* combination [AWW96], [YL02] is used. We apply this to the boundary values of occurrences. The introduction of pairwise combination can be found in Chapter 2. Using this method a huge number of combinations are eliminated. But then a new problem occurs. When the number of instances is really huge, for example when there are 40 occurrences in a XML Schema, the number of exhaustive combination is 3^{40}, but after pairwise combination, there are only 21 combinations that can be generated. Sometimes this quantity can not satisfy the requirements of testing. When the pairwise combinations are not sufficient for use as test cases, and in order to avoid random selection from the exhaustive combinations, we use a method which can also apply the triangle-wise, quadrangle-wise or even n-wise, ($2 \leq n \leq$ number of occurrences) combinations in an XML Schema or subschema. The method of using n-wise combinations is presented in Chapter 2. During the generation, TAXI will compare the expected number of instances with the number of applied exhaustive combinations and n-wise combinations. Let's call the expected number as E, the number of the exhaustive combinations as T, the number of pairwise combinations as P, and the number of n-wise combinations as N. The strategy for combination method selection is shown below:

```
1.  Compare E with T
2.      if E ≥ T or E ~ T
3.      apply exhaustive combination
4.      else if E*10 < T
5.      apply pairwise combination
6.      if E ≤ P
7.          break
8.      else
9.          do n-wise combination (n ∈ [3, number of
occurrence in the XML Schema])
10.           until P+N ≥ E
```

As shown in this strategy, for each subschema, TAXI first compares the expected number with the number of all possible combinations, if the required number is equal or greater, then all combinations of occurrence values are applied. Otherwise TAXI compares the required number with the number of combinations after pairwise; if the required number is equal or less than the number of pairwise combinations, then TAXI applies pairwise. Otherwise TAXI applies n-wise combinations until the generated combinations equals the required number.

After Occurrence Analysis, the combinations of occurrence values are derived that will be used for Intermediate Instance Generation. For instance, there is only one subschema of "Pets.xsd" with the element "cat" requiring Occurrence Analysis. "Cat" has an attribute "maxOccurs" and has already been assigned a value in the Preprocessor, (see Figure 5.14). Since there is only one occurrence attribute in the schema, the combination is simple, and three combinations are derived, which are 1, 2, 3. This means the occurrence value of "Cat" can be any one of these three values.

5.4.4 Intermediate Instance Derivation

This activity completes the implementation of the Test Specification Generation step. A set of Intermediate Instances is derived based on Initial Intermediate Instance by assigning values to the occurrences of the elements. Each Occurrence Combination generates one Intermediate Instance; if there are subschemas, Intermediate Instances are generated from all subschemas.

Element occurrence attributes in Intermediate Instance have exact values. These values have been defined by combinations that are derived during the Occurrence Analysis activity. The Intermediate Instance is similar to the Initial Intermediate Instance, but elements in the Intermediate Instance have exact values of occurrence. These are the basis of the Final Instance Generation.

For example, there are five Intermediate Instances derived from the schema "Pets.xsd", because one of its subschemas with the element "Cat" has an occurrence attribute, so there are three Intermediate Instances generated from it, and another two instances are generated from the subschemas with the elements "Bird" and "Dog". Figure 5.18 shows these five Intermediate Instances.

```
<Pets xmlns:xsi="http:www.w3.org/2001/XMLSchema-instance">
  <Cat type="xs:string">
    <ISTI_et_CNR_occurs occursID="0">
      <ISTI_et_CNR_min_invalid value="0"/>
      <ISTI_et_CNR_min value="1"/>
      <ISTI_et_CNR_middle value="2"/>
      <ISTI_et_CNR_max value="3"/>
    </ISTI_et_CNR_occurs>
  </Cat>
</Pets>

<Pets xmlns:xsi="http:www.w3.org/2001/XMLSchema-instance">
  <Bird type="xs:string"></Bird>
</Pets>

<Pets xmlns:xsi="http:www.w3.org/2001/XMLSchema-instance">
  <Dog type="xs:string"></Dog>
</Pets>
```

Subschemas of "Pets.xsd"

```
<Pets xmlns:xsi="http:www.w3.org/2001/XMLSchema-instance">
  <Cat type="xs:string"></Cat>
</Pets>

<Pets xmlns:xsi="http:www.w3.org/2001/XMLSchema-instance">
  <Cat type="xs:string"></Cat>
  <Cat type="xs:string"></Cat>
</Pets>

<Pets xmlns:xsi="http:www.w3.org/2001/XMLSchema-instance">
  <Cat type="xs:string"></Cat>
  <Cat type="xs:string"></Cat>
  <Cat type="xs:string"></Cat>
</Pets>

<Pets xmlns:xsi="http:www.w3.org/2001/XMLSchema-instance">
  <Bird type="xs:string"></Bird>
</Pets>

<Pets xmlns:xsi="http:www.w3.org/2001/XMLSchema-instance">
  <Dog type="xs:string"></Dog>
</Pets>
```

Intermediate Instances of "Pets.xsd"

Figure 5.18: Example “Pets.xsd” - From Subschema to Intermediate Instance

5.4.5 Final Instance Derivation

This is the last step of the generation process. The XSA component contacts TSS and VS to confirm the number of Final Instances, and the method for getting the element values. The class diagram of Final Instance Derivation is shown in Figure 5.19

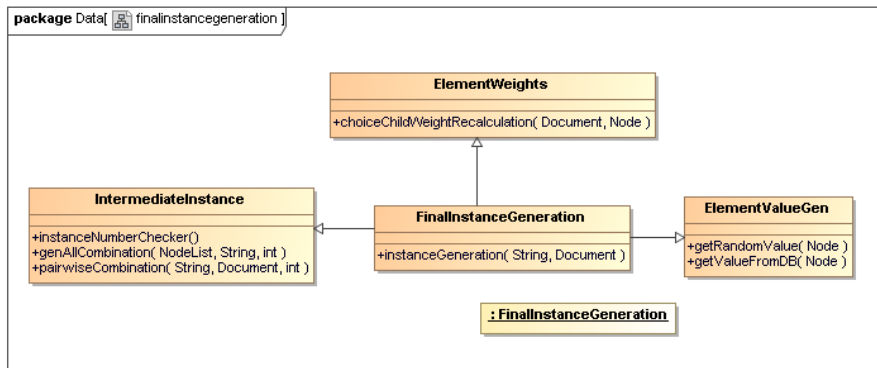

Figure 5.19: Class diagram of Final Instance Derivation

The Final Instance is generated based on the Intermediate Instance by assigning a value to the element. There are three ways to get the element value: it could be generated randomly by TAXI according to type and facet of the element; or picked from a user-defined value set in the database; or if the element value has already been defined by the XML Schema, TAXI will get it and assign to the corresponding element.

If the element does not have a schema-defined value, by default TAXI first checks the database, and finds if there is a value file that has the same name as the corresponding element. If a value file is found, then TAXI gets the values from the value file and takes one

of those predefined values randomly. Otherwise if there is no correspond value file for an element in the database, TAXI then generates a random value automatically. For random value generation, TAXI takes into consideration the elements type and facet specified in the XML Schema definition. For example, if the element type is “integer” constrained by *minInclusive*=“100”, TAXI will select an integer from the interval [100, 2147483647]. Although in its current version, TAXI can generate only valid instance, for the future extension, there is also an invalid value set [-2147483648,100) built in TAXI for applying invalid instance generation, for valid test cases generation, TAXI will not select values from the invalid set.

In order to reduce the number of derived instances, in TAXI, the combination of element values is omitted, even if there are the predefined values in the database; TAXI takes those values by random selection. Figure 5.20 shows all Final Instances with distinct structures from “Pets.xsd”. In order to show the different between database value selection and random value generation by TAXI, we set values for the element “Cat” as *“Singapura, Exotic, Ragdoll, Ocicat, Somali, Chartreux, Burmese, Korat, Maine, Chartrux, Laperm”*, and use random values for the elements “Dog” and “Bird”.

```
<Pets xmlns:xsi="http:www.w3.org/2001/XMLSchema-instance">
  <Cat type="xs:string"></Cat>
</Pets>

<Pets xmlns:xsi="http:www.w3.org/2001/XMLSchema-instance">
  <Cat type="xs:string"></Cat>
  <Cat type="xs:string"></Cat>
</Pets>

<Pets xmlns:xsi="http:www.w3.org/2001/XMLSchema-instance">
  <Cat type="xs:string"></Cat>
  <Cat type="xs:string"></Cat>
  <Cat type="xs:string"></Cat>
</Pets>

<Pets xmlns:xsi="http:www.w3.org/2001/XMLSchema-instance">
  <Bird type="xs:string"></Bird>
</Pets>

<Pets xmlns:xsi="http:www.w3.org/2001/XMLSchema-instance">
  <Dog type="xs:string"></Dog>
</Pets>
```

Intermediate Instances of "Pets.xsd"

```
<Pets xmlns:xsi="http:www.w3.org/2001/XMLSchema-instance">
  <Cat>Ragdoll</Cat>
</Pets>

<Pets xmlns:xsi="http:www.w3.org/2001/XMLSchema-instance">
  <Cat>Burmess</Cat>
  <Cat>Korat</Cat>
</Pets>

<Pets xmlns:xsi="http:www.w3.org/2001/XMLSchema-instance">
  <Cat type>Laperm</Cat>
  <Cat type>Ragdoll</Cat>
  <Cat type>Somali</Cat>
</Pets>

<Pets xmlns:xsi="http:www.w3.org/2001/XMLSchema-instance">
  <Bird>irjtklhtd</Bird>
</Pets>

<Pets xmlns:xsi="http:www.w3.org/2001/XMLSchema-instance">
  <Dog>iOFljif</Dog>
</Pets>
```

Final Instances of "Pets.xsd"

Figure 5.20: Example “Pets.xsd” - Final Instances

TAXI has an embedded XML Schema validator, so when an instance is generated, it will be validated against the input XML Schema immediately. If it does not conform to the input XML Schema, TAXI will write the details of the validation information into the log file, and give an error message at the end of the generation. The XML Schema validator ensures that the instances derived by TAXI are valid. If there is an invalid instance, the user will also get a clue from the log file, which helps to find the bugs in TAXI.

5.5 Implementation of TSS

A common problem with methodologies like Category-partition is explosion of generated test case numbers. Similarly, the XPT methodology suffers from the same problem. As described before, to partially overcome this explosion in TAXI, we have implemented a set of weight-based strategies that permit the user to focus on the most important parts of the Schema.

5.5.1 Application of Weights

Because XML Schema can refer and redefine elements flexibly, there are some cases where we need recalculate the weights.

Recalculation For <choice> Combination

When there is more than one <choice> element in a schema, then there will be a combination for their children.

For example, let's consider two <choice> elements in an XML Schema, each of them having two child nodes. The first <choice>, *choiceA*, has two children, *choiceA1* assigned with a weight of 0.3, and *choiceA2* with weight of 0.7. The second <choice> element is *choiceB*, and its children are called *choiceB1*, with an assigned weight of 0.2, and *choiceB2*, with a weight of 0.8.

When we combine these two elements, we get four subschemas, each of them including one <choice> child combination. So we need to recalculate the weight for the subschemas according to the child elements they include. The weight of each subschema is equal to the product of the weights of its including <choice> child elements, as shown in Figure 5.21. So the weight of these four trees are: $0.3 * 0.2 = 0.06$; 0.3 * 0.8 = 0.24; 0.7 * 0.2 =0.14; 0.7 * 0.8 = 0.56. The sum of for subschemas' weights is still equal to 1.

Recalculation For <choice> Rewriting

XML Schema gives the possibility of rewriting an element by <extension> or <restriction> and <redefine>, a <choice> could be referred and a node added or deleted. In that case, we need to recalculate to keep the weight of the redefined <choice> element as 1. If there is a new node added to the original <choice> element, we give the new node a default weight of "0.5", and distribute the remaining weight to each node according to their proportion in the <choice> element. We will use the above *choiceA* to do an example: if we add a new node *choiceA3*, assigned the default value "0.5". the weight of *choiceA* is 0.3 + 0.7 + 0.5 =1.5. The total weight of *choiceA* now is 1.5, but we know it is required to be always 1, so we recalculate the weight of each node in proportion its to original weight. As result the new weights of each child node is: *choiceA1*: 0.3 /1.5 = 0.2; *choiceA2*: 0.7 / 1.5 = 0.47; and *choiceA3*: 0.5 / 1.5 = 0.33. After the recalculation the weight of *choiceA* returns to 1, and the proportion of each node is not changed.

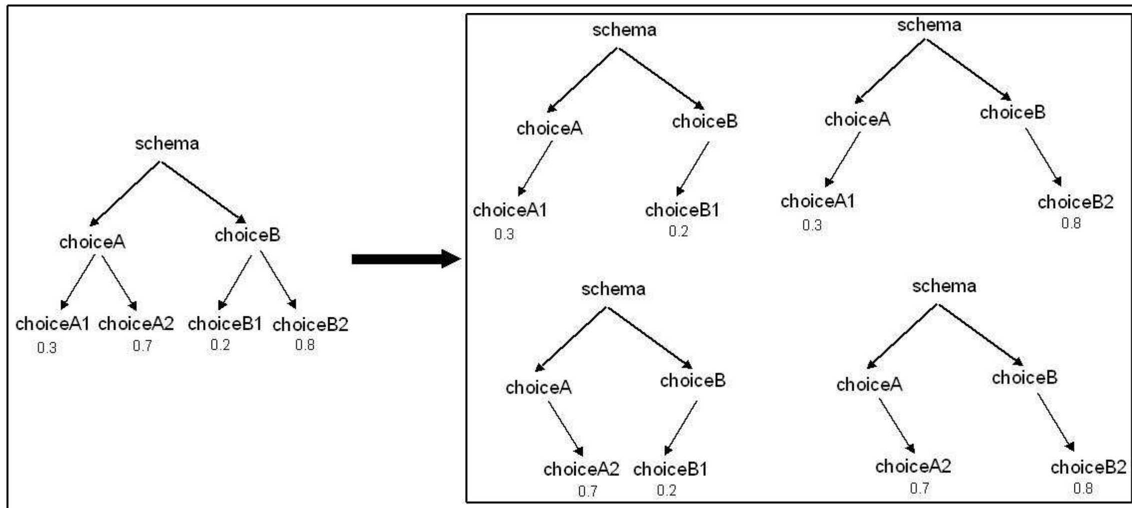

Figure 5.21: Weight recalculation

If a child node is deleted from <choice>, the distribution method is similar to the previous case. We just assign the lost weight to the nodes that are still in the <choice> element. For instance, suppose a <choice> element has three children, the weight of them being: 0.3, 0.3, 0.4, but in another element that refers to this <choice>, one child element, weighted 0.4, is deleted. In this case the weight of this <choice> element becomes 0.6. What Weight Assignment does is just redistribute the weight of the <choice> according to the original children's weights, so that finally the weights of the two remaining children become: 0.3 / 0.6 =1.5; 0.3 / 0.6 = 0.5, and the total weight remains 1.

5.5.2 Strategy Selection

As presented in Chapter 4, TAXI provides four test strategies to formulate the results of a derived instance set. These are: are:

generate with a fixed number of instances

In particular this strategy could be a case in which a finite set of test cases must be derived. The instances are generated from all subschemas according to their final weights. For implementation, first read the number input by the user; let's call it N. As presented in the previous section, each subschema has its weight calculated by TAXI according to the weights of the <choice> child elements it contains. TAXI then distributes N to the subschemas according to their weights. For instance, if there are three subschemas, and the weights are weight$_{subschema1}$="0.3", weight$_{subschema2}$="0.5", weight$_{subschema3}$="0.2"; and $N = 100$, then the number of instances from subschema$_1$ should be 100 * 0.3 = 30; the number of instances from subschema$_2$ should be 100 * 0.5 = 50; and the number of instances from subschema$_3$ should be 100 * 0.2 = 20;

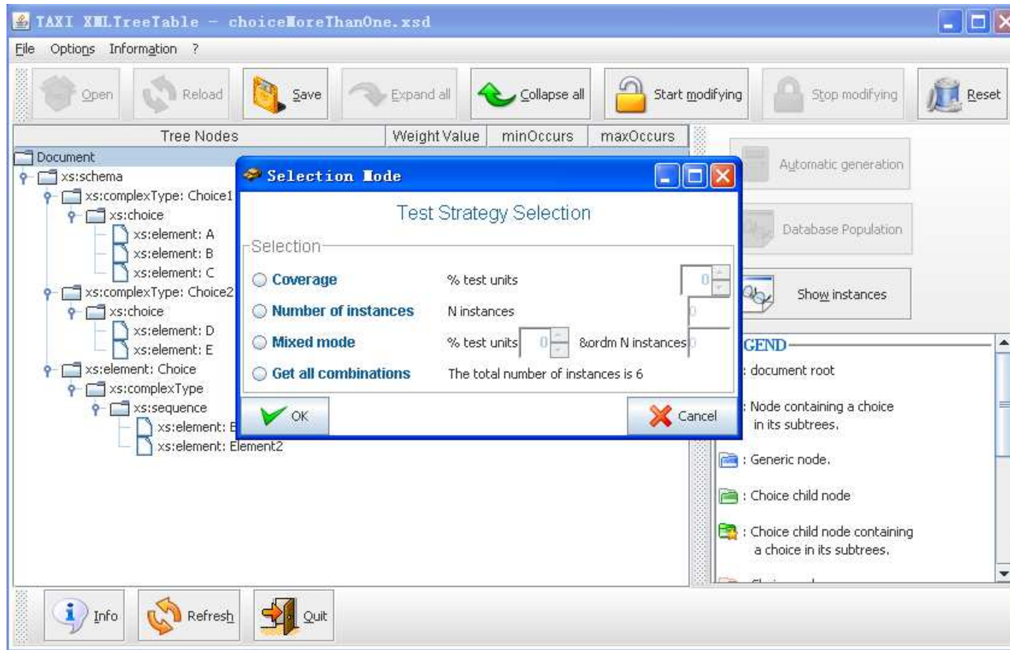

Figure 5.22: Interface of Test Strategy Selection

So in this first test strategy, for each subschema, the expected number of instances times the subschema weight is equal to the number of instances that should generate from that subschema.

generate with fixed functional coverage

When a certain percentage of functional test coverage is established as an exit criterion for instance generation (and then subsequently for testing), the subschemas are selected from the heaviest to the lightest by their weight, until the sum of weights is equal to or greater than the coverage established. Then all instances from these subschemas are generated. Using the same example as in the first strategy, there are three subschemas with the weights "0.3", "0.5", and "0.2"; the coverage input by user is "80%". Therefore $subschema_2$ will be first selected since it is the heaviest one of these three subschemas, but the total weight does not reach 80%. As the second heaviest subschema, $subschema_1$ will be selected next. Now the total of the weights is 0.5 + 0.3 = 0.8, so TAXI stops the subschema selection, and the instances will be generated. Using only $subschema_1$ and $subschema_2$. When this strategy is selected, be aware that TAXI will generate all possible instances from each selected subschema. Sometimes this will leads a very huge number of instances. To avoid this situation, it is better to use the third strategy that combines the first two strategies together.

generate with a fixed functional coverage and number of instances

This is a strategy that combines the two strategies mentioned before. TAXI first selects the proper substructures useful for reaching a certain percentage of functional coverage just as in the second test strategy, then derives the proper number of instances, considering the selected subschemas weights. The rules for the combined method selection are the same as for the first strategy.

generate all possible instances

With this strategy, the user can get all possible instances from all the all subschemas. Sometimes the number is very large, even unreasonable. In this case TAXI will suggest a number to avoid huge time costs for the generation.

On one hand the application of weight-based strategies might decrease the overall effectiveness of the test cases, but on the other hand, by these strategies the test effort can focus on the most critical parts of the schema, and omit the instances affecting those parts that have only marginal impact on the overall system behavior, and the functionalities that are considered as trustable.

5.6 Summary

In this chapter we presented the implementation of TAXI, which use XPT methodology. TAXI is graphical tool, written in Java. In the chapter we described the tool by its components: XSA (XML Schema Analysis) which is the core part of the tool, used to apply the XPT methodology; TSS (Test Strategy Selector), which applies to the selection of three test strategies, VS (Value Storage) which stores and manages the values in a database for generating the final instances; and UI (User Interface) which provides a graphic interface for communicating with users. We presented in detail the implementation of TAXI by a sequence of activities. Using this tool it is possible to prove XPT methodology by doing case studies. We have also tried to improve the tool, to make it meet the requirements for application in the industry.

Chapter 6

Cover Set Of TAXI

This chapter presents the test suite that is used to test the TAXI tool.

As already presented in Chapter 2, XML Schema defines a set of element and attributes. In order to guarantee the correctness of TAXI implementation, we have created a set of test cases to test TAXI. We firstly created the basic XML Schemas for each type of elements depending on the specification of element, which includes only elements, not any attributes. The specification of XML Schema gives a very clear description of what attributes and constraints can be involved in an element. According to the specification, we have created a set of XML Schema files based on the basic schema, which can present possible structures for the target element.

In the following, the test case schemas will be presented; the test cases are designed by the syntax of XML Schema elements. For almost every for each possible child of the element we create a test case; a road map of the test cases is shown in Table 6.1. In this chapter, for each test point (the test points include the element, attribute and facets specified in the definition of XML Schema), we show first a simple schema as a test case without any constraints and attributes, and then show several extended schemas depending on the possible structures that can be included in this element.

For clarity, the XML Schema for each test case will be shown, but not all the derived XML Instances will be shown in the thesis. We will select and show several representative ones from the derived set. Because even though for some test cases we ask TAXI to generate less then fifty test cases, it takes a lot of space to list all of them, and it would make the test cases difficult to read.

6.1 Test Cases of <element>

As the syntax of XML Schema, <element> could include some specific attributes, for instance “ref”, “type”, “abstract” and so on. We can not test all of them with a single element, because some of these attributes can be used only if the parent of the element is not the schema element. Therefore in this section we give test cases for the single element and for the attributes that can be used for the root single element.

Element	**Children of the elements**		**No.**
<element>	Single element		TC1
	default		TC2
	fixed		TC3
<simpleType>	<restriction>	minInclusive, maxInclusive	TC4
	<union>	restriction, enumeration	TC5
	<list>		TC6
<complexType>	*<simpleContent>*	extension	TC7
		restriction	TC8
	<complexContent>	restriction	TC9
		extension	TC10
	<sequence>	element	TC11
		minOccurs, maxOccurs	TC12
		attribute	TC13
		attributeGroup	TC14
		element with “ref” attribute	TC15
		group	TC16
		sequence	TC17
		choice	TC18
	<all>	element	TC19
	<choice>	element	TC20
		minOccurs, maxOccurs	TC21
		group	TC22
		choice	TC23
		sequence	TC24
		two parallel choice elements	TC25
		apply test strategies	TC26
<redefine>	<complexType>	extension	TC27
	<simpleType>	enumeration	TC28
<any>			TC29
<anyAttribute>			TC30

Table 6.1: Road map of the test cases

6.1.1 <element>

<element> is used to define an element in the schema; it is the most important element in the schema. The simplest schema can include only one simple element, for instance XML Schema 1 that is shown in Figure 6.1. This schema includes only one element named “name”; the element type is “xs:string”. We predefine some values of the element “name”, which are “*Mary, Michael, Tom, Brook, John, Sissy, Smith, Jack*”.

XML Schema 1 has only one element; it is one of the simplest structures of XML Schema, and there is only one possible XML structure that can be generated. In order to

test the value selection as well, we want TAXI generate to more than one test case. We have decided to get 30 instances from TAXI, because it is almost three times than the number of element values, although it is not a big number, but enough for testing this simple schema. We have picked three instances randomly from the derived set since the structure of these instances is same and the value of the element is selected randomly (see XML Instance 1 in Figure 6.1).

XML Schema 1	XML Instances 1

`<?xml version="1.0" encoding="UTF-8"?>`
`<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">`
`  <xs:element name="name" type="xs:string"/>`
`</xs:schema>`
	1. `<?xml version="1.0" encoding="UTF-8"?>`
	`<name xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">`
	`Jack`
	`</name>`
	2. `<?xml version="1.0" encoding="UTF-8"?>`
	`<name xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">`
	`Michael`
	`</name>`
	3. `<?xml version="1.0" encoding="UTF-8"?>`
	`<name xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">`
	`Tom`
	`</name>`

Figure 6.1: Test Case 1 single <element>

6.1.2 <element> With "default" Attribute

XML Schema 2, as shown in Figure 6.2, is similar to XML Schema 1, but the element "name" has an attribute "default", which means if the value of "name" is null, then the processor will assign the value of the default attribute to the element. But as presented in Chapter 5, TAXI always provides a value to the element from the predefined value set, or generates it randomly, so that in XML Instances generated from TAXI, the value of a "default" attribute will not be used.

With regard to the test case, we still use the same predefined values of the element name in Test Case 1, which are "*Mary, Michael, Tom, Brook, John, Sissy, Smith, Jack*". XML Schema 2 is very simple, and the derived XML Instances will have the same structure, so it is not necessary to generate the huge number of instances, used in Test Case 1. We have decided to let TAXI generate 30 instances. In these 30 instances, all values are selected, but the default value "Henry" dose not appear. We have selected three instances with different values and have shown them in the Figure 6.2.

6.1.3 <element> With "fixed" Attribute

The $< element >$ could also have the attribute "fixed", which means the fixed value of the element. When the element appears in the instance, then its value must equal the fixed one. XML Schema 3 shown in Figure 6.3, is a schema with an element "name" and an attribute "fixed". In this case, the value of "name" element should be always fixed and equal to "Henry".

``<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="name" type="xs:string" default="Henry"/>
</xs:schema>```
 	
XML Schema 2
	``<?xml version="1.0" encoding="UTF-8"?>
	<name xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
	  Tom
	</name>
	
	<?xml version="1.0" encoding="UTF-8"?>
	<name xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
	  Mary
	</name>
	
	<?xml version="1.0" encoding="UTF-8"?>
	<name xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
	  Smith
	</name>```
	
	XML Instances 2

Figure 6.2: Test Case 2 single <element> with "default" attribute

We still use the same predefined values as in Test Case 1, which are "*Mary, Michael, Tom, Brook, John, Sissy, Smith, Jack*". And since the XML Schema 3 is quite similar to XML Schema 2, we also let TAXI generate 30 instances. After the generation, we found the element "name" in all of these 30 instances has the same value; they are all equal to "Henry". We have selected 3 instances randomly and shown them in Figure 6.3.

``<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="name" type="xs:string" fixed="Henry"/>
</xs:schema>```
 	
XML Schema 3
	``<?xml version="1.0" encoding="UTF-8"?>
	<name xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
	  Henry
	</name>
	
	<?xml version="1.0" encoding="UTF-8"?>
	<name xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
	  Henry
	</name>
	
	<?xml version="1.0" encoding="UTF-8"?>
	<name xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
	  Henry
	</name>```
	
	XML Instances 3

Figure 6.3: Test Case 3 single <element> with "fixed" attribute

6.2 Test Cases of <SimpleType>

In XML Schema the user can define a simple type element by using the tag <simpleType>. The simple type element specifies a simple type element, and the restrictions and information about the values of attributes or text-only elements. The child of the simple type element can be <restriction>, <union> or <list>.

6.2.1 <simpleType> With Child <restriction>

XML Schema 4 (see in Figure 6.4) is a schema that has a simple type element "age". This simple type element includes a child with tag <restriction>. In the tag of <restriction> there are restrictions that are used to define the acceptable values of the XML elements or attributes. Here in Test Case 4, the restrictions are "maxInclusive" which defines the upper boundary of the numeric element values; and "minInclusive", which specifies the

smallest value that the element value could be. In this definition, the valid values of element "age" must between "0" to "100", and these boundary values can be included. We did not set the predefined values before generation; and the values in the instances are generated by TAXI automatically. The generated XML Documents have the same structure, and different values. We have asked TAXI to generate 200 instances, because we wanted to test the generation of <simpleType> with restrictions; also we wanted to test if TAXI could generate the random integers correctly. The results are good; all 200 instances have the expected structure, and the value of their elements are random but all in the range of "0" to "100". Since the structures of the instances are the same, and the values are random, we have just selected two instances randomly, and shown them in Figure 6.4.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="age">
    <xs:simpleType>
      <xs:restriction base="xs:integer">
        <xs:minInclusive value="0"/>
        <xs:maxInclusive value="100"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:element>
</xs:schema>
```

XML Schema 4

```
<?xml version="1.0" encoding="UTF8"?>
<age xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  87
</age>

<?xml version="1.0" encoding="UTF8"?>
<age xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  6
</age>

<?xml version="1.0" encoding="UTF8"?>
<age xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  48
</age>
```

XML Instances 4

Figure 6.4: Test Case 4 <simpleType> with child "restriction"

6.2.2 <simpleType> With Child <union>

The <union> element in XML Schema defines a collection of element values, which could be obtained from more than one specified simple data type. When the simple type element contains a <union> child element, it means the value of the simple type element must be selected from the collection of value data types. As shown in Figure 6.5, XML Schema 5 has an element "jeans_size" with a child <union>, which specifies that the value of "jeans_size" should come from the collection of the values defined by the elements "sizebyno" and "sizebystring". So the XML Document generated from XML Schema 5 should have the same structure, only the element "jeans_size". The value of the element comes from the definition of the simple type elements, "sizebyno" or "sizebystring". The values of the element "sizebyno" in the instances are generated by TAXI; while the values of the element "sizebystring" are taken from the enumeration restrictions by TAXI. We have selected four derived instances in 6.5.

6.2.3 <simpleType> With Child "list"

The list element defines a simple type element as a list of values of a specified data type. In the current version of TAXI, the number of list items (which represents how many

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="jeans_size">
  <xs:simpleType>
    <xs:union memberTypes="sizebyno sizebystring" />
  </xs:simpleType>
</xs:element>

<xs:simpleType name="sizebyno">
  <xs:restriction base="xs:positiveInteger">
    <xs:maxInclusive value="42"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="sizebystring">
  <xs:restriction base="xs:string">
    <xs:enumeration value="small"/>
    <xs:enumeration value="medium"/>
    <xs:enumeration value="large"/>
  </xs:restriction>
</xs:simpleType>
</xs:schema>
```

XML Schema 5

```
<?xml version="1.0" encoding="UTF8"?>
<jeans_size xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  small
</jeans_size>

<?xml version="1.0" encoding="UTF8"?>
<jeans_size xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  38
</jeans_size>

<?xml version="1.0" encoding="UTF8"?>
<jeans_size xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  3
</jeans_size>
```

XML Instances 5

Figure 6.5: Test Case 5 <simpleType> with child “union”

values could be included in the list element) can not be controlled by user. Instead, TAXI will generate a random number for list items from 1 to 10. If there are predefined values in the database, then the value of the list will be selected from the database, otherwise TAXI will generate random values for list elements. In Test Case 6, shown in Figure 6.6, XML Schema 6 have an element “sentence”, which is a string list element. Here we add some values for the sentence; the values are *“How, Are, You, It, Is, A, Good, Day”*.

For XML Schema 6 we want to check if the item number and the list items generate correctly. With regard to the structure of the schema, we ask TAXI to generate 30 instances, which can be generated in a very short time and are enough for this simple schema. In Figure 6.6, we select two of the instances randomly. We can see in these two instances, the values of the “sentence” elements are different both in length and in content. But these strings are meaningless because the words are selected randomly.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="sentence" type="stringList"/>

<xs:simpleType name="stringList">
  <xs:list itemType="xs:string"/>
</xs:simpleType>

</xs:schema>
```

XML Schema 6

```
<?xml version="1.0" encoding="UTF-8"?>
<sentence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  You How Good Good
</sentence>

<?xml version="1.0" encoding="UTF-8"?>
<sentence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  Are Is Day You Good You
</sentence>
```

XML Instances 6

Figure 6.6: Test Case 6 <simpleType> with child “list”

6.3 Test Cases of <complexType>

Complex type is written within the tag <complexType>; it defines an XML element that contains other elements, and/or attributes. We have designed the test cases of $< complexType >$ according to the relationship among the contained elements.

6.3.1 <complexType> With Child <simpleContent>

The <simpleContent> element contains extension or restriction on a text-only complex type or on a simple type as content and contains no element. The parent of <simpleContent> can only be <complexType>. If the <simpleContent> is based on an XSD data type, it can be extended, but not be restricted.

<simpleContent> With <extension>

In the XML Schema 7 of Test Case 7 (see Figure 6.7), the type of element "JeansSize" is an integer, and it has an attribute "country", which has a fixed value "Canada". We use a <complexType> with a <simpleContent> child to present this structure, (see the schema in Figure 6.7). In this test case, we set the accepted values of "JeansSize" as "*34, 36, 38, 40, 42, 44 46, 48*", and for attribute "country" we set the values as "US, China, Italy, France, Spain". Since the structure of the Schema 7 is simple, we do not need a huge number of instances for testing. But we want to check if the values can be assigned properly, we let TAXI to generate 30 instances, it is more than three times of the "JeansSize" values and 10 times the value of "country". After generation, all the instances are valid, and therefore TAXI performed correctly. In Figure 6.7 we show four instances; three of them have the same structure, but the values of "JeanSize" are different, and the values of "country" are the same since the "country" attribute is fixed with the value "Canada". Another instance has a different structure, because the "country" attribute is not a required attribute, so it appears randomly; for example in this instance, it does not occur.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="JeansSize">
  <xs:complexType>
   <xs:simpleContent>
    <xs:extension base="xs:integer">
     <xs:attribute name="country" fixed="Canada" />
    </xs:extension>
   </xs:simpleContent>
  </xs:complexType>
</xs:element>
</xs:schema>
```

XML Schema 7

```
<?xml version="1.0" encoding="UTF8"?>
<JeansSize xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 46
</JeansSize>

<?xml version="1.0" encoding="UTF8"?>
<JeansSize xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" country="Canada">
 12
</JeansSize>

<?xml version="1.0" encoding="UTF8"?>
<JeansSize xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" country="Canada">
 34
</JeansSize>

<?xml version="1.0" encoding="UTF8"?>
<JeansSize xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" country="Canada">
 48
</JeansSize>
```

XML Instances 7

Figure 6.7: Test Case 7 <simpleContent> with <extension>

<simpleContent> With <restriction>

Test Case 8 (in Figure 6.8) shows the restricted simple content element "SmallSizeType", which is based on another simple content element "SizeType". In this test case, no predefined values will be given, besides the < $s$impleContent> structure generation; we want to test the random value derivation of TAXI. Considering XML Schema 8 there is one possible structure that could be generated. So we ask TAXI to generate 50 XML Instances,

which is five times the value number of element “SmallSizeType”; it is a proper number in order to catch the error if the value generation of TAXI does not work properly.

In Figure 6.8 we select four XML Instances, each of them with a different value of “SmallSize”. All of these values are in the range from “0” to “11”, none of them reach the boundaries, and the values of “itemName” are the meaningless strings generated by TAXI randomly.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:complexType name="SmallSizeType">
  <xs:simpleContent>
   <xs:restriction base="SizeType">
    <xs:minExclusive value="0"/>
    <xs:maxExclusive value="11"/>
    <xs:attribute  name="itamName" type="xs:token" use="required"/>
   </xs:restriction>
  </xs:simpleContent>
 </xs:complexType>

 <xs:complexType name="SizeType">
  <xs:simpleContent>
   <xs:extension base="xs:integer">
    <xs:attribute name="itemName" type="xs:token"/>
   </xs:extension>
  </xs:simpleContent>
 </xs:complexType>

<xs:element name="SmallSize" type="SmallSizeType"/>
</xs:schema>
```

XML Schema 8

```
<?xml version="1.0" encoding="UTF-8"?>
<SmallSize xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 itemName="dIyT4O">2
</SmallSize>

<?xml version="1.0" encoding="UTF-8"?>
<SmallSize xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 itemName="NFd0">7
</SmallSize>

<?xml version="1.0" encoding="UTF-8"?>
<SmallSize xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 itemName="0d87e">1
</SmallSize>

<?xml version="1.0" encoding="UTF-8"?>
<SmallSize xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 itemName="87DFi">9
</SmallSize>
```

XML Insances 8

Figure 6.8: Test case 8 <simpleContent> with <restriction>

6.3.2 <complexType> With Child <complexContent>

The <complexContent> element defines extensions or restrictions on a complex type that contains mixed content or elements only. <complexType> is the only possible parent of the <complexContent> element, and it can include <restriction> or <extension> as child.

<complextContent> With <restriction>

Test Case 9 shows the schema with a <comlexContent> element that includes a <restriction> child (see Figure 6.9). The type of element “address” is based on “USAddress”, but it has an attribute “countryCode”. So the “address” should have the same elements as “USAddress”, and the attribute “countryCode”. We set the values for the elements in the schema, in which the values of element “street” are “*Pennsylvania Ave, Hollywoodvine Road, Mission & Valencia Street, Shady Prairie View*”; the values of “city” are “*London, New York, Sydney, Beijing, Rome*”; the values of “zipcode” are “*M2J2C6, 2806, 3455, C7D8K9, 35838*”. The instance generated by TAXI is shown in the frame at the right side of Figure 6.9.

<complexContent> With <extension>

In Figure 6.10, XML Schema 10 shows a schema with <complexContent> and <extension> child. The schema of Test Case 10 has an element “Staff”; its type is “staffinfo”. The

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:complexType name="addressType">
  <xs:sequence>
    <xs:element name="street" type="xs:string" />
    <xs:element name="city" type="xs:string" />
    <xs:element name="zipcode" type="xs:integer" />
    <xs:element name="country" type="xs:string" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="USAddressType">
  <xs:complexContent>
    <xs:restriction base="addressType">
      <xs:sequence>
        <xs:element name="street" type="xs:string" />
        <xs:element name="city" type="xs:string" />
        <xs:element name="zipcode" type="xs:integer" />
        <xs:element name="country" type="xs:string" fixed="US" />
      </xs:sequence>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>

<xs:element name="USAddress" type="USAddressType"/>
</xs:schema>
```

XML Schema 9

```
<USAddress xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <street>
    Hollywoodvine Road
  </street>
  <city>
    New York
  </city>
  <zipcode>
    C7D8K9
  </zipcode>
  <country>
    US
  </country>
</USAddress>
```

XML Instance 9

Figure 6.9: Test Case 9 <complexContent> element with <restriction> child

"staffinfo" is a complex type that extends two more elements based on another complex type "basicinfo". For the value of element "firstName" and "lastName", we set the values of "birthday" as "*10-03-1987, 20-12-1976, 08-10-1977*" and "gender" with the values "*male, female*". We required TAXI to generate all possible instances. There is one instance generated by TAXI, which is shown in the Figure 6.10.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="Staff" type="staffinfo"/>

<xs:complexType name="basicinfo">
  <xs:sequence>
    <xs:element name="firstName" type="xs:string"/>
    <xs:element name="lastName" type="xs:string"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="staffinfo">
  <xs:complexContent>
    <xs:extension base="basicinfo">
      <xs:sequence>
        <xs:element name="birthday" type="xs:string"/>
        <xs:element name="gender" type="xs:string"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
</xs:schema>
```

XML Schema 10

```
<?xml version="1.0" encoding="UTF-8"?>
<Staff xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <firstName>
    Jack
  </firstname>
  <lastName>
    Chen
  </lastname>
  <birthday>
    08-10-1977
  </birthday>
  <gender>
    female
  </gender>
</Staff>
```

XML Instance 10

Figure 6.10: Test Case 10 <complexContent> element with <extension> child

6.3.3 Test Cases of <sequence>

The <sequence> element means that in the related XML Documents, the children of the <sequence> elements must appear in the specific sequence, and the occurrence time of its child elements can be 0 or any number.

Single <sequence> Element

Here we give a simple test case that has a <sequence> element “name”. The children of “name” are “firstName” and “lastName”, and their type is “xs:string” (see XML Schema 11 in Figure 6.11).

As already presented in Chapter 4, we know from XML Schema 11 that there is one valid XML structure that can be generated. Of course with TAXI you can get as many XML Instances as expected, but all of them will have the same structure. We set the values for the elements before generation. For “firstName” we set the content of the value file as “*Mary, Maggie, Michael, Tom, Ivy, Brook, John, Sissy, Eva, Smith, Jack, Randey*”; for the element “lastName” we set the values as “*Smith, Brook, Wolf, Chen, He, Zhang, Johnson, Brown, Gao, Julie, Fisher, Hans*”. In XML Schema 11 there is one possible structure that can be generated, but we want more XML Instances to check if TAXI can handle <sequnce> element correctly. Since XML Schema 11 is a simple schema, we ask TAXI to generate 30 instances, which is thirty times the possible structure. This is sufficient for testing the valid structure generation. The results are good; all of these XML Instances are valid. In Figure 6.11, we select and show three instances randomly, because all these thirty instances have the same structure, and their values are also selected at random by TAXI.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="name">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="firstName" type="xs:string"/>
        <xs:element name="lastName" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

XML Schema 11

```
<?xml version="1.0" encoding="UTF8"?>
<name xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <firstName>Sissy</firstName>
  <lastName>Wolf</lastName>
</name>
<?xml version="1.0" encoding="UTF8"?>
<name xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <firstName>Tom</firstName>
  <lastName>Chen</lastName>
</name>
<?xml version="1.0" encoding="UTF8"?>
<name xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <firstName>Michael</firstName>
  <lastName>Zhang</lastName>
</name>
```

XML Instances 11

Figure 6.11: Test Case 11 <sequence> element

<sequence> Element With “minOccurs” and “maxOccurs”

The children of <sequence> element can occur from 0 to any numbers of times. Test Case 12 shows a schema similar to Test Case 11, but the child elements have occurrence attributes. This schema is shown in the first half of Figure 6.12. According to the combination method mentioned in Chapter 4, from the schema of Test Case 12 there are nine Intermediate Instances that can be generated, so that there are nine different structures of the derived instances. We have used the same values as in Test Case 11, and required 50 instances from TAXI. In Figure 6.12, we select nine XML Instances that all of them are from the different structures, and show them in Figure 6.12.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/xmlSchema">
  <xs:element name="name">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="firstName" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="lastName" type="xs:string" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

XML Schema 12

```
<?xml version="1.0" encoding="UTF8"?>
<name xmlns:xsi="http://www.w3.org/2001/
  XMLSchema-instance">
  <lastName>Smith</lastName>
</name>

<?xml version="1.0" encoding="UTF8"?>
<name xmlns:xsi="http://www.w3.org/2001/
  XMLSchema-instance">
  <firstName>Michael</firstName>
  <lastName>Brook</lastName>
</name>

<?xml version="1.0" encoding="UTF8"?>
<name xmlns:xsi="http://www.w3.org/2001/
  XMLSchema-instance">
  <lastName>Zhang</lastName>
  <lastName>Smith</lastName>
</name>

<?xml version="1.0" encoding="UTF8"?>
<name xmlns:xsi="http://www.w3.org/2001/
  XMLSchema-instance">
  <lastName>Chen</lastName>
  <lastName>Brook</lastName>
  <lastName>Rayn</lastName>
</name>

<?xml version="1.0" encoding="UTF8"?>
<name xmlns:xsi="http://www.w3.org/2001/
  XMLSchema-instance">
  <firstName>Tom</firstName>
  <lastName>Brook</lastName>
  <lastName>Gao</lastName>
</name>

<?xml version="1.0" encoding="UTF8"?>
<name xmlns:xsi="http://www.w3.org/2001/
  XMLSchema-instance">
  <firstName>Sissi</firstName>
  <firstName>Smith<firstName>
  <firstName>Jack</firstName>
  <lastName>He</lastName>
</name>

<?xml version="1.0" encoding="UTF8"?>
<name xmlns:xsi="http://www.w3.org/2001/
  XMLSchema-instance">
  <firstName>Simth</firstName>
  <firstName>Mary<firstName>
  <firstName>Jack</firstName>
  <lastName>Jack</lastName>
  <lastName>Gao</lastName>
</name>

<?xml version="1.0" encoding="UTF8"?>
<name xmlns:xsi="http://www.w3.org/2001/
  XMLSchema-instance">
  <firstName>Mary<firstName>
  <lastName>Zhang</lastName>
  <lastName>Smith</lastName>
  <lastName>Julie</lastName>
</name>

<?xml version="1.0" encoding="UTF8"?>
<name xmlns:xsi="http://www.w3.org/2001/
  XMLSchema-instance">
  <firstName>Brook</firstName>
  <firstName>Jack<firstName>
  <firstName>Michael</firstName>
  <lastName>Hans</lastName>
  <lastName>Zhang</lastName>
  <lastName>Julie</lastName>
</name>
```

XML Instances 12

Figure 6.12: Test Case 12 <sequence> element with occurrence attributes

<sequence> Element With “attribute”

In Figure 6.13, the XML Schema 13 has a <sequence> element “name”, and it has an attribute “ID”. The type of “ID” is integer. Before the instance generation we had not established any values for this attribute, so the values of “ID” are generated randomly by TAXI. For the child element of the element “name” we set its value as “*Mary, Michael, Tom, Brook, John, Sissy, Smith, Jack*”, and asked TAXI to generate 30 instances. Because XML Schema 13 is simple, 30 instances can cover the possible instances. We selected three from the derived instances randomly, and show them in Figure 6.13.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="name">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="firstName" type="xs:string"/>
        <xs:element name="lastName" type="xs:string"/>
      </xs:sequence>
      <xs:attribute name="ID" type="xs:integer" use="required"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

XML Schema 13

```
<?xml version="1.0" encoding="UTF8"?>
<name xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ID="2876">
  <firstName>Tom</firstName>
  <lastName>Wolf</lastName>
</name>

<?xml version="1.0" encoding="UTF8"?>
<name xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ID="23">
  <firstName>Ivy</firstName>
  <lastName>Smith</lastName>
</name>

<?xml version="1.0" encoding="UTF8"?>
<name xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ID="345675">
  <firstName>Jack</firstName>
  <lastName>Chen</lastName>
</name>
```

XML Instances 13

Figure 6.13: Test Case 13 <sequence> element with “attribute”

<sequence> Element With "attributeGroup"

In Figure 6.14, the XML Schema 14 has a <sequence> element "personType", and it has an "attributeGroup" that includes "ID". The type of "ID" is integer and "gender". Before generating the instance we used the same values for "firstName" and "lastName", and "female, male" for the attribute "gender", and "00,01,02,03,04,05,06,07,08,09" for the attribute "ID". Because XML Schema 14 is simple, 30 instances can cover the possible instances. We have selected two of the derived instances, one with the attribute "gender", and one without, because it is not required. The two instances are shown in Figure 6.14.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:attributeGroup name="info">
  <xs:attribute name="ID" type="xs:integer" use="required"/>
  <xs:attribute name="gender" type="xs:string"/>
</xs:attributeGroup>

<xs:complexType name="personInfoType">
  <xs:sequence>
    <xs:element name="firstName" type="xs:string"/>
    <xs:element name="lastName" type="xs:string"/>
  </xs:sequence>
  <xs:attributeGroup ref="info"/>
  </xs:complexType>
  <xs:element name="personInfo" type="personInfoType"/>
</xs:schema>
```
XML Schema 14

```
<?xml version="1.0" encoding="UTF-8"?>
<personInfo xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  ID="01" gender="female">
    <firstName>Julie</firstName>
    <lastName>zhang</lastName>
</personInfo>

<?xml version="1.0" encoding="UTF-8"?>
<personInfo xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ID="09">
    <firstName>Jack</firstName>
    <lastName>zhang</lastName>
</personInfo>
```
XML Instance 14

Figure 6.14: Test Case 14 <sequence> element with "attributeGroup"

<sequence> With <ref> Element

In XML Schema 15, shown in Figure 6.15, the element "personInfo" is a <sequence> element, and two of its children are referred from other elements. Since in this test case we want only to test if TAXI can deal with <ref> correctly, we asked TAXI to generate all instances, then one valid instance is generated. We show this instance on the right side of Figure 6.15.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="personInfo">
  <xs:complexType>
    <xs:sequence>
     <xs:element ref="firstName"/>
     <xs:element ref="lastName"/>
    <xs:element name="gender" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
 </xs:element>
 <xs:element name="firstName" type="xs:string"/>
 <xs:element name="lastName" type="xs:string"/>
</xs:schema>
```
XML Schema 15

```
<?xml version="1.0" encoding="UTF-8"?>
<personInfo xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <firstName>
    Tom
  </firstName>
  <lastName>
    He
  </lastName>
  <gender>
    female
  </gender>
</personInfo>
```
XML Instance 15

Figure 6.15: Test Case 15 <sequence> element with <ref> element

<sequence> In <group> Element

The <group> element defines a group of elements to be used in complex type definitions. The XML Schema 16 shown in Figure 6.16 defines an element "student". The type of the

element refers to a <group> element "infoGroup". Using TAXI we first set the values of the elements in the schema. For the elements "firstName" we set the values as "*Mary, Maggie, Michael, Tom, Ivy, Brook, John, Sissy, Eva, Smith, Jack, Randey*"; for the "lastName" the values are "*Smith, Brook, Wolf, Chen, He, Zhang, Johnson, Brown, Gao, Julie, Fisher, Hans*". There are two values "*male, female*" for element "gender", and the values of "birthday" are "*10-03-1987, 20-12-1976, 08-10-1977*" the same as test case 13. We let TAXI to generate all different instances. Finally we obtained one instance, which is shown on the right side of Figure 6.16. Moreover, TAXI could generate more instances from this schema, and all of them would have the same structure, but with the different values for the elements.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:group name="infoGroup">
 <xs:sequence>
  <xs:element name="firstName" type="xs:string"/>
  <xs:element name="lastName" type="xs:string"/>
  <xs:element name="gender" type="xs:string"/>
  <xs:element name="birthday" type="xs:date"/>
 </xs:sequence>
</xs:group>

<xs:element name="student" type="studentType"/>

<xs:complexType name="studentType">
  <xs:group ref="infoGroup"/>
</xs:complexType>
</xs:schema>
```

XML Schema 16

```
<?xml version="1.0" encoding="UTF-8"?>
<student xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <firstName>
    Maggie
  </firstName>
  <lastName>
    Brook
  </lastName>
  <gender>
    male
  </gender>
  <birthday>
    10-03-1987
  </birthday>
</student>
```

XML Instance 16

Figure 6.16: Test Case 16 <sequence> in <group> element

<sequence> Within a <sequence>

The <sequence> element could also include a complex type, such as in XML Schema 17, shown in Figure 6.17. The child <sequence> element has occurrence attributes. We have used the same value as in Test Case 16, and let TAXI generate all possible instances. We show the instances in Figure 6.17.

<choice> within a <sequence>

The <sequence> element could also include the <choice> as child element, as in XML Schema 18 in Figure 6.18. We have used the same value as in Test Case 16, and let TAXI generate all possible instances. We show the instances in Figure 6.18.

6.3.4 Test Cases of <all> Element

The element <all> defines an element in which each of its child elements can occur zero or one time in any sequence. By the specification of element <all>, the occurrence value could be only "0" or "1", but the order of the child elements are not prescribed. As already presented, TAXI gives a random sequence of <all>'s child elements in each generated

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="personInfo">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="gender" type="xs:string"/>
    <xs:sequence maxOccurs="unbounded">
      <xs:element name="firstName" type="xs:string"/>
      <xs:element name="lastName" type="xs:string"/>
     </xs:sequence>
  </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

XML Schema 17

```
<?xml version="1.0" encoding="UTF-8"?>
<personInfo xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <gender>male</gender>
 <firstName>Randey</firstName>
 <lastName>Smith</lastName>
</personInfo>

<?xml version="1.0" encoding="UTF-8"?>
<personInfo xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <gender>male</gender>
  <firstName>Eva</firstName>
  <lastName>Chen</lastName>
  <firstName>Randey</firstName>
  <lastName>Wolf</lastName>
</personInfo>

<?xml version="1.0" encoding="UTF-8"?>
<personInfo xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <gender>male</gender>
  <firstName>Mary</firstName>
  <lastName>Wolf</lastName>
  <firstName>Ivy</firstName>
  <lastName>Smith</lastName>
<firstName>Sissy</firstName>
<lastName>He</lastName>
</personInfo>
```

XML Instances 17

Figure 6.17: Test Case 17 <sequence> within <sequence> element

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="personInfo">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="firstName" type="xs:string"/>
        <xs:element name="lastName" type="xs:string"/>
        <xs:choice>
          <xs:element name="gender" type="xs:string"/>
      <xs:element name="birthday" type="xs:date"/>
    </xs:choice>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

XML Schema 18

```
<?xml version="1.0" encoding="UTF-8"?>
<personInfo xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <firstName>Tom</firstName>
 <lastName>Johnson</lastName>
 <gender>female</gender>
</personInfo>

<?xml version="1.0" encoding="UTF-8"?>
<personInfo xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <firstName>Fisher</firstName>
 <lastName>Johnson</lastName>
 <birthday>20-12-1976</birthday>
</personInfo>
```

XML Instances 18

Figure 6.18: Test Case 18 <choice> within <sequence> element

XML Document to reduce the number of test case, and cover as many different structures as possible. XML Schema 19 in Figure 6.19 is schema with a <all> element. The schema describes the animal types in a pet shop, which include cat, dog and bird. We set the values for the element “Dog” as “*Bandog, Leonberger, Huntaway, Pekinese, Puggy, Rottweiler, Puli*”; for element “Cat” we set the values as “*Singapura, Exotic, Ragdoll, Ocicat, Somali, Chartreux, Burmese, Korat, Maine, Chartrux, Laperm*”; and the values for “Bird” are “*Partridges, Cockatiels, Rosellas, Mynahs, Kakapo, Budgerigars*”. We let TAXI generate 30 instances, which is 10 times the element, enough to see if the children of <all> occur with a random sequence. The results are good, and we have selected three instances of <all>’s children sequence, and show them in Figure 6.19. In each of them the children of the “petShop” occur in a different sequence.

The children of <all> element can occur only 0 time or once, and the combination of occurrence rules is the same as <sequence>, so we do not show that test case here.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="petShop">
  <xs:complexType>
    <xs:all>
      <xs:element name="Cat" type="xs:string"/>
      <xs:element name="Dog" type="xs:string"/>
      <xs:element name="Bird" type="xs:string"/>
    </xs:all>
  </xs:complexType>
</xs:element>
</xs:schema>
```
XML Schema 19

```
<?xml version="1.0" encoding="UTF-8"?>
<petShop xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Cat>Chartreux</Cat>
  <Dog>Leonberger</Dog>
  <Bird>Cockatiels</Bird>
</petShop>

<?xml version="1.0" encoding="UTF-8"?>
<petShop xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Dog>Pekinese</Dog>
  <Cat>Somali</Cat>
  <Bird>Cockatiels</Bird>
</petShop>

<?xml version="1.0" encoding="UTF-8"?>
<petShop xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Bird>Budgerigars</Bird>
  <Cat>Chartrux</Cat>
  <Dog>Bandog</Dog>
</petShop>
```
XML Instance 19

Figure 6.19: Test Case 19 <all> element

6.3.5 Test Cases of <choice> Element

<choice> element specifies an element that could contain more than one child element, but only one of them can appear within the corresponding element of the generated XML Documents.

<choice> Element

As shown in the Figure 6.20, XML Schema 20 has a <choice> element "Animal", and the child elements "Cat", "Dog" and "Bird". Using TAXI three different Intermediate Instances can be generated, each of them containing one different child element of "Animal". We have used the same values as in Test Case 19 for these three elements in the schema, and ask TAXI to generate 30 instances, which is ten times the intermediate instances, so that we can check if TAXI can deal with all the instances correctly. In Figure 6.20, we show three instances, each of them containing a different <choice> child.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/xmlSchema">
  <xs:element name="Animal">
    <xs:complexType>
      <xs:choice>
        <xs:element name="Cat" type="xs:string"/>
        <xs:element name="Dog" type="xs:string"/>
        <xs:element name="Bird" type="xs:string"/>
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>
```
XML Schema 20

```
<?xml version="1.0" encoding="UTF8"?>
<Animal xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >
  <Cat>Singapura</Cat>
</Animal>

<?xml version="1.0" encoding="UTF8"?>
<Animal xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >
  <Bird>Rosellas</Bird>
</Animal>

<?xml version="1.0" encoding="UTF8"?>
<Animal xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >
  <Dog>Bandog</Dog>
</Animal>
```
XML Instance 20

Figure 6.20: Test Case 20 <choice> element

<choice> With "minOccurs" and "maxOccurs"

Test Case 21 presents the <choice> element with occurrence attributes. In contrast to <sequence> and <all>, the occurrence combinations of <choice> elements are not

based on the whole schema, but on the derived subschemas (The process of how to generate subschemas is presented in Chapter 4). The XML Schema 21 shown in Figure 6.21 is based on Test Case 20; the difference is that the child elements “Cat” and “Bird” have the occurrence attributes. We have still used the same values as in Test Case 20, and let TAXI to generate all different instances. Seven instances are generated by TAXI, and are shown in Figure 6.21.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/xmlSchema">
  <xs:element name="Animal">
    <xs:complexType>
      <xs:choice>
        <xs:element name="Cat" type="xs:string" maxOccurs="unbounded"/>
        <xs:element name="Dog" type="xs:string"/>
        <xs:element name="Bird" type="xs:string" maxOccurs="unbounded"/>
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

XML Schema 21

```
<?xml version="1.0" encoding="UTF8"?>
<Animal xmlns:xsi="http://www.w3.org/2001/
  XMLSchema-instance" >
  <Cat>Chartreux</Cat>
  <Cat>Singapura</Cat>
  <Cat>Exotic</Cat>
</Animal>

<?xml version="1.0" encoding="UTF8"?>
<Animal xmlns:xsi="http://www.w3.org/2001/
  XMLSchema-instance" >
  <Cat>Burmese</Cat>
  <Cat>Ragdoll</Cat>
</Animal>

<?xml version="1.0" encoding="UTF8"?>
<Animal xmlns:xsi="http://www.w3.org/2001/
  XMLSchema-instance" >
  <Cat>Ocicat</Cat>
</Animal>

<?xml version="1.0" encoding="UTF8"?>
<Animal xmlns:xsi="http://www.w3.org/2001/
  XMLSchema-instance" >
  <Bird>Kakapo</Bird>
  <Bird>Mynahs</Bird>
  <Bird>Budgerigars</Bird>
</Animal>

<?xml version="1.0" encoding="UTF8"?>
<Animal xmlns:xsi="http://www.w3.org/2001/
  XMLSchema-instance" >
  <Bird>Kakapo</Bird>
  <Bird>Partridges</Bird>
</Animal>

<?xml version="1.0" encoding="UTF8"?>
<Animal xmlns:xsi="http://www.w3.org/2001/
  XMLSchema-instance" >
  <Bird>Rosellas</Bird>
</Animal>

<?xml version="1.0" encoding="UTF8"?>
<Animal xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance" >
  <Dog>Bandog</Dog>
</Animal>
```

XML Instances 21

Figure 6.21: Test Case 21 <choice> element with occurrence attributes

<choice> In a <group>

Figure 6.22 shows the test case of a <choice> element in a <group>. From the XML Schema 22, TAXI can generate 3 instances which are shown in Figure 6.22.

Two Nested <choice>s

As already known, TAXI will combine the children of <choice> elements to generate the subschemas. When there is only one <choice> element, it is easy. TAXI only separates the child elements of <choice> into different instances, and we have already shown this in Test Case 20. On the other hand if there are more than one <choice> in a schema, the combination becomes complicated.

In Test Case 23 shown in Figure 6.23, the element “Animal” in XML Schema 23 has three child elements, the first two are simple elements that have the type “xs:string”. The last child element “Bird” is also a <choice> element, which has two child elements

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="petShop" type="animalType"/>

<xs:group name="groupOfAnimal">
 <xs:choice>
  <xs:element name="Dog" type="xs:string"/>
  <xs:element name="Cat" type="xs:string"/>
  <xs:element name="Bird" type="xs:string"/>
 </xs:choice>
</xs:group>

<xs:complexType name="animalType">
  <xs:group ref="groupOfAnimal"/>
</xs:complexType>
</xs:schema>
```
XML Schema 22

```
<?xml version="1.0" encoding="UTF-8"?>
<petShop xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <Dog>Leonberger</Dog>
</petShop>

<?xml version="1.0" encoding="UTF-8"?>
<petShop xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <Cat>Ocicat</Cat>
</petShop>

<?xml version="1.0" encoding="UTF-8"?>
<petShop xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <Bird>Partridges</Bird>
</petShop>
```
XML Instances 22

Figure 6.22: Test Case 22 <choice> element within a <group>

"Parrot" and "Sparrow". We have used the same values for element "Dog" and "Cat" as in Test Case 19, and set the values of "Parrot" as "*Kakapo, Macaw*", and the values of "Sparrow" as "*Seaside, Canary*", and let TAXI generate all possible instances. From this schema, four instances are derived, which are shown in Figure 6.23.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/xmlSchema">
  <xs:element name="Animal"/>
    <xs:complexType>
      <xs:choice>
        <xs:element name="Cat" type="xs:string"/>
        <xs:element name="Dog" type="xs:string"/>
        <xs:element name="Bird">
          <xs:complexType>
            <xs:choice>
              <xs:element name="Parrot" type="xs:string"/>
              <xs:element name="Sparrow" type="xs:string"/>
            </xs:choice>
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>
```
XML Schema 23

```
<?xml version="1.0" encoding="UTF8"?>
<Animal xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >
  <Dog>Leonberger</Dog>
</Animal>

<?xml version="1.0" encoding="UTF8"?>
<Animal xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >
  <Bird>
    <Parrot>Kakapo</Parrot>
  </Bird>
</Animal>

<?xml version="1.0" encoding="UTF8"?>
<Animal xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >
  <Cat>Korat</Cat>
</Animal>

<?xml version="1.0" encoding="UTF8"?>
<Animal xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >
  <Bird>
    <Sparrow>Seaside</Sparrow>
  </Bird>
</Animal>
```
XML Instances 23

Figure 6.23: Test Case 23 <choice> within a <choice> element

<sequence> Within a <choice> Element

This test case is used for testing if TAXI can work well when a <sequence> element within a <choice>. In XML Schema 24 (see Figure 6.24), the element "shipForm" is a <choice> element in which one of its children is a <sequence> element. In this test case, we did not predefine any values for the elements, but just let TAXI generate all possible instances. As a result, there are two instances derived, and the elements in the instances have random values that generate from TAXI automatically. We show them in XML Instances 24 in Figure 6.24.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="shipFormType">
    <xs:choice>
      <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="price" type="xs:decimal"/>
      </xs:sequence>
  <xs:element name="info" type="xs:string"/>
    </xs:choice>
  </xs:complexType>
  <xs:element name="shipForm" type="shipFormType"/>
</xs:schema>
```

XML Schema 24

```
<?xml version="1.0" encoding="UTF-8"?>
<shipForm xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <name>IFnleisf</name>
  <price>995.346454</price>
</shipForm>

<?xml version="1.0" encoding="UTF-8"?>
<shipForm xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <info>SolrllFe34IFdg</info>
</shipForm>
```

XML Instances 24

Figure 6.24: Test Case 24 <sequence> within a <choice> element

Parallel <choice> Elements

When an XML Schema contains more than one <choice> elements, their child elements will be combined in the generated XML Documents. As shown in Figure 6.25, XML Schema 25 has the root element "BiologyType", which is a <sequence> element, and two <choice> element children "Animal" and "Plant". For the children of "Animal", we have used the same values as in Test Case 19, for the children of "Plant", the values for "Tree" are "*Apple, Cherry, Baldcypress, Lemon, Maple*", and the values of "Flower" are set as "*Rose, Lily, Lotus, Hydrangea, Tulip*". We have asked TAXI to generate all possible instances. There are four XML Instances generated. From the XML Instances 25 in Figure 6.25 we can see that each of the instance includes a combination of the <choice> child elements.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/xmlSchema">
  <xs:element name="Biology" type="BiologyType"/>
    <xs:complexType name="BiologyType">
      <xs:sequence>
        <xs:element name="Animal">
          <xs:complexType>
            <xs:choice>
              <xs:element name="Dog" type="xs:string"/>
              <xs:element name="Cat" type="xs:string"/>
            </xs:choice>
          </xs:complexType>
        </xs:element>
        <xs:element name="Plant">
          <xs:complexType>
            <xs:choice>
              <xs:element name="Flower" type="xs:string"/>
              <xs:element name="Tree" type="xs:string"/>
            </xs:choice>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
</xs:schema>
```

XML Schema 25

```
<?xml version="1.0" encoding="UTF8"?>
<Biology xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >
   <Animal>
     <Dog>Puli</Dog>
   </Animal>
   <Plant>
     <Tree>Cherry</Tree>
   </Plant>
  </Animal>
</Biology>

<?xml version="1.0" encoding="UTF8"?>
<Biology xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >
   <Animal>
     <Cat>Somali</Cat>
   </Animal>
   <Plant>
     <Tree>Baldcypress</Tree>
   </Plant>
  </Animal>
</Biology>
```

XML Instances 25

Figure 6.25: Test Case 25 two parallel <choice> elements

With Test Strategies

Test Case 26 checks if TAXI can perform test strategies correctly. In this test case, we have used the same schema as in Test Case 20, which is simple, with a <choice> element.

For this generation we have also used the same values for the elements as in Test Cast 20. However, we have used the different test strategies. To do the testing, we needed to assign the weights for the child elements of “Animal” first. In this test case we set the weight of “Cat” as 0.2, the weight of “Dog” as 0.3, and the weight of “Bird” as 0.5. In order to show the schema and the weight clearly, we present the XML Schema 20 as a tree structure.

First we tried the first test strategy, which generates the instances by the given instance number. We set the instance number as 10 (the fact that we set the number as ten does not have particular significance; the user can choose any number they want. Here we use 10 so that it is easier to show all instances in the thesis). As a result we get 10 * 0.2 = 2 instances generated from the subschema that include “Cat” element, 10 * 0.3 = 3 instances from the subschema with “Dog” element, and 10 * 0.5 = 5 instances from the subschema with “Bird” element. We show these instances in Figure 6.26.

Next we will tried second test strategy, which generates the instances by the input coverage with the same weights as the element. The value of the weight can influence which subschemas will be used for instance generation. For example we can set the weight as 80%. Then TAXI will select the subschema beginning with the heaviest on,e until the total weight of selected subschemas is equal to or bigger than the input coverage. With this test strategy, we get two instances, one with the element “Bird” and one with “Dog”. Because the heaviest subschema is with “Bird” child, the weight is 0.5 smaller than the input coverage, so TAXI selects the second heaviest subschema which is the one with the “Dog” element. Then the total weight of the selected subschemas is 0.8 which is equal to the input coverage, so TAXI stops the selection and generates the instances only from the selected subschemas. See these instances in Figure 6.26.

Finally we tried the third test strategy, which mix the first two strategies together. In this case we still set the coverage as “80%” and the number of instances as 10. Finally we get 10 instances, 4 instances with the element “Dog”, and 6 with the element “Bird”. These instances are shown in Figure 6.26.

6.4 Test Cases Of <Redefine>

6.4.1 Redefine The <ComplexType> Element

The <Redefine> element can redefine an element in different XML Schemas. In Figure 6.27, an XML Schema “name.xsd” defines a complex type “person”, which has two children “firstName” and “lastName”. while in the schema “author.xsd”, “person” is redefined by extension with a child “gender”. As a result if we generate from the schema “author.xsd”, we will get an instance with a “author” element, which has three children “firstName”, “lastName” and “gender”. The instance generated by TAXI is shown in Figure 6.27.

6.4.2 Redefine The <SimpleType> Element

In Figure 6.28, the XML Schema “city.xsd” has a simple type “cityType”, which has two enumeration values “LA” and “Pisa”; while in the schema “simpleCity.xsd”, “cityType” is redefined by restriction with only one value “Pisa”. As a result, if we generate from the schema “simpleCity.xsd”, we will get an instance with “smallCity” element, and its value will always be “Pisa”. This instance generated by TAXI is shown in Figure 6.28.

6.5 Test Cases Of <any> and <anyAttribute>

Element <any> and <anyAttribute> are very special in XML Schema; they are used to enable the user to extend the XML Document with elements or attributes that are not specified by the schema. They do not define a specified element or attribute, but give freedom to the user to fill in any attribute he/she expects.

6.5.1 Test Case Of <any> Element

Figure 6.29 shows the XML Schema 29. It defines an element “student”, which is a <sequence> element with three child elements. The last child element is <any> element, which means in the corresponding XML Instances, the user can extend the “student” with any element. With TAXI, when there is the <any> element, TAXI will create an element named “any”, and the value of the element is also filled in with “any”. In this test case, the <any> element has an occurrence attribute, three different instances could be generated. We show these three instances on the right side of the Figure 6.29. Also for the elements “firstName” and “lastName” we use the same values as Test Case 19.

6.5.2 <anyAttribute> Element

Test Case 30 shows a schema with an <anyAttribute> element. Similar to the <any> element, TAXI creates the attribute named “anyAttribute”, filled with the value “anyAttribute”. From the schema in Figure 6.30, we ask TAXI to generate 30 instances. Because “anyAttribute” is not required in the schema, we want more instances to test if it occurs optionally. There are two different structures among these instances since <anyAttribute> is optional. Therefore in some instances the element “student” has the “anyAttribute”, in some it is omitted it. We have selected two: one with the <anyAttribute> and one without. The generated instances are shown on the right side of Figure 6.30.

6.6 Summary

In this section we have presented a set of test cases used for testing the TAXI tool. These test cases are focused on the specific elements and attributes defined by the XML Schema, They help us to ensure that TAXI can deal with the elements and attributes of the XML

Schema in the correct way. For the derived instances, we have not show all instances derived by TAXI, in order to make it easier to read; we have just selected typical ones and have shown them in the test cases. Also, from the Table it is clear we have not presented all test cases used for testing TAXI, because some of them are large, and not easy to understand if we show them in text format. However, the purpose of this chapter is to show the details of the specific element generation, and the correctness of the behaviors of the TAXI tool.

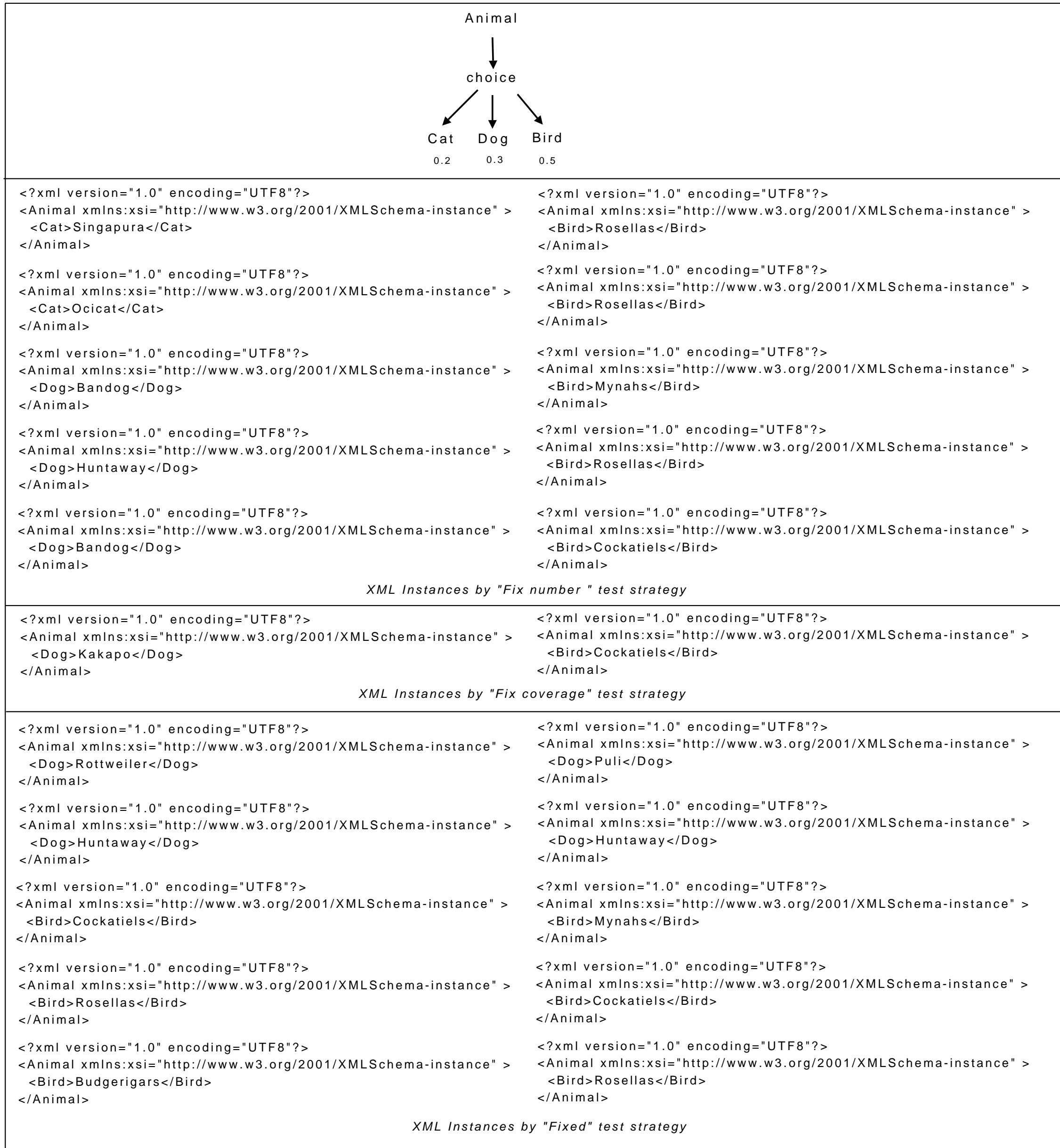

Figure 6.26: Test Case 26 Apply test strategies

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="person">
    <xs:sequence>
      <xs:element name="fistName" type="xs:string"/>
      <xs:element name="lastName" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>

          name.xsd
```

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:redefine schemaLocation="name.xsd">
    <xs:complexType name="person">
      <xs:complexContent>
        <xs:extension base="person">
          <xs:sequence>
            <xs:element name="gender" type="xs:string"/>
          </xs:sequence>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:redefine>
<xs:element name="author" type="personName"/>
</xs:schema>
                    auther.xsd
```

XML Schema 27

```
<?xml version="1.0" encoding="UTF-8"?>
<author>
  <firstName>Sissy</firstName>
  <lastName>Smith</lastName>
  <gender>female</gender>
</author>
```

XML Instance 27

Figure 6.27: Test Case 27 Redefine the complex type element

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:simpleType name="cityType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="LA"/>
      <xs:enumeration value="Pisa"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:element name="city" type="cityType"/>
</xs:schema>
              city.xsd
```

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:redefine schemaLocation="city.xsd">
  <xs:simpleType name="cityType">
    <xs:restriction base="cityType">
      <xs:enumeration value="Pisa"/>
    </xs:restriction>
  </xs:simpleType>
</xs:redefine>
<xs:element name="smallCity" type="cityType">
</xs:schema>          simpleCity.xsd
```

XML Schema 28

```
<?xml version="1.0" encoding="UTF-8"?>
<smallCity>Pisa</smallCity>
```

XML Instance 28

Figure 6.28: Test Case 28 Redefine the simple type element

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="student">
<xs:complexType>
    <xs:sequence>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
      <xs:any minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

xml Schema 29

```
<?xml version="1.0" encoding="UTF-8"?>
<student xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<firstname>Eva</firstname>
<lastname>Brown</lastname>
<any>any<any>
</student>

<?xml version="1.0" encoding="UTF-8"?>
<student xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<firstname>Eva</firstname>
<lastname>Johnson</lastname>
<any>any<any>
<any>any<any>
</student>

<?xml version="1.0" encoding="UTF-8"?>
<student xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<firstname>Sissi</firstname>
<lastname>Julie</lastname>
</student>
```

xml Instances 29

Figure 6.29: Test Case 29 <any> element

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="student">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
    </xs:sequence>
    <xs:anyAttribute/>
  </xs:complexType>
</xs:element>
</xs:schema>
```

XML Schema 30

```
<?xml version="1.0" encoding="UTF-8"?>
<student xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <firstname>Tom</firstname>
   <lastname>He</lastname>
</student>

<?xml version="1.0" encoding="UTF-8"?>
<student xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  anyattribute="anyattribute">
   <firstname>Mary</firstname>
   <lastname>Hans</lastname>
</student>
```

XML Instances 30

Figure 6.30: Test Case 30 <anyAttribute> element

Part III

XPT Applications

Chapter 7

XSLT Transformation Testing

This chapter presents an application of TAXI, which is an automatic validation of an XSLT document.

7.1 Introduction

As introduced in Chapter 2, XML is used as a standard way to interchange information between systems, since it is easily written and read by both the computers and humans. The systems that need to interchange the information may not belong to the same organization, or even be involved in the different application domains. Therefore it is common for the information to be defined by different formats. If the XML Documents need to transfer data between those applications, they must be understood by the system of both sides, so it needs data format transformation that can transform the XML Document from one format, used by one application, to another format, used by the other one. The XSLT language [XSL99] is developed to target this problem.

XSLT (eXtensible Stylesheet Language: Transformations) is a language primarily designed for transforming an XML Document into another XML Document. It can also transform XML Documents to HTML, XHTML and some other text-based formats. XSLT has been a W3C Recommendation since 1999. XSLT creates the formatting structures that interpret and modify the existing XML element, and output the modified structure to the document with the required format, such as HTML, PDF, etc. The formatting structures are written in the stylesheet, which are constructed using template rules that match one or more nodes in the source XML Document, which are selected nodes. The target document will be created by the selected nodes, and compose the structure of them by the definition of XSLT.

Along with the widespread use of XML-based applications, XSLT has become more and more important. The correctness of the XSLT document critically affects the interchange of the systems. Also, as the XML Documents become longer and more complex, correspondingly the complexity of the XSLT stylesheet increases rapidly. Moreover the particular nature of XSLT Stylesheet containing a set of directives destined to be inter-

preted by an XSLT engine makes them certainly not easily checked by human readers. Concerning the properties of TAXI, our goal is to create a test environment for validating the transformation from XML Documents to other XML Documents by XSLT, and for validating the whole process of testing, including test case generation, test execution and the test result analysis totally automatically.

This chapter is structured as follows: in Section 7.2 we give a general idea of our approach to XSLT stylesheet validation; in Section 7.3 we show a case study for XSLT validation to the bibliographic domain. Finally, in Section 7.3.3 we present the comparisons between TAXI and other XML benchmark tools.

7.2 Automatic Validation of XSLT Stylesheet

This section discusses why our approach seems particularly suitable for testing XSLT stylesheets for which the starting and arrival formats are specified using XML Schema.

The general testing framework is based on software applications taking as input data structured in XML format. Such a framework is particularly suitable for stateless applications, as is the case of an XSLT engine that derives an XML file reformatting the data retrieved from another XML file and following the rules defined within an XSLT Stylesheet.

In general, the XML Instance from which the data are retrieved does not have to comply with any kind of XML Schema or other formatting technology. Nevertheless, the availability of an XML Schema specifies the structure of accessible data of the XML, but it is not an exception. Having this assumption in mind, it immediately follows that TAXI can be a powerful tool for generating a relevant set of XML Instances to be provided to the XSLT engine. Also, as presented in the Chapter 2, today in many projects, XML Schema acts as a document for helping users to understand easily the structure of the XML Documents used in the software system. Also when transforming the XML Documents between different applications, it is common for both the source and the target XML Documents to be accompanied by the related XML Schemas. The idea of XSLT stylesheet validation is actually like black-box testing, using TAXI to generate the XML Instances from the source schema, transform the XML Instances by XSLT stylesheet to the target XML Documents, and check if all the transformed XML Documents conform to the target schema. In Figure 7.1 we show the process of XSLT validation with TAXI.

As shown in Figure 7.1, the aim is to get XML Instances from a defined schema by TAXI, and put the instances into the XSLT engine. After one XML Instance is passed, the XSLT engine will provide back another XML Instance.

The availability of an XML Schema for the instances derived and returned by the XSLT engine according to the Stylesheet permits the derivation of a simple oracle to check if the derived instances conform to the corresponding schema. This scenario leads to a *completely automated framework* to check if an XSLT stylesheet is correct.

The test environment is composed of five parts: TAXI tool; the schema of the source XML Documents (here we call it source schema); the XSLT stylesheet (this is the object

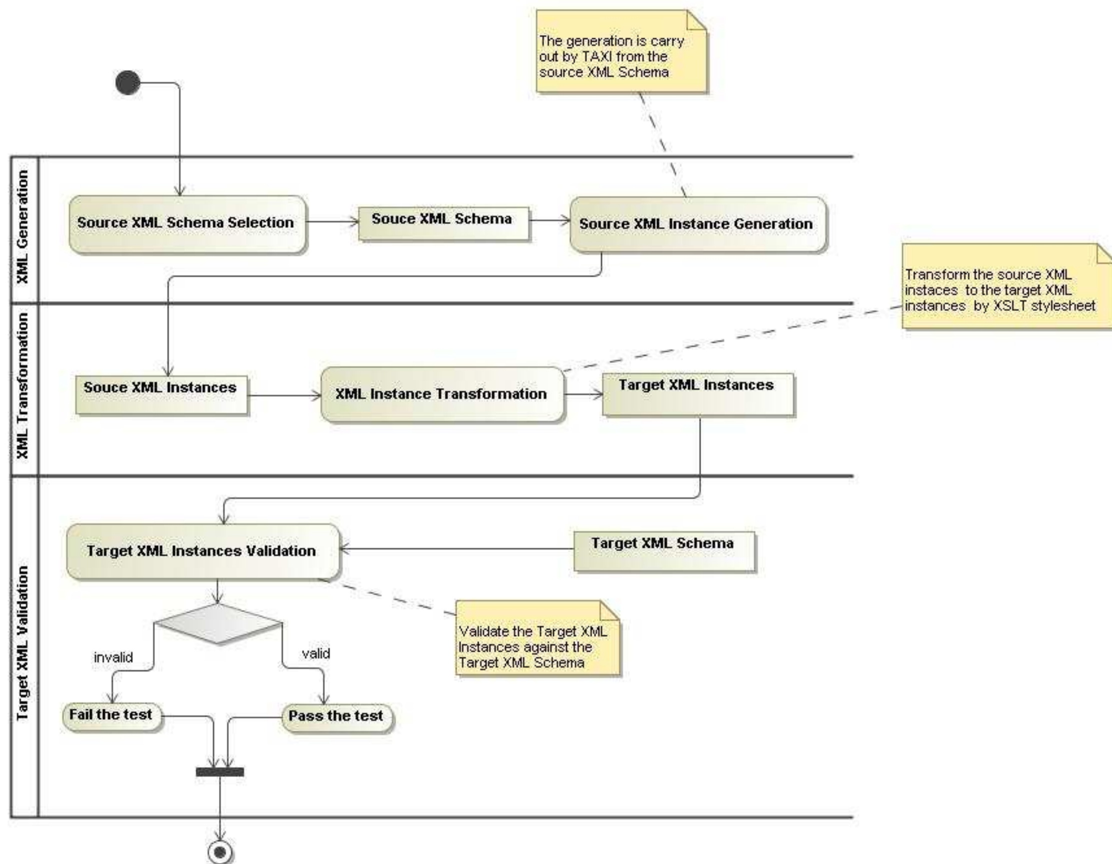

Figure 7.1: Activity diagram of XSLT stylesheet checking

we want to test); the schema of the target XML Documents (we call it the target schema); and the oracle to analyse the results of the testing (it is actually a validator). Then we will describe the details of the approach, follow the steps from the instance generation, to XML Instance transformation, and finally to the result checking.

The method of automatic XSLT validation has three steps: test case generation, XML transformation, and transformed XML validation. TAXI first generates a set of valid instances that conform to the source XML Schema. The number of instances is very important. As presented in Chapter 5, the user could set the number of the instances and create the value files for the elements if necessary. If the full number of instances is not very large, it is better to generate all the possible instances; otherwise it is possible that the omitted instances may cause errors, so that those transformations will not be carried out, and the results of the testing may not be the accurate. If the number of instances is very huge, then the number of instances should at least be equal to or bigger than the instance number with pairwise testing. If the element value is also a factor that can effect the transformation, the user should create value files for the elements before generation, and input the valid values to the files, so that the elements in the derived instances will have meaningful values. The derived instances will be used as test cases in the next step,

and this step is actually the test case derivation.

After the instance generation, the second step is XML Instance transformation. The XSLT stylesheet used in the XSLT engine is the object under test. The testing aims to check if the XSLT stylesheet under test can transform correctly all the instances derived from the source XML Schema. The XSLT engine takes the XML Instances one by one, and transforms them to the target XML Document. This process is executed automatically.

The last step is the test result checking. We use an XML validator as the oracle. When an XML Instance is transformed to the target XML Document, it will be taken to the XML validator and checked immediately whether it is conformed to the target XML Schema. If it is conformed, then this instance is passed; otherwise it is failed, the result will be output to the test report file.

After validation, if all transformed XML Documents are conformed to the target XML Schema, we can say that the XSLT stylesheet has passed the testing. Otherwise it has not passed, and there must be somewhere in the stylesheet that is not correct, and can not transform the XML correctly. The test report gives some clues for bug shooting if the XML Document does not pass validation. The unconformable place will likely be the place where the XSLT stylesheet can not transform properly.

Most importantly, the steps of the XSLT transformation testing are executed completely automatically. The only effort of the tester is to design and populate the element values before testing. However it is not required, if the transformation is not related to the value of elements, in that case value population is not necessary. In the next section, we give a case study that applies this method to testing the transformation between different data models.

7.3 Case Study

In this section we present the case study in which automatically validate an XSLT stylesheet by applying the framework outlined in Figure 7.1.

7.3.1 Marc21 and Dublin Core

This case study is to test the transformation from MARC XML to Dublin Core. For better understanding, we first introduce two important concepts: MARC XML [mar06] and Dublin Core [dub07a].

MARC means *MAchine-Readable Cataloging*: a MARC record is a machine-readable cataloging record, formatted according to MARC 21, which is a set of standards for representing and exchanging disparate data such as authority, bibliographic, classification, community information, and holdings in a machine-readable form. MARC XML is a framework for working with MARC data in an XML environment. This framework is being developed by the Library of Congress' Network Development [LCN06] and MARC

Standards Office [MAR07d]. The MARC XML framework consists of many components, including schemas, stylesheets and software tools.

To specify the structure of MARC complying documents, a MARC XML Schema is created. In the schema all types of MARC 21 records are accommodated, such as: bibliographic, holdings, bibliographic with embedded holdings, authority, classification, community information, and so on. We show the MARC XML Schema in Figure 7.2.

From this Figure, we can see the root element of a MARC record is `collection` with a child `record`. The `record` element represents the structure of the MARC record. It has three children, `leader`, `controlfield` and `datafield`. For all MARC records `leader` is always the first field; it contains data elements, fixed in 24 character positions, that provide processing information of the record [web05]. `Controlfield` is a tagged "00X" field, where different tags yield different meanings. For example, fields 001-006 contain control numbers and other control and coded information. The `Datafield` element is a variable data field tagged with "01X-8XX". There are two indicator position attributes within the `datafield`, which are `ind1` and `ind2`; these indicator positions contain values which interpret or supplement the data found in the field. There is a child element of `datafield` that is `subfield`. The attribute `code` of `subfield` is an alphabetical indicator that specifies the type of information in each `subfield` shown in a MARC record.

The Dublin Core metadata standard defines a set of elements used to describe a wide range of networked resources. The schemas of the Dublin Core defines the structure and syntax of metadata specifications in a formal way. The Dublin Core standard includes `Simple level` and `Qualified level`, so there are qualified schema and simple schema. In this case study we use the simple level schema. The schema defines terms for Simple Dublin Core; there are 15 elements defined in the schema, such as "title", "creator", "subject", "description", "publisher" and so on. All the default content type of the elements is *xs:string* with the attribute *"xml:lang"*.

Along with usage of MARC record, exchanging information from a MARC-based system to other systems is often required, so MARC Standard offices has produced a set of XSLT stylesheets to convert from MARC record to other formats, such as MOD and Dublin Core, as well as stylesheets for the opposite transformation. This case study tests the transformation from MARC XML to Dublin Core, therefore MARC 21 schema is the source schema. We generate MARC XML Instances, and use them as test cases to automatically test the standard stylesheet that is used to transform from MARC XML to Dublin Core metadata. There are three available stylesheets transforming metadata from MARC record to Dublin Core. In this case study we has chosen the one used to transform from MARC to simple Dublin Core, in particular we have chosen the XSLT named "MARC21slim2SRWDC.xsl" [MAR05]. Since MARC is richer in data than Dublin Core, the elements of Dublin Core are only mapped to some data fields of MARC, so specific tag values are required for MARC XML files. Otherwise, if tag values are not specified, it means there will not be the corresponding fields in the Dublin Core. In that case the MARC XML will transform to an empty Dublin Core instance, which has the root element of Dublin Core instance but without any data inside. In the next section we will

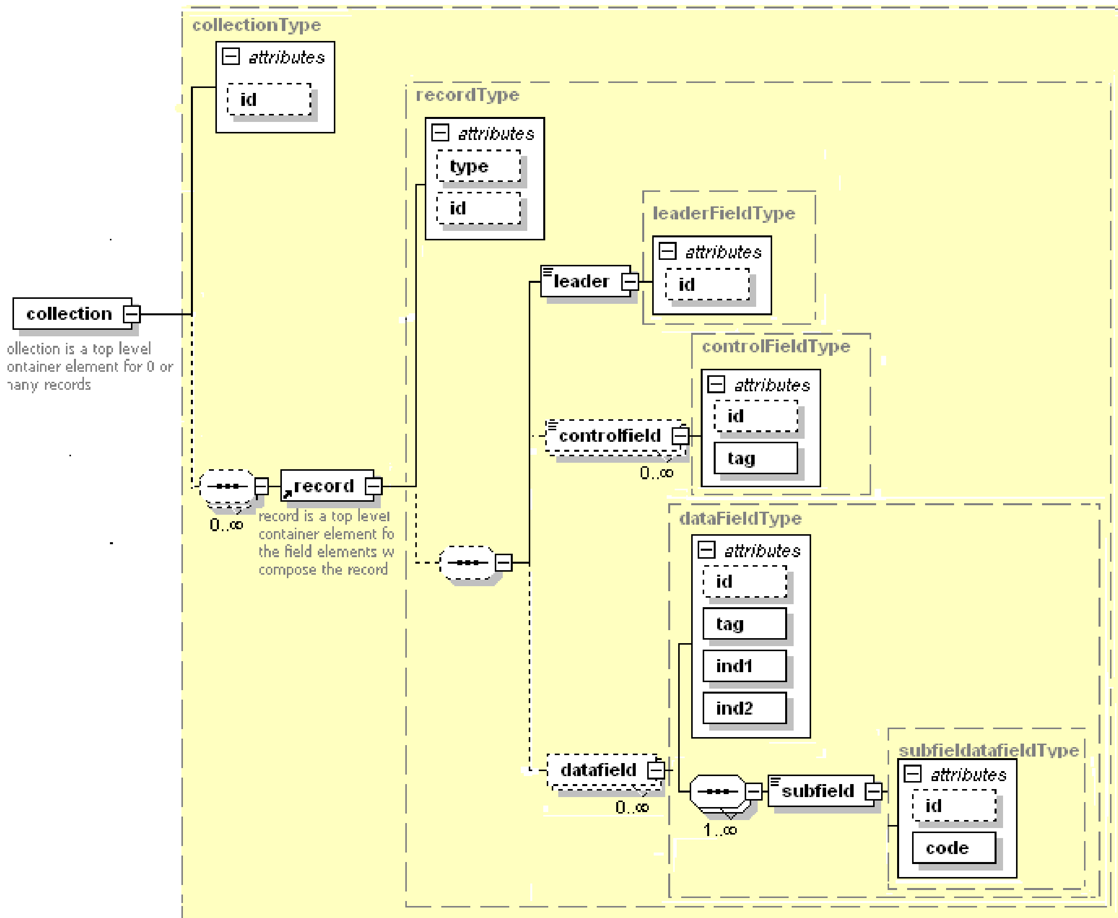

Figure 7.2: MARC 21 XML Schema

present the details of the mapping from MARC to unqualified Dublin Core.

Mapping from MARC to unqualified Dublin Core

As already presented, the MARC 21 fields are listed by field number with specific subfields if applicable. Since Dublin Core is simpler in data than Marc, not all possible Marc subfields can find corresponding Dublin Core elements. Therefore [mar01] gives a table mapping the Marc 21 subfields to Dublin Core, here we focus specifically on the map to unqualified Dublin Core. This is shown in Table 7.1. In this table, the first row is the field code of the MARC XML, the second row is the corresponding Dublin Core element, and the last row is the notes on implementation. In the row of MARC XML, the code written by a number is the datafield, the character "$" is used to specify the subfield used, and the letter after "$" is the code of the subfield. If none is specified, it means all subfields can be used.

MARC Fields	DC Element	Implementation Notes
100, 110, 111, 700, 710, 711	Contributor	
720		
651, 662	Coverage	
751, 752		
	Creator	Creator element not used
008/07-10	Date	
260cg		
500-599,except 506, 530, 540, 546	Description	
340	Format	
856$q		
020$a, 022$a, 024$a	Identifier	
856$u		
008/35-37	Language	
041abdefghj		
546		
260ab	Publisher	
530, 760-787ot	Relation	
506 540	Rights	
534$t	Source	
786ot		
050, 060, 080, 082	Subject	
600, 610, 611, 630, 650, 653		
245, 246	Title	Repeat dc:title for each. Some applications may wish to include 210, 222, 240, 242, 243 and 247.
Leader06, Leader07	Type	
655		

Table 7.1: MARC to Dublin Core Crosswalk (Unqualified) [mar01]

7.3.2 Driving the Generation Process

The process of the automatic XSLT testing approach is according to what was presented in Section 7.2. First, a set of XML Instances is derived from the MARC 21 XML Schema. The instances are then given as input to the XSLT and the resulting XML Instances are validated against the Dublin Core named dc-schema.xsd [dcS04]. The derivation of XML Instances is done by means of the tool TAXI. For validating the XSLT results against the XML Schema of Dublin Core we use instead the validator for javax API [val04]. For conducting the experiment, TAXI and the validator have been integrated in a unique environment, so that for both instances, derivation and checking for conformance of the transformed instances are completely automated.

There is no <choice> element in Marc 21 XML Schema; as presented in Chapter 4 and Chapter 5, the number of derived instances depends on the occurrence numbers. From Figure 7.2 we can see there are 4 elements that have occurrence attributes, so that from MARC 21 XML Schema, TAXI can generate $3^4 = 81$ Intermediate Instances. We show some of them in Figure 7.3. This means there are 81 different instance structures that can be generated. The Intermediate Instances are then used for generating the final instances set by associating appropriately selected values with each element.

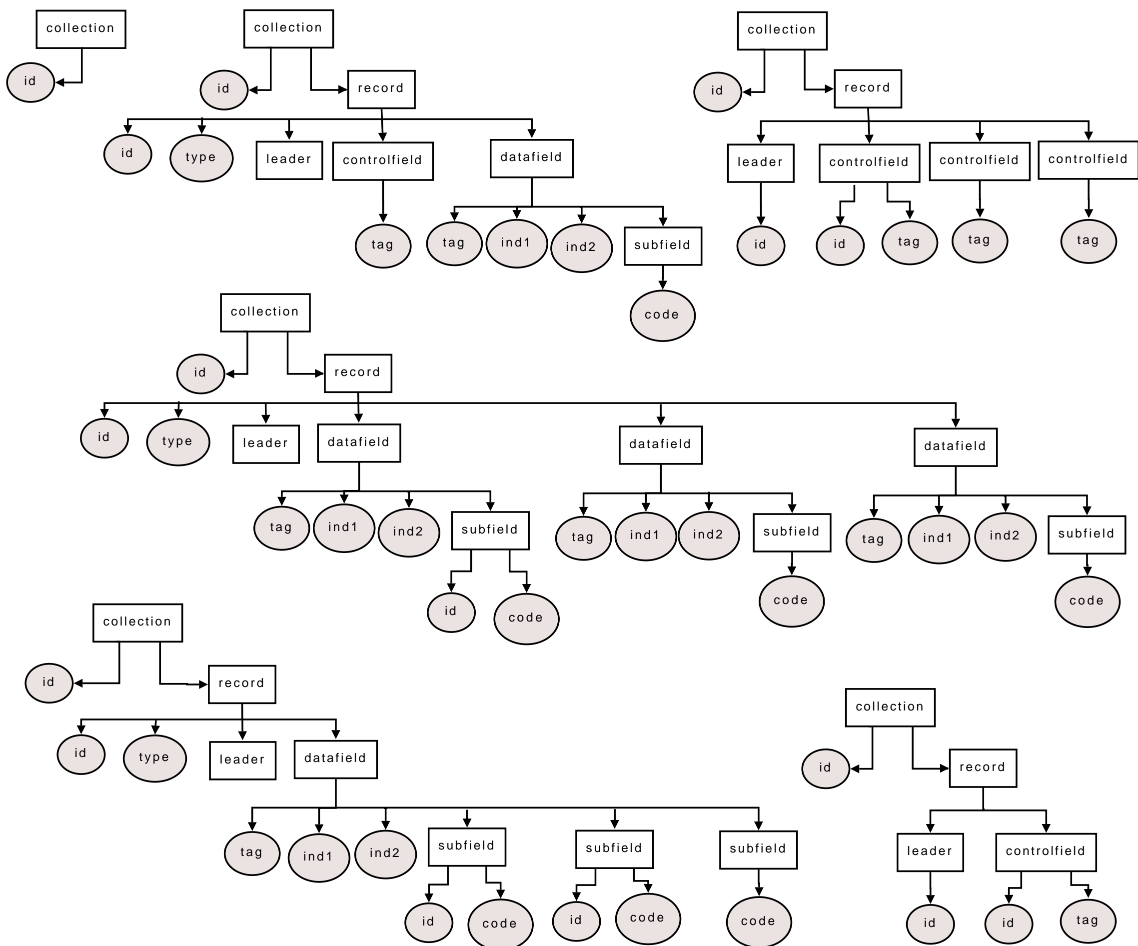

Figure 7.3: Example of Intermediate Instances derived by TAXI

As described in Section 7.3.1, there are some specific values for the elements of MARC Schema that represent different fields of data. So we populate the value files according to those definitions.

- For element “leader”, “leader” is a string with 24 characters. It has no indicators or subfield codes; the data elements are positionally defined. The characters in different positions represent different information, and in each position only some specific characters can be used. For example the characters from 00 to 04 mean

the length of the record; the fifth character means the record status: “a” means an increase in encoding level; “c” means corrected or revised; “d” mean deleted; “n” means new; and “p” means an increase in encoding level from prepublication. The definition of the characters of “leader” can be found in [MAR07c]. We checked the MARC21slim2SRWDC.xsl, and found that only the sixth and seventh characters are mentioned during the transformation. The sixth character represents type of record, in which:
“a” means language material;
“c” means notated music;
“d” means manuscript notated music;
“e” means cartographic material;
“f” means manuscript cartographic material;
“g” means projected medium;
“i” means nonmusical sound recording;
“j” means musical sound recording;
“k” means two-dimensional nonprojectable graphic;
“m” means computer file;
“o” means kit;
“p” means mixed materials;
“r” means three-dimensional artifact or naturally occurring object;
“t” means manuscript language material.

The seventh character represents the bibliographic level. It could be:
“a” means monographic component part;
“b” means serial component part;
“c” means collection;
“d” means subunit;
“i” means integrating resource;
“m” means monograph or item;
“s” means serial.

With regard to the combination possibilities of the sixth and seventh characters, we create 14 * 7 = 98 values for the “leader” element; each value includes one combination of the sixth and seventh characters. The characters of the other positions are selected randomly, but the value can contain all possible combinations of letters of the sixth and seventh character.

- For the element “controlfield”: “controlfield” is used to control numbers and other kinds of coded information control that are used in the machine-readable bibliographic records process. It has the attribute “tag”; the value of “tag” actually contains the information. The value of “tag” could be presented as “XX0”. The meanings of the values are listed below. The details of “controlfield” can be found

in [MAR07a].

"001" means the control number. When "tag" equals 001, the value of "controlfield" represents the organization that creates, uses or distributes the records;
"003" means the control number identifier. When "tag" equals 003 value of "controlfield" is the MARC code for the organization that is mentioned in field 001
"005" means the date and time of latest transaction.
"006" means the fixed-length data elements-additional material, that is the data elements defined by the form of material postionally.
"007" means the physical description fixed field, the positional definition of data elements by category of material.
"008" means the fixed-length data elements, that is the positional definition of data elements by type of material.
In the MARC21slim2SRWDC.xsl, only the last case is mentioned. This means that in Dublin Core, no other cases have related elements. In order to simplify the instances and also meet the requirement of the XSLT, we have given only one value "008" for "tag". When "tag" equal to "008", each character of the value of "controlfield" presents different meaning. In particular the 18-34 characters are in seven separate sections corresponding to the type of material configurations, for instance Books (BK), Computer Files (CF), Maps (MP) and so on. Since there are not specific requirements for the "controlfield" value for the transformation, we have found a set of random values (around 20 values) for this element.

- For the element "datafield": "datafield" has the attribute "tag", "ind1" and "ind2". The value of "tag" represents the data field of the record. As already presented, the values of "tag" can not all be used for the transformation from MARC XML to Dublin Core, A table and list of the accepted values is given in Table 7.1. In [MAR07b] a very detailed introduction of the number signification can be found. We don't describe it here. Therefore the values for "tag" are very clear; they are the values that are listed in able 7.1. In that table also define the value of "code", which is the attribute of "subfield". So we take all values mentioned in the Table 7.1, and put them in the value file of "code". The value of "subfield" is not important during the transformation, so we set it with a set of random values (around 20 strings). For the attribute of "ind1" and "ind2", the value could be a number from 0 to 9, or a single letters. Since their values do not influence the transformation, we set "0" to "9" as the value set of those two attributes.

After value setting, we can start instance generation. The automatic generation of instances from XML Schema can include two different kinds of variability. On one hand it is possible to generate instances that are distinguishable on the basis of their structure (i.e. they can be represented with different tree structures); on the other hand instances can be distinguished on the basis of the value assumed by the various attributes and elements composing the instance.

In order to illustrate whether the number of instance can influence the test result, we have tested the XSLT with different quantities of instances. In the first test, we generate the minimum number of instances, fixing the number of instances as “9”, since we know that with this umber, TAXI will automatically start to apply *pairwise testing*. Then we fix the number of instances as “81” which is the number of Intermediate Instances, then we fix the number as “250”, and “1000”. The results of testing using different numbers of instances are listed in Table 7.2.

Number of generated instances	**Number of test cases rising a failure**	**Number of different faults**
Pairwise (9)	3	1
81	27	1
250	81	1
1000	364	1

Table 7.2: Conducted experiments

We can see that with an increase in the number of test cases, the number of faults increases too, but all of those failures are caused by the same fault. The only failure occurs when the occurrence of “record” is equal to “0”. When the content of “collection” is empty, the transformation can not work properly. The transformed XML Document is not well-formed.

7.3.3 Comparison of TAXI With Other Instance Generators

In the previous case study, we used our tool TAXI to automatically generate XML Instances from the MARC 21 schema. Some other tools exist for XML Instance generation, such as: XMLSpy [XML05b], oXygen [oxy07], Toxgene [tox05], and so on. TAXI, however, surpasses existing tools, as it is unique in its combination of ease of use and powerfulness. In this section we briefly discuss the comparison between TAXI and other XML Instance generators.

Toxgene is a very powerful tool for instance generation; it can generate a large, consistent collection of synthetic XML Documents and can also be used for benchmarking, as can TAXI. However, Toxgene is template-based, thus before it can be applied, the user needs to learn the template specification language and manually write down the generation instructions. In our experience, the effort to write the template increases with the complexity of the XML Schema. In contrast, by using TAXI, the user does not need to make any extra effort to configure the generation. Moreover, in Toxgene the resulting set of instances will heavily depend on the template instructions, while TAXI is configured to systematically consider all elements of the schema. It is possible to use Toxgene to generate the instances of the MARC 21 XML Schema, but the instance coverage can not be guaranteed, and it needs much more preparation works than using TAXI.

XMLSpy and Oxygen are well-known and easy-to-use tools that can generate XML Instances from a schema. The problem is that in reality they can generate only a few specific instances from a schema, whose structure and value are fixed in advance by the tools. Therefore they can not be used to do the testing done by of TAXI, because their test is very dependent on the value of the derived XML Instances. If the values are fixed, and not the values already defined in the MARC standard, the transformed XML Documents will always be empty, so the test can not be executed.

By comparison, TAXI can combine differing numbers of occurrences of the elements, and also provides the capability of interacting with the value sets for differentiating value generation and selection. To our knowledge, it is the only tool that can be used for this kind of XSLT testing without the necessity of high costs for study and preparation.

7.4 Summary

In this chapter we have presented an application of TAXI, an automatic validation of XSLT stylesheets. XSLT stylesheets testing is a domain of particular relevance today; and it seems particularly suitable to the application of our approach. We have used TAXI to generate instances of an input XML Schema, taking those instances as test cases, transforming them by the under test XSLT stylesheet, and validating the transformed XML Document against the target XML Schema. In this case, the process can be completely automated from the generation of instances to the definition of "powerful" oracles, by using the input and target Schema as the model to which the transformed instances should conform. We have also presented a case study that tests the transformation from MARC XML to Dublin Core. In this way, our work has shown some of the first experimental results obtained in the bibliographic domain.

Chapter 8

XML Database Mapping Test

This Chapter presents an application of TAXI for Black-box Testing, Specifically, we will focus on the testing of XML database population.

8.1 Introduction

The most traditional approach to software testing (see, e.g., [Mye04]) is constituted of Black-box Partition testing, in which a system is tested at its I/O interface by identifying relevant classes of input values, and by systematically choosing some representative test input values for each identified class. In three decades of research in software testing, many sophisticated and advanced techniques have been proposed. However partition testing remains the most intuitive approach, and perhaps the most practised on a wide scale, since it only relies on the functional specifications describing the desired I/O behaviour and does not require any special-purpose notation or technical expertise. The idea is that tester examines the functional specification documentation and (ad hoc) selects those input points that (in his/her judgement) exercise the program in relevant ways.

In this chapter we will present a method that applies Black-box Testing to test the population of an XML database. An XML database is a data persistence software system that allows data to be stored on XML format, and enables the data be queried, exported and serialized into any format that the developer wishes [XTY07]. As presented in Chapter 2, XML is a well-structured, self-describing language, that describes the data in a tree structure; all of these features are make XML an advantageous database format.

There are two major classes of XML databases: *XML-enabled databases* (XEDB) and *Native XML databases* (NXD). XEDB takes XML as input, maps all XML to a traditional database (for instance a relational database), and renders XML as output. NXD defines the internal model of databases depending on XML, the fundamental unit of the database for storage is XML Documents [ABC03].

In this Chapter we focus on the application of an XML-enabled database. Along with the widespread using of XML in various application domains, the demand for efficient XML data management is increasing. The major relational database systems such as

IBM DB2 [Rub06], Oracle [Bur06], Microsoft SQL Server [Sin00] are also extending and offering forms of XML support. An XML database could be populated by an XML Document, and with an XML supported database, XML can be stored as an XML Document, a text document or even be built into table structures; all of these require mapping between XML Documents and the database.

Not all XML Documents can map to an XML database. The particular database usually has a schema for defining the structure of acceptable XML Documents. There are various methods for converting an XML Document into a database effectively and automatically, for example, the Oracle XML-SQL Utility, IBM DB2 XML Extender, etc. [Day01].

In this chapter we will use TAXI to test the method for mapping and populating the data of XML Document to XML database. TAXI can generate various instances from the schema that defines acceptable XML Document, and those XML Documents can be used as test cases to test the transformation method from XML Documents to an XML database.

The Chapter is structured as followings: in Section 8.2 we show our concept of XML database mapping testing; and in Section 8.3 we give a case study that uses TAXI to test a mapping method from XML Documents to a MySQL database [DuB07b].

8.2 Black-Box Testing For XML-database Mapping

This section we will discuss the method for testing the data transformation from XML Documents to an XML database.

Data is stored in an XML Document as characters or strings, but the data stored in the database has various data types. If we want to store the data from an XML Document in a database, we must solve the problem of how to map the data in the XML Document to the corresponding XML database. Usually databases do not have the capability to do this transformation, but various methods have been developed to solve this problem. The database vendors such as Oracle, IBM, and Sybase have also developed tools to convert XML Documents into relational tables, for instance XML-SQL Utility (XSU) [Ora] for Oracle database, XML Extender [JC00] for IBM DB2, and Sybase Adaptive Server for Sybase database.

Learning from these tools, we have found that when they do the XML Document to XML database mapping, all of them need rules for defining the XML Document. Some of the rules are definition files, some are functions and some are XML Schemas (DTD or XML Schema). If the XML Documents that can be accepted by the tool are defined by XML Schema, TAXI is very suitable for using to test this tool.

Since only XML Documents that conforming to the schema can be accepted by the tool, the Schema is actually the specification of the input domain of the tool. The first step of testing is to generate conformed XML Instances from the defined XML Schema.

Then the test is executed by running the tool to map the derived XML Instances to the database. If all of the documents can be transformed correctly, the mapping tool passes

the testing; otherwise there are bugs in the tool.

The oracle in this testing usually is the tool itself, because when there is the error, most tools could throw an exception message or at least record the error in the log file. We can just check the output of the tools to get the result of testing.

In the next section we will present a case study of testing the XML-database mapping.

8.3 Case Study

In this section we will present a case study in which a tool for XML Document and XML database transformation is tested. At the beginning we will first introduce the concept and tools that are important for understanding the case study.

8.3.1 MySQL Database

MySQL [Sue02] [SM] is a relational database management system developed by the Swedish company MySQL AB. Because of its consistently fast performance, high reliability, and ease of use MySQL has become one of the the most popular open source databases. MySQL can run on different computer systems, such as Unix, Linux, and Windows. MySQL can also be written in many languages. Currently it is widely used for web applications.

The MySQL database system uses a client-server architecture. The server is the program that actually manipulates the database. The client programs communicate the intent of the user to the server by means of queries written in Structured Query Language (SQL) [DuB07b]. The client and server of the database may be installed in the different computers. Client programs can be written for many different purposes; each of them connects and interacts with the server, sending SQL queries to the server and receiving the query result.

MySQL has now become a subsidiary of Sun Microsystems [sun08]. The source code of the MySQL project is available under the terms of the GNU General Public License.

MySQL database does not have the native capabilities for supporting XML, but there are many languages for writing MySQL applications provide the XML support, and those languages connect XML with the relational databases. In the following section, we will introduce a mapping tool that parses XML format data, and saves it in a MySQL database.

8.3.2 XML-database Mapper(myXDM)

XML-database Mapper (myXDM) [myx06] is an open source Java based application that maps between XML and the database. It parses the data in XML format, and populates it into the database; it can also query the data from the database and transform the data back to the XML Document.

But not all XML Documents can be processed by myXDM; XML Documents that can be mapped to the MySQL database by myXDM must conform to a specific specifica-

tion defined by a schema. Initially the schema is written in DTD, and called "myXDM-xml2db", we show it in Figure 8.1.

```
<!ELEMENT XDM (Table+)>
<!ELEMENT Table (PrimaryKey,ForeignKey*,Record*)>
<!ATTLIST Table Name CDATA #REQUIRED
Action (insert|update) #IMPLIED
Commit (true|false) #IMPLIED
>
<!ELEMENT PrimaryKey (Generator)>
<!ATTLIST PrimaryKey Field CDATA #REQUIRED>
<!ELEMENT Generator (#PCDATA)>
<!ATTLIST Genera<!ELEMENT XDM (Table+)>
<!ELEMENT Table (PrimaryKey,ForeignKey*,Record*)>
<!ATTLIST Table Name CDATA #REQUIRED
Action (insert|update) #IMPLIED
Commit (true|false) #IMPLIED
>
<!ELEMENT PrimaryKey (Generator)>
<!ATTLIST PrimaryKey Field CDATA #REQUIRED>
<!ELEMENT Generator (#PCDATA)>
<!ATTLIST Generator Type (uuid|sequence|assigned|identity) #IMPLIED
Name CDATA #IMPLIED>
<!ELEMENT ForeignKey (#PCDATA)>
<!ATTLIST ForeignKey Field CDATA #REQUIRED
Ref CDATA #REQUIRED
Type CDATA #IMPLIED
>
<!ELEMENT Record (Field+,Table*)>
<!ATTLIST Record Commit (true|false) #IMPLIED>
<!ELEMENT Field (#PCDATA)>
<!ATTLIST Field Name CDATA #REQUIRED>
<!ATTLIST Field Type (blob|clob|date|int|long|double|float|date|time|timestamp) #IMPLIED>
<!ATTLIST Field Format CDATA #IMPLIED>tor Type (uuid|sequence|assigned|identity) #IMPLIED
Name CDATA #IMPLIED>
<!ELEMENT ForeignKey (#PCDATA)>
<!ATTLIST ForeignKey Field CDATA #REQUIRED
Ref CDATA #REQUIRED
Type CDATA #IMPLIED
>
<!ELEMENT Record (Field+,Table*)>
<!ATTLIST Record Commit (true|false) #IMPLIED>
<!ELEMENT Field (#PCDATA)>
<!ATTLIST Field Name CDATA #REQUIRED>
<!ATTLIST Field Type (blob|clob|int|long|double|float|date|time|timestamp) #IMPLIED>
<!ATTLIST Field Format CDATA #IMPLIED>
```

Figure 8.1: The DTD of myXDM-xml2db

As already presented in Chapter 5, TAXI supports only XML Schema, so we first need to transform the schema from a DTD to an XML Schema. There are a lot of tools that can convert a DTD file to an XML Schema. From them we have chosen choose the well-known tool XMLSpy to do the conversion, we have called the converted XML Schema "myXDM-xml2dbSchema". Figure 8.2 shows the converted XML Schema.

Below we introduce the elements in "myXDM-xml2dbSchema":

- <XDM> is the root element.
- <Table> contains the data that will be saved into the database. <Table> has three attributes:
  - "Name" is the table named in the database; it is required.

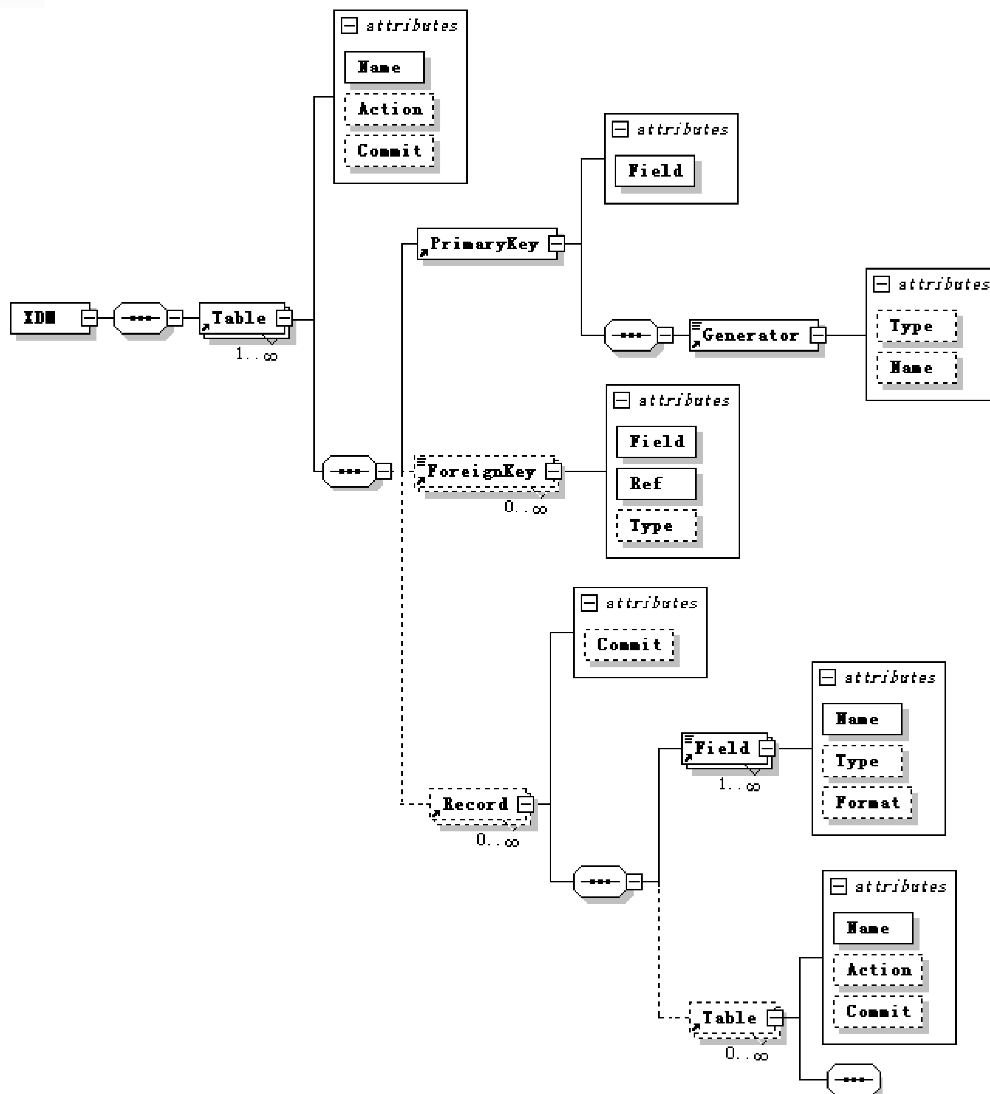

Figure 8.2: The XML Schema myXDM-xml2dbSchema

 – “Action” has two values “insert” and “update” that are defined in the schema. When the attribute “Action” is equal to “insert”, it means the data under the <Table> should be inserted into the database. When the value of “Action” is “update” it denotes the data under <Table> should update the data in the database. “Commit” has two values “true” and “false”. If its value is equal to “true” then the transaction will be committed after the data has been saved into the database, otherwise the transaction will not be commit.

- <PrimaryKey> element describes the primary key field. Its attribute “Field” is required and denotes the primary key field name of the database.

- <Generator> element denotes the method for generating the primary key value.

This method is defined by the value of the attribute “Type”. There are several values for “Type” defined by the schema, which are:

  - “uuid” which means the string should be generated by UUID algorithm and the length of the string should be 36 characters.
  - “sequence” which means the primary key generation method provided by the Oracle database should be used. In this case the name of this method must defined by the value of “Name” attribute.
  - “assigned” which means the primary key value is assigned by the node <Field>.
  - “identity” which means the value of primary key is assigned by the field characters in the database.

- <ForeignKey> defines the field name. There are three attributes of this element:
  - “Field” attribute defines the field name in child table.
  - “Ref” attribute is the field of primary table; the format should be “Table.Field”.
  - “Type” attribute defines the data type of the “Field”. MyXDM supports only some specific types, they are: *“int, long, float, double, date, time, datetime, BLOB, image, bytea, CLOB, text”*.
- <Record> element defines the record data that is inserted into the database. <Record> is a sequence element that has two children that are referred from the elements <Field> and <Table>, a record could include a table. It has an attribute “Commit” that denotes whether to commit the transaction after the record data insert into the database.
- <Field> defines the properties of the database. In XML Documents the data is stored as characters, whereas in the database there are many types of data. Different data types need specific methods for transformation. The attribute ‘Type” is used to denote the type of the data.

8.3.3 Testing myXDM Tool

In this case study, the tool myXDM is the system under testing. We will use TAXI to test if this tool can map XML Documents to a MySQL database correctly, and if it is strong enough for dealing with a variety of documents.

As already presented, the input of this tool is a group of XML Documents, which are conformed to the “myXDM-xml2dbSchema” (see in Figure 8.2), so the schema could be considered as the specification for the myXDM input domain. According to the specifications of myXDM, this tool can deal with all documents that conform to the “myXDM-xml2dbSchema”, so our task is trying to find as many as possible different conformed XML Documents, and check if all of them can be mapped correctly. TAXI is a very suitable tool for solving this problems. The process of testing is presented in Figure 8.3.

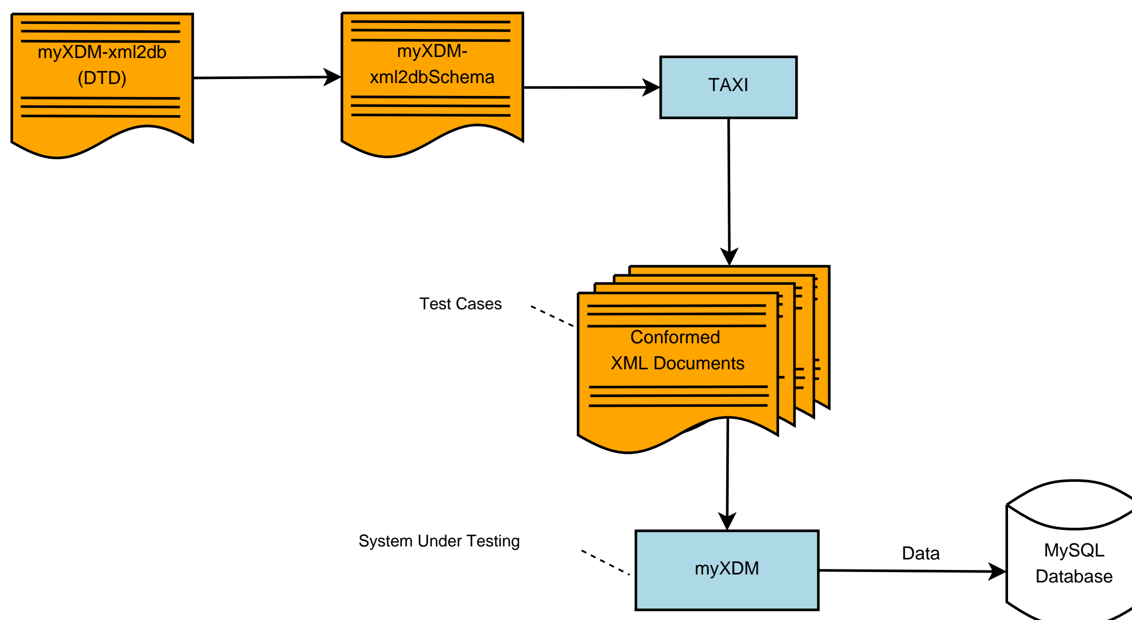

Figure 8.3: XML database mapping test

Before using TAXI to generate the XML Instances of the “myXDM-xml2dbSchema”, we must solve a problem. During the introduction of the “myXDM-xml2dbSchema”, it was easy to see that the values of the elements are actually related to the attribute values. For example in the element <ForeignKey>, the attribute “Type” defines the data type of “Field”. This means that with a specific value of an attribute, the related element must have a correspond value.

As we know, an XML Schema defines the structure and data types of a class of XML Documents. It dose not define the relationships among the elements and attributes with the different values. Therefore TAXI in its current version, also does not support the generation of instances according to the value of elements and attributes. In order to overcome this problem, we can use a set of XML Schemas to describe these relations. These XML Schemas are based on “myXDM-xml2dbSchema”, and are derived by the relationships of the values and elements according to the myXDM specification. With each particular value of the attribute, we set the correct values that correspond to the attribute value during XML Instance generation.

According to the specifications of myXDM, we found there are two elements that could influence other elements by the value of their attributes. They are the elements <Table> and <Generator>. In the following section we will describe this in detail.

Generate Schema From Element <Table>

Specifically, according to the myXDM specification, for the <Table> element, the attribute “Name” is required, the type is string, and its value can not affect other elements. The value of the attribute “Action” then is different. When it equals “insert”, then the element <Generator> in <PrimaryKey> will become optional. And when it equals “update” the value should be obtained from the element <Record>. So from here the

“myXDM-xml2dbSchema” schema could be divided into two schemas: in one schema the value of “Action” is always equals “insert”, and the element <Generator> is optional (let’s call it *schema-insert*). In another schema the value of “Action” equals “update”; when we generate an instance from this schema we must set only the same value for both the attribute “Field” of <PrimaryKey> and the element of <Field> in element <Record> (let’s call this schema *schema-update*).

Generate Schema From Element <Generator>
Another element that can affect the instance generation by attribute values is the element <Generator>. The value of the attribute “Type” affects the value of attribute “Name” in <Generator>. From both schemas we generated by <Table> element, the new schemas are derived according the value of <Generator>’s attribute “Type”, and the value of attribute “Field” in <PrimaryKey>. The value of “Type” could be: “uuid”, “sequence”. “assigned” and “Identity”. In the first derived schema, the value of “Type” equals “uuid” and the primary key value will be generated by myXDM automatically. In the second schema the value of “Type” always is equal to “sequence”, and it means the Primary key will be generated by myXDM, but with the methods defined in the Oracle database. When the “Type” value equals “sequence”, the attribute “Name” in <Generator> is required, and its value must equal the method name that already exists in the database. So when we generate instances from this schema, we must set the values of attribute “Name” as the names of methods for primary key generation defined by oracle, which are: “*increment, identity, sequence, hilo, seqhilo, uuid.hex, uuid.string, native, assigned*”.

In the third schema the value of “Type” equals “assigned”, and the primary key should be obtained from the <Field> in <Record>; this will be generated by the tool, so we do not need to set the values. In the fourth schema, the value of “Type” equals “identity”, and the primary key is controlled by the field of the primary key. It also will be generated by myXDM.

Because these schemas are generated both from the *schema-insert* and from the *schema-update*, we now have 2 * 4 = 8 schemas generated from “myXDM-xml2dbSchema”. These schemas cover the relationships of the values and elements specified in myXDM specification.

Next we can start instance generation from these eight generated XML Schemas. Before generation, we need to set the value of the attribute “Type” in <ForeignKey>, because in the specification, the value of “Type” can only be “int, long, float, double, date, time, datetime, BLOB, image, bytea, CLOB, text”, so we must predefine those values for “Type”. We have not predefined specific values for any other elements in the schemas, if they do not have the values that defined by the schema, then random values generated by TAXI will be assigned.

In order to cover as many instances as possible that can be accepted by myXDM, for all schemas we choose the test strategy “*Get all possible instances*”. From these 8 schemas we get $7 * 3^5 + 3^6 = 2430$ instances, because among the schemas, there are 7 that have five occurrence attributes. and one schema that has six occurrence attributes. In this schema the value of <Generator>’s attribute “Action” is “insert”, so in the element <PrimaryKey> the <Generator> element becomes optional; therefore this schema has

one more occurrence attribute.

There is one thing that should be mentioned; in the XML Schema, <Table> is a recursive element. In the current version of TAXI, the recursive element will be looped once in the derived instances. The occurrence attribute of the element in the first loop will not attend the combination of intermediate instance generations. In the loop, if the element is optional or could occur more than once, its occurrence will be fixed as one by TAXI.

Since myXDM is written in Java, it can be easily executed by Java comments. We have embedded the myXDM execution into the TAXI environment. When an XML Instance is generated, it will be immediately mapped to the MySQL database by myXDM automatically. The result of the execution will be written into the log of myXDM. So when we finish the XML Instances generation by TAXI, we actually have also finish the testing execution.

Initially we expected to generate several sets of instances with values for the occurrence. For the first set, we took the value “0” as the occurrence value assigned by TAXI, and then for the next sets, we increased the value by 10 each time until the value equaled to 100. In this way we hoped to get 11 sets and 11 * 2430 = 26730 XML Instances. However during the test execution, when the occurrence number is equal to 50, myXDM crashed, and threw the “out of memory error”. So we stopped the generation, and finally we ran 6 sets 14580 test cases. We show the result in Table 8.1.

Occurrence Value	Size of Instances	Error
0	1KB	0
10	1KB – 10KB	0
20	1KB – 170KB	0
30	1KB – 3MB	0
40	1KB – 6.5MB	0
50	1KB – 13MB	3

Table 8.1: Conducted experiments

By Testing, we can say myXDM can transform the data from XML to MySQL database correctly, but it not robust enough for dealing with the big XML documents.

The possibility of using other generation tools for doing XML-based Black-box Testing has already been presented in Chapter 7, and we will not repeat it here.

8.4 Summary

In this chapter we presented an application of TAXI for testing XML Document to database mapping tool. We first gave an overview of how to use TAXI for testing XML database population tools, then we give a case study. In the case study we attempted to test a XML database mapping tool, myXDM. We first analyzed the XML Schema used to define the input of the myXDM, and then used TAXI to generate a set of conformed XML

Instances as test cases. After running the test cases in myXDM, we found the tool can map XML Documents correctly, but it is not robust enough to deal with the big size XML documents.

Chapter 9

The Application Of TAXI to XML Benchmarks

This chapter presents the application of using TAXI to do XML benchmarking. We will explain why TAXI can be used as an XML benchmark tool, and illustrate a case study as well.

9.1 Introduction

The original meaning of the term "benchmark" is the chiseled horizontal marks in stone structures, that an angle-iron could be placed in, to form a "bench" for a leveling rod, ensuring the leveling rod could be accurately repositioned in the same place in the future. More recently benchmarks are used as reference point in land surveying; they are typically placed by a government agency or private survey firm, and many governments maintain a register of these marks so that the records are available to all. [wik]. Today, people have extended the meaning of the term "benchmark" to different areas; generally, it means the datum mark, the reference point, or the standard by which something can be measured.

In the computing field, the term "benchmark" means the act of assessing the performance of an object by running a set of programs or other operations; sometimes these programs may be standard tests. Benchmarks provide a method for evaluating the performance of software or hardware. The comparison could among different systems or the same system running in different environments.

Specifically, with the widespread usage of XML technologies, the requirements for XML Benchmarks is becoming more and more evident. XML-based systems need XML Benchmarks to assess their performance; a great number of methods for storing XML information have been developed in recent years, especially for database systems based on XML. The advantages and disadvantages of these systems need to be careful considered. Unfortunately, there is no industry standard for XML Benchmarks currently available. Along with the increase in requirements, there have been a lot of XML Benchmark tools developed, such as XBench [XBe], XMark [XMa], Toxgene [tox05], and so on.

For different applications, XML Documents for benchmarking need a range different of numbers, complexities, sizes, etc. As already presented in chapter 5, TAXI is a tool that can generate XML Instances from XML Schema automatically and systematically. The tool gives the possibility of configuring the number and size of derived XML Instances. Also, the values of the instance elements can be set to the user's expectations. Therefore, the size and the number of TAXI derived instances are varied and controllable. These properties make TAXI a suitable tool for XML benchmarking.

We will first present how to apply XML Benchmarking with TAXI in Section 9.2, then in Section 9.3 we will illustrate a case study using TAXI to assess the algorithms of XML Schema validation; finally, in Section 9.4 we review the advantages and disadvantages of using TAXI.

9.2 Benchmarks for XML-based Applications

As already presented, XML technologies are in widespread use. As XML has gained more and more momentum, and commercial products have began to appear on the market; also many XML-based applications are under development. The performance of these products has received a lot of attention. XML Benchmark tools then take on the challenge and develop features for evaluating the performance of these XML applications.

The work flow of XML Benchmarking in generally is:

- make a plan for the benchmark;
- derive a specific set of XML Documents as benchmark tests;
- run the tests on the target XML-based applications with the particular goals;
- record the relevant data for the benchmark goals; and
- analyze the record to evaluate the performance of the application.

For different goal of benchmark, the work flow may vary.

XML Benchmarks could be use to evaluate many XML-based applications, such as an XML database benchmark, an XSLT translation benchmark, an XML security benchmark and so on. The idea is to use TAXI for applying XML Benchmarks to XML-based applications. Since TAXI can produce fairly complex XML Documents systematically, it can capture the most common kinds of integrity constraints for popular XML benchmarks.

The environment for an XML-based application benchmark maybe not the same for the different products. But in general the aim of an XML Benchmark is to focus on the following points:

- How fast is each XML application;
- What hardware resources does each application need;

- What is the best for the intended purpose.

Of course, there are other particular factors for each specific application. The environment of an XML-based application benchmark should include at least three primary parts: the XML Document derivation, the benchmark execution and the benchmark report output.

XML Documents may be generated from different schemas. When they are based on an XML Schema, TAXI can serve as a good tool for the test derivation of the XML Benchmark. Tester should first design XML Schemas that are suitable for creating the benchmark by means of a test plan, then use TAXI to generate the XML Instances for the XML Schema. Depending on the purpose of benchmark, the derived XML Instances will vary. The tester may test for instances with a specific number, or instances of different size, or instances with particular elements or values; moreover invalid instances sometimes are required.

Next these derived XML Documents will be run on the under test XML-based applications. Meanwhile the data that can measure the application performance, such as time costs, resource costs, and other relevant items will be recorded. At the end of the benchmarking, the data obtained during the running of application will be compared and analyzed. The results of comparison and analysis is the basis for evaluating the performance of the application.

9.3 Case Study

In the case study, we will use TAXI to benchmarking the performance of two methodology of XML validators, which are based on the line-time translation algorithms.

9.3.1 Background

Before describing the case study, we must introduce first the fundamental concepts used in the case study. The first is Regular Expressions (REs) that are formalisms used in the domain to describe sets of strings of an alphabet. REs form the basis of most XML type languages such as DTD, XML Schema types, etc. Therefore the interweaving operator such as in XSD, Relax-NG would be a natural addition to the language of REs [GG08]. In this work, a liner-time translation algorithm is defined to check whether a word satisfies the resulting constraints. An algorithm that extends to the Type language could be express as below:

$T ::= \epsilon \mid a\,[m..n] \mid T + T \mid T \cdot T \mid T \,\&\, T$

In which instead of REs, the term type language with counting, disjunction, concatenation, and interleaving for strings over a finite alpha $\sum$. The ϵ represents an empty word, and an expression whose language is $\{\epsilon\}$. a [m..n] with $a \in \sum$ contains the words composed by j repetitions of a, with $m \leq j \leq n$.

The algorithm is then extended from the words to trees, specific to the XML Schemas. The details of the algorithm can be found in [GG08].

To understand the algorithms, we must first explain the extended DTDs (EDTDs) [WG06]. If we define a DTD is a tuple (Σ, d, s_d), where d is a function maps symbols in Σ to the elements, and $s_d \in \Sigma$ is the start symbol. The EDTDs can be expressed with a quintuple$(\Sigma, \Sigma', d, s, \mu)$, where Σ' is an alphabet of types, (Σ', d, s) is a DTD over Σ', and μ is a mapping from Σ' to Σ. A tree t satisfies an EDTD if t could be rewritten as $\mu(t')$, where t' must be valid of the DTD (Σ, d, s).

And the *single-type* EDTD (EDTDst) is an EDTD $(\Sigma, \Sigma', d, s, \mu)$ with the property that for every $a \in \Sigma'$, in the regular expression $d(a)$ no two types b^i and b^j with $i \neq j$ can occur.

An XML Schema could be modeled with the EDTDst, with $(\Sigma, \Delta, \tau, \mu, \rho)$, where Σ is a set of labels, Δ is a set of type-names, τ is a function mapping each type-name to a content-model, which is a type expressed on the alphabet Δ, μ is a function from Δ to Σ, and $\rho \in \Delta$ is the root type-name. By the constraint specification of *Element Declarations Consistent* μ must be injective when restricted to a specific content model [HST04]. Therefore it is possible to check the membership of the XML tree in an XML Schema. For example, a content-model T = $\tau(\beta)$, which β is a specific type-name. To check if $\langle a_1 \rangle x_1 \langle / a_1 \rangle ... \langle a_n \rangle x_n \langle / a_n \rangle$ satisfies T, it needs to retrieve the content model T_i = $\tau(\mu_\beta^{-1}(a_i))$ of each subelement, check that each x_i matches T_i, and check if the sequence $\omega = \mu_\beta^{-1}(a_1)...\mu_\beta^{-1}(a_n)$ matches T ($\mu_\beta^{-1}(_)$ is the inverse of μ, restricted to the type-names appearing in the content model of β.)

9.3.2 XML Schema Validation Benchmark

A case study of XML Benchmarking by TAXI will be presented in this section. [1] In the case study, TAXI is used to evaluate the performance of the algorithms for validating an XML Document against an XML Schema.

There are two algorithms of XML Schema validation that are developed based on the algorithm that we described in the background, called *lazy validation* and *eager validation*. The lazy and eager algorithms are both developed in Java; their input is in the XML Schema but written in EDTDst format. Therefor to apply both of the algorithms, the XML Schema needs first to translate them into the form of EDTDst.

The *lazy validation* gets its name because with this algorithm the validation will not be called until all the elements under validation are read; in other words, when the close tag of an element is reached, the validation will be carried out.

With the algorithm *eager validation*, when the start tag of an element is read, the validation is called, and starts the calculation of the residue along with the reading of the element content. When the close tag is reached, the validation of the element is finished.

In order to evaluate the performance of these algorithms, we have used the validation

[1] This case study is from a collaboration between the author and the work of Professor Giorgio Ghelli at the University of Pisa.

that has been included in the standard Java API Xerces [xer05] as a reference. The XML Schemas that are used to do the XML Benchmarking are shown in Figure 9.1. One is the schema of DBLP database, which describes the structures of the data that could be used in DBLP database. Another schema is the "expense-report" which describes the data structure of the data for people expense. These two schemas include both <sequence> and <choice> elements, and the size of the generated XML Documents are suitable for the requirements of the benchmarking.

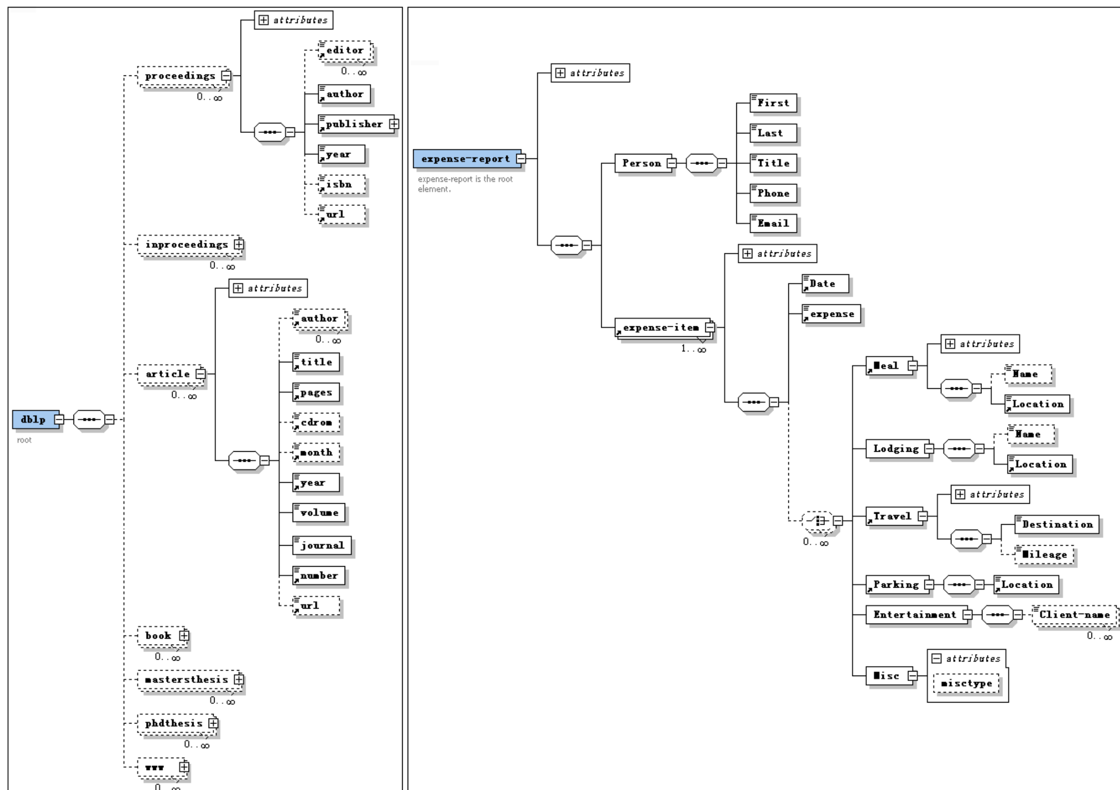

Figure 9.1: The schemas that are used for XML Schema Validation Benchmark

We need first to generate the XML Instances from the XML Schemas. The aim of the benchmark is to evaluate the time costs of the XML validators. The algorithms of "lazy" and "eager" are only focused on the validation of the XML structure, and ignore the correctness of the element values.

According to the aim of the benchmark, we get first the requirements for the derived XML Instances. First, the elements in the instances do not need to have particular values, so we can skip the step of value pre-define, and let the instances use the random values generated by XPT. The second requirement is the size of the documents. The benchmark needs various documents from very small to quite large, so that we can check the rising of time cost according to those instances. Also, the benchmark needs a large number of the instances.

As presented in Chapter 5, the derived instances of TAXI may not have the same size if

there are elements in the schema that have occurrence attributes, since those elements will occur at different times in the derived instances. In addition to this, there are two ways for getting different sizes of XML Instances using TAXI: one is by element selection, especially when there is <choice> element in the XML Schema. We can select specific subschema for instance generation, according to the <choice> child in the subschema, and the size of the document may be different. Another way is by setting the value of the element occurrences. When there is the occurrence valued with "unbounded", by default TAXI sets it at a default value, for example "3", but this could be changed by the user. With a bigger value for the occurrence, the derived instances will have a greater size.

In order to make the instance size distribute in different ranges, we set the occurrence value of schema "DBLP" and "expense-report" from 0 to 10000, as 0, 10, 100, 200, 500, 800, 1000, 2000, 3000, 4000, 5000, 6000, 70000, 8000, 9000, 10000, and run TAXI several times with different values of element occurrence.

Finally, we select 1040 XML Instances from the schema "DBLP" ranging from 1 KB to 300 MB, and 1000 instances from schema "expense-report" ranging from 1 KB to 5 GB. The details of these XML Instances are shown Figure 9.2.

DBLP.xsd		expense-report.xsd	
Instance size	Instance number	Instance size	Instance number
1KB~10KB	20	1KB~10KB	20
50KB~100KB	20	50KB~500KB	20
300KB~400KB	50	500KB~1MB	50
500KB~650KB	50	1MB~10MB	50
700KB~800KB	50	10MB~50MB	100
800KB~1MB	50	50MB~100MB	100
5MB~10MB	100	100MB~200MB	100
10MB~20MB	100	200MB~400MB	100
20MB~50MB	100	400MB~600MB	100
50MB~70MB	100	600MB~800MB	100
70MB~100MB	100	800MG~1GB	100
100MB~200MB	150	1GB~2GB	100
200MB~300MB	150	2GB~5GB	60

Figure 9.2: The size and number of the XML Instances

From this figure, we can see that size of the instances needs to increase uniformly. Since the value of the elements were not considered during the benchmarking, we did not populate the value files, so that all elements in the derived instances have random values generated by TAXI automatically.

After XML data generation, all these instances are taken as input to the "lazy valida-

tion", "eager validation" and to the validator within Xerces. After the time costs analysis, we get the result shown in Figure 9.3 and Figure 9.4. In thees figures, the horizontal axis represents the size of the XML Documents under the validation; the unit is "megabyte". The vertical axis denotes the time spent on the validation; the unit is "second". There are three lines with different colors in the figures. The red line is the timeline of "lazy validation", the blue line represents the timeline of "eager validation", and the green one denotes the result of Xerces.

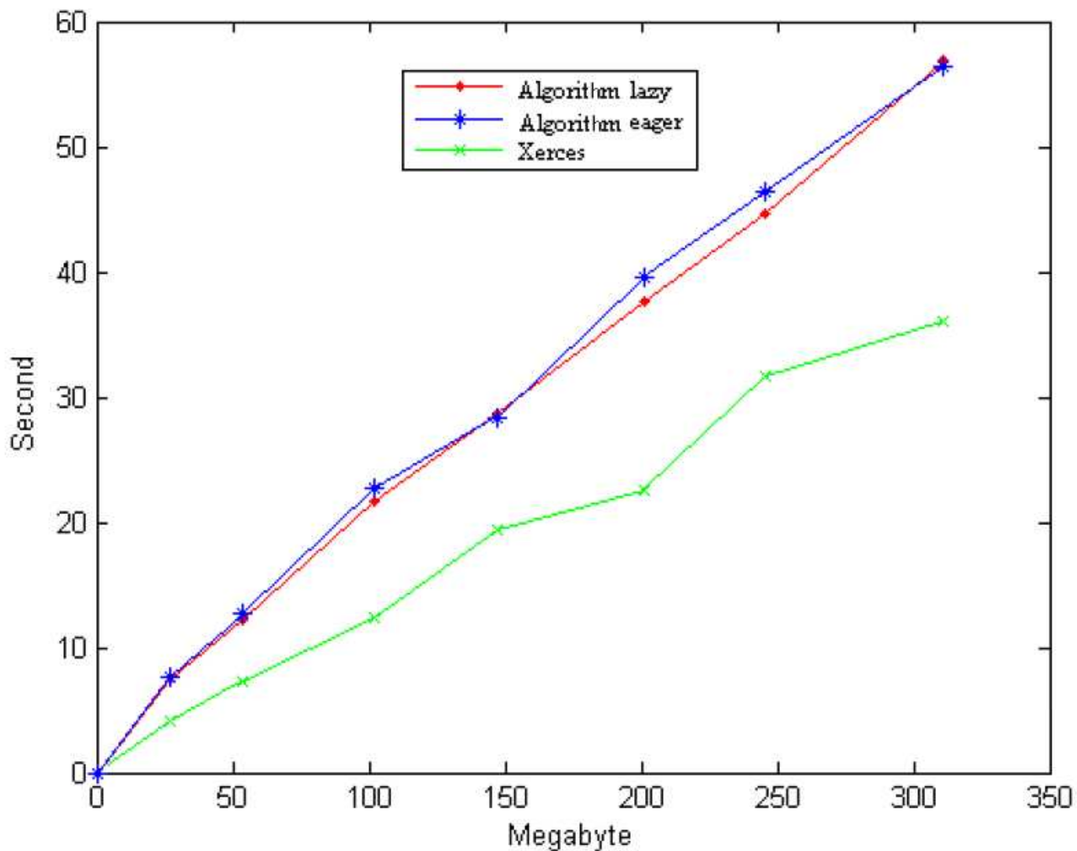

Figure 9.3: The graph of time measured for "DBLP" validation

After benchmarking and analysis of the performance of the algorithms, we can conclude that the algorithms of "lazy validation" and "eager validation" perform well, although they are more than three times slower than the validator within Xerces. Since these two algorithms are without any optimizations, while Xerces is a mature tool, this result could be accepted, and we believe that the optimistic liner-time algorithms will perform well for validating XML Documents.

In this case study, TAXI plays as an important role, since this benchmarking requires XML Schemas that are used in the real-word, and needs schemas containing different types of elements. XML benchmark tools such as XMark [XMa], XBench [XBe], X007 [SB01] can only accept specific schemas. Moreover TAXI provides a sufficient number

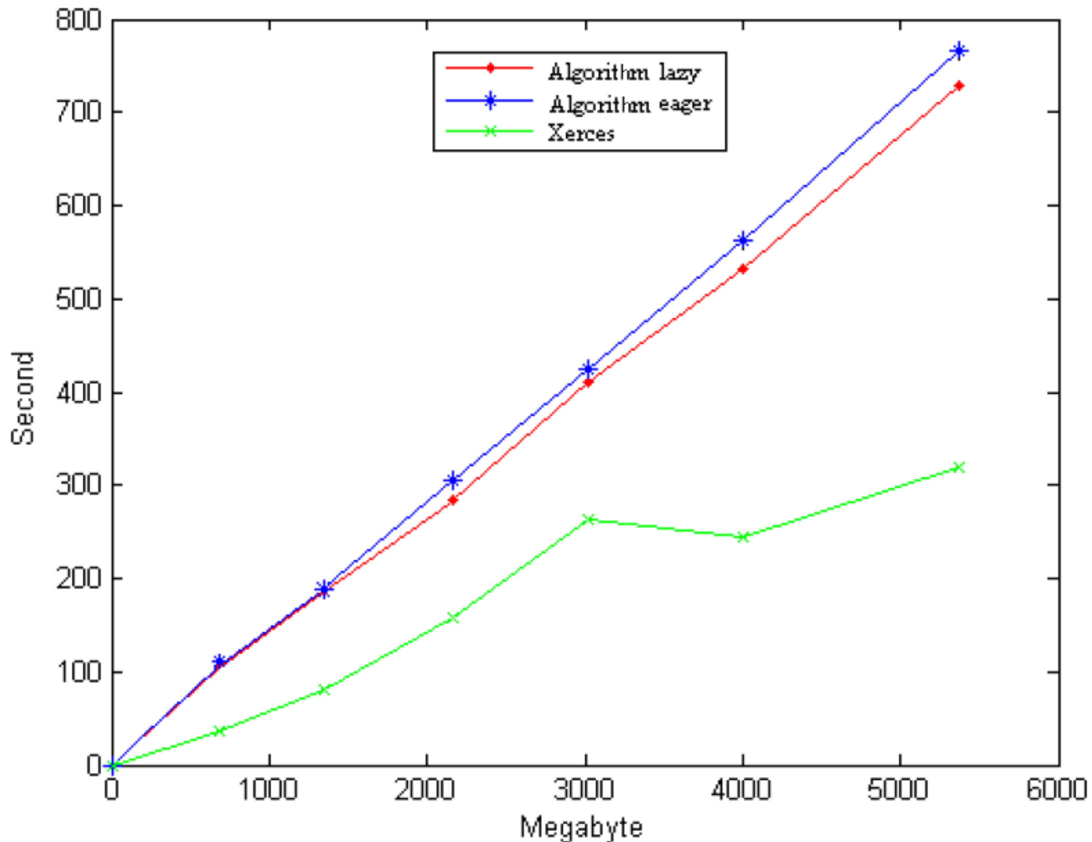

Figure 9.4: The graph of time measured for "expense-report" validation

and quality of XML Instances for benchmarking. The automatic mechanism avoids the costs of studying the schemas, and avoids mistakes caused by the lack of the experience.

9.4 Advantages and Disadvantages of TAXI In Meeting The Requirements of Benchmarks

In this section we will discuss the requirements of tools for XML benchmarks, and how the TAXI tool satisfies these requirements. We will describe our own criteria for benchmarks, but be aware that they are not a wildly accepted standards. Also we have found that some XML Benchmark tools do not tend to satisfy some of our criteria, but have their own features.

9.4.1 How TAXI Meets the Requirements of XML Benchmark

TAXI has some features that are really suited for service as an XML benchmark tool. The details of XML Document generation by TAXI have already been shown in Section 5; in this section we will not repeat them but just give a summary of the the TAXI features.

Various sizes of documents. First an XML Benchmark needs to include various sizes of documents. To evaluate a system's performance, the ability to handle different sizes of documents is an important aspect. A tool for doing XML Benchmarking should also have capability to generate various sizes of documents. As presented in Chapter 5, XML Documents derived by TAXI include different structures; each structure will lead to a different size. Moreover, TAXI gives the user the possibility of changing occurrences, when "maxOccurs" equals to "unbounded". The default value will be set as 3 by TAXI automatically, but it can be changed by the user. This means TAXI can generate very big of XML Documents. On the other hand, when the derived instance includes a combination of very small occurrence values, the derived XML Instance will be very small. So the smallest size of a TAXI derived document could be 1 KB. The upper boundary is unknown, the biggest document we generated is more than 10 GB. Therefore the size range of XML Documents derived by TAXI is really wide; this could perfectly meet the requirement of a benchmark.

Larger number of derived documents. The performance in handing a large number of documents is another important aspect as sometimes a large number needs to be evaluated during XML benchmarking. Therefore benchmark tools are required to have good capability for generating a sufficient number of documents. The number of XML Documents derived from TAXI does not depend on the structure of the XML Schema, but can be configured by the user, that the number of derived documents can be as many as required. The maximum we tried is to generate more than 1000000 instances for a schema. There is no upper limit to the number of documents that TAXI can derive.

The variability of the schemas and documents. An XML benchmark could be used for different applications, for instance an XML database, a business system, even a tool for an XML Benchmark, etc. These applications may need information in different formats. The variability of the schemas and documents sometimes becomes the critical property for benchmarking. However in many tools, the elements and attributes are fixed, such as in XMark [XMa], XPathMark [Fra05], X007[SB01], and so on. TAXI allows the user to use XML Schemas as needed and expected. This gives great flexibility to the benchmark; moreover it is not necessary to prepare instructions for the derivation when the schema is changed.

Element value configuration. With value population, it is possible to generate meaningful XML Documents. Using TAXI, the user can set the values for each element, or use random values that generate automatically. This gives more flexibility to the user, and makes the tool suitable for more objects of XML Benchmarking.

XML Document validation. XML Document validation against the schema can ensure the documents used for benchmarking are correct and conformed to the XML Schema. Otherwise they may give misleadings result. TAXI validate the derived XML Documents as soon as they are generated, and ensure their conformance.

9.4.2 Limitations of the TAXI tool for Benchmarking

As described before, TAXI has many advantages for doing XML benchmarking. Some features are really outstanding compared to other tools. As with anything else, the tool is not perfect; there are still some shortcomings that need to be improved.

The first is the control of document size. As presented in Chapter 5, the size of derived documents relies on two factors: the structure of the documents, and the occurrence values set by the users. However, it is difficult to know exactly how large the documents could be generated. Especially with a complex schema, size control is difficult to do. To solve this problem, we generate the XML Documents with different values of occurrence, experimenting with the relationship between the document size and the value of occurrence; we then documented the required size of the documents, according to the experience. Even then, it was difficult to generate precisely the required size of a document. It may be necessary to run TAXI several times to get a satisfactory size of documents.

Another limitation of TAXI is that instances can only extend in breadth, not in depth. As we know, XML Schema can go broad by the extension of occurrences, and extend deep by the recursive use of elements. While TAXI supports both of these, but it does not support the configuration of element recursions. When there is a recursive element, TAXI allows only one recursion, so that the derived instances cannot extend to different recursive levels. This limitation sometimes cannot satisfy the requirements of a specific benchmark.

A finally limitation is the time costs of the XML Document generation. Since now TAXI is in its initial version, the optimization of the system is not very adavanced. For the generation of small documents, the difference in time costs is not obvious, but the time to generate large documents is much longer. For example to generate a 5 GB XML Document, TAXI needs around 3 hours. It should become must shorter after system optimization.

In summary, TAXI can be used as a tool to do XML Benchmarking. It is new, and has many outstanding features for XML Document generation, and the flexibility of using schemas. But it still has limitations that could be improved in the future.

9.5 Summary

In this section we presented the second application of TAXI: XML benchmarking. The aim of this application is to evaluate the performance of XML-based applications. First we explained what and how to do XML Benchmarking, then we gave a case study, using TAXI to do an XML Schema validator benchmark. In the case study we evaluated the performance of three XML validators, using test cases generated from TAXI. After the case study we discussed why TAXI is suitable to be an XML Benchmark tool, and the properties of the instances generated from TAXI. Additionally we described also the limitations of the TAXI tool, then we conclude that although there are some limitations, TAXI still a powerful tool for using as an XML Benchmark.

Chapter 10

Conclusion and Future Work

In this Thesis we have presented the methodology for automatically testing XML-based applications, automated through the whole process of testing, from *Test Condition Identification*, *Test Case Design*, *Test Case Generation*, *Test Case Execution*, to *Test Result Analysis*.

Through in-depth study of the literature and consideration of the real requirements of developers, we have tried to find a solution that can be used in the real world, not only the theoretically on paper. Therefore we have put a lot of efforts into the implementation of the methodology, and the application of the tool. If a method or a tool is to be accepted for real projects by the industry, it must have good usability (that is easy to use) and be able to adapt itself well to the requirement of real environments within the industry. It must also have high efficiency, so that it can help to reduce the costs of project; this sometimes requires automation. The method presented in this thesis is also aimed towards these goals. We have made the tool very easy to understand and use, and create environments for testing which the maximizes automation. In the latter part of the thesis we have focused on the practical and real applications of the method, with three real case studies. It is easy to understand the significance and contributes of our research by case studies, and the comparison between our solutions with others. With the case study, we can see our methodology can be used in different application domains, such as XML-based Black-box Testing, XSLT transformation Testing and XML application Benchmarks and so on. Some test processes using our tool are totally automatic. Comparing with the existing tools for XML-based testing, our methodology is able to generate more systematic test cases without predefined instructions; it is adaptable for different applications, and it gives more flexible to the user.

Besides the contribution, we have gained also the information which will inform what we should continue doing in the future.

As presented in the previous Chapters of this Thesis, our solution and tool have the advantages, but there is still some work that needs to be done in the future. For the improvement of the methodology, we have several tasks.

First, we need to consider how to relate the value and the element during XML Instance generation. In our case studies and in a lot of real projects, this is a very common

and critical requirement. This actually is a limitation of XML Schema, but if we can not overcome this problem, the areas of our method application will be greatly restricted. Even if the user can find a some remedy to solve this problem, it still increases the costs, and reduces the usability of the method.

Second we should think about how to control the size of the generated instances. As presented in Chapter 9, our method could be used for doing XML benchmarking. The benchmark application asks various questions of XML Instances, not only about different structures, but also about different sizes. TAXI can generate different sizes of instances, but the size of the instances can not be evaluated before generation. Since TAXI can take any XML Schema for instance generation, it is difficult to evaluate the instances. However, it is possible to forecast size after the first generation, and give suggestions to the user, guiding him/her to get instances with a satisfactory size more easily.

With regard to the tool TAXI, currently it is still under development for improving its performance and functionalities. The version we used in this thesis is not the latest version of TAXI. In a newer version more functions have been added, such as XSLT transformation. And the new version of TAXI does not use the eXist database for storing values. In the future, there are still some aspects of the tool could be improved.

Currently, for different applications of testing, we need to customize TAXI to create an automated testing environment. In the future we hope to make TAXI as an integrated tool, which gives the user the possibility of customizing the tool. For tests that are already defined, TAXI provides predefined automatic test environments; the user does not need to know the details of the tool, but just needs to choose the type of testing he/she wants to do.

We will improve the performance and the usability of TAXI. Currently TAXI does not generates instances very efficiently. If the Schema is complicated or the derived instance is large, it takes a long time to generate. We need to optimize the arithmetic of XML Schema analysis and instance building, to make the tool more adopted to the requirements of industry applications. Also we need to improve the tool and rise it usability to adapt various of applications. Now for different testing application, we need to customize TAXI. In the future, TAXI should be an integrated tool that can be used for mulriple testing domains.

Until now, for different applications, TAXI use third-party software as oracle to analyse the result of the testing. In the future, we would like to have the specific oracle for TAXI. Some of the oracle could have more analysis function, for example for XSLT Testing, the oracle could include mutation testing on XSLT stylesheet transformation testing. Or we would like to add more method such as some White-box Testing methods to analyse and measure the quality of the derived test cases, and make the test suite more reliable.

Finally, the most important task for our work in the future is to spread the application of our methodology. In this Thesis we have already presented three applications. Compared with the applications of XML technology, these applications are only the tip of the iceberg. We believe there are more potential applications waiting to be found.

Bibliography

[ABC03] Roberto Zicari Akmal B. Chaudhri, Awais Rashid. *XML Data Management Native XML and XML-Enabled Database Systems*. Addison-Wesley, 2003.

[AM07] Darko Marinov Sarfraz Khurshid Aleksandar Milicevic, Sasa Misailovic. Korat: A tool for generating structurally complex test inputs. In *ICSE 2007*, pages 771–774, 2007.

[AP97] M. A. Vouk Amit Paradkar, K.C. Tai. Specification-based testing using cause-effect graphs. *Annals of Software Engineering archive*, 4(133157), 1997.

[AWW96] Robert L. Probert Alan W. Williams. A practical strategy for testing pair-wise coverage of network inter-faces. In *Proceedings of the 7th International Symposium on Software Reliability Engineering (ISSRE96)*, New York, USA, October 1996.

[BdR04] Utsav Boobna and Michel de Rougemont. Correctors for xml data. In *International XML Database Symposium 2004*, pages 97-111, Toronto, Canada, 2004.

[Bei90a] Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold Co., New York, NY, USA, 2nd edition edition, 1990.

[Bei90b] Boris Beizer. *Software Testing Techniques*. International Thomson Computer Press, June 1990.

[Bei95] Boris Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. Wiley, May 1995.

[Ber04] Antonia Bertolino. *Guide to the Software Engineering Body of Knowledge*, chapter SWEBOK: SOFTWARE TESTING. Joint IEEE ACM Software Engineering Cordination Committee., On line at http://www.swebok.org, 2004.

[Ber07] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *ICSE 2007*, pages 85-103, Minneapolis, USA, May 2007.

[BM01] J. Berlin and M. Motro. Autoplex: automated discovery of content for virtual databases. In *Proc 9th Int Conf OnCooperative Information Systems (CoopIS)*, volume 2172, pages 108-122, Berlin Heidelberg NewYork, 2001. Springer. Lecture Notes in Computer Science.

[Bur06] Donald K. Burleson. *Oracle Tuning The Definitive Reference*. Rampant Techpress, 2006.

[CK99] Hung Quoc Nguyen Cem Kaner, Jack Falk. *Testing Computer Software*. Wiley computer publishing, second edition edition, 1999.

[Coe08] David J. Coe. A review of boundary value analysis techniques. *STSC CrossTalk*, April 2008.

[Coh06] Sara Cohen. Count-constraints for generating xml. In *NGITS*, pages 153-164, Kibbutz Shefayim, Israel, 2006.

[Coh08] Sara Cohen. Generating xml structure using examples and constraints. *PVLDB*, 1(1):490-501, 2008.

[Con03] Software diagnostics and conformance testing division: Web technologies. http://xw2k.sdct.itl.nist.gov/brady/xml/index.asp, 2003.

[Cor08] Stylus Studio Corporate. Stylus studio powerful xml integrated development environment. http://www.stylusstudio.com/xml_product_index.html, 2008.

[Day01] Igor Dayen. Storing xml in relational databases. http://www.xml.com/pub/a/2001/06/20/databases.html?page=2#ibm, June 2001.

[dcS04] dc-Schema. http://www.loc.gov/standards/sru/resources/dc-schema.xsd, April 2004.

[DDH01] AH. Doan, P. Domingos, and A. Halevy. Reconciling schemas of disparate data sources: a machine-learning approach. In *Proc ACM SIGMOD Conf*, pages 509-520, 2001.

[DDL00] AH. Doan, P. Domingos, and A. Levy. Learning source descriptions for data integration. In *Proc Web DB Workshop*, pages 81-92, 2000.

[DK07] Jaeyoung Choi Chae-Woo Yoo Dongkyu Kwack, Yongyun Cho. A xml-based testing tool for embedded softwares. In *Multimedia and Ubiquitous Engineering, 2007. MUE '07. International Conference on*, pages 431-438, Seoul, 2007.

[dlR07] Jose Tuya Javier de la Riva, Claudio Garcia-Fanjul. Systematic design of test case for xpath queries using partition-based techniques. In *Latin America Transactions, IEEE (Revista IEEE America Latina)*, volume 5, pages 259-264, July 2007.

[DM03] Dumitru Daniliuc Sarfraz Khurshid Martin Rinard Darko Marinov, Alexandr Andoni. An evaluation of exhaustive testing for data structures. Technical report, MIT Computer Science and Artificial Intelligence Laboratory, 2003.

[DMC97] M. L. Fredman G. C. Patton D. M. Cohen, S. R. Dalal. The aetg system: An approach to testing based on combinatorial design. *IEEE Transactions On Software Engineering*, 23(7), July 1997.

[DTD96] DTD. http://www.w3.org/TR/2000/CRSVG20001102/svgdtd.html, 1996.

[dub07a] The dublin core metadata initiative dublin core. http://dublincore.org/, 2007.

[DuB07b] Paul DuBois. *MySQL Cookbook*. O'Reilly Media, second edition edition, January 2007.

[dV04] Eric Van der Vlist. *RELAX NG*. O'Reilly, 2004.

[easnd] EasycheXML. http://www.stonebroom.com/xmlcheck.htm, nd.

[ED00] John Paul Elfriede Dustin, Jeff Rashka. *Automated Software Testing : Introduction, management and performance*. Addison-Wesley, 2000.

[EJB03] EJBSourcegenerator. http://ejbgen.sourceforge.net/, 2003.

[Elm97] W. R. Elmendorf. Cause-effect graphs in functional testing. Technical Report TR-00.2487, IBM Systems Development Division, 1997.

[exi] exist. http://exist-db.org/.

[Fra05] M. Franceschet. Xpathmark-an xpath benchmark for xmark generated data. In *International XML Database Symposium(XSYM)*, pages 129-143, 2005.

[GG08] Carlo Sartiani Giorgio Ghelli, Dario Colazzo. Linear time membership for a class of xml types with interleaving and counting. In *In Proceedings of PLAN-X 2008 Programming Language Techniques for XML*, San Francisco, California, January 2008.

[GW97] R. Goldman and J. Widom. Dataguides: enabling query formulation and optimization in semistructured databases. In *Proc 23th Int Conf On Very Large Data Bases*, pages 436-445, 1997.

[Ham94] Richard Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970/–978. Wiley, 1994.

[Har99] Elliotte Rusty Harold. *XML Bible*. IDG Books Worldwide, Inc., 1999.

[HHS04] Langer Hagen, Lungen Harald, and Bayerl Petra Saskia. Text type structure and logical document structure. In Bonnie Webber and Donna K. Byron, editors, *ACL 2004 Workshop on Discourse Annotation*, pages 49-56, Barcelona, Spain, July 2004. Association for Computational Linguistics.

[HST04] Murray Maloney Noah Mendelsohn Henry S. Thompson, David Beech. Xml schema part 1: Structures second edition. Technical report, World Wide Web Consortium, October 2004.

[IEE90] Ieee standard glossary of software engineering terminology. In *IEEE Std 610.121990*, 1990.

[Jav03] JAVA XMLbindlets. http://wwws.sun.com/software/xml/developers /instancegenerator/index.html, 2003.

[Jav06] The JAVA XML validation api. http://java.sun.com/developer/technicalArticles/ xml/validationxpath/, 2006.

[JC00] Jane Xu Josephine Cheng. Ibm db2 xml extender, an end-to-end solution for storing and retrieving xml documents. In *ICDE'00 Conference*, San Diego, USA, 2000. IEEE.

[JO06] Lisa Liu Jeff Offutt, Paul Ammann. Mutation testing implements grammar-based testing. In *Proc. Second Workshop Mutation Analysis*, November 06.

[Kab08] Barbar Kablan. Automatic generator of xml documents editors based on attributed grammars. In *CSTST '08: Proceedings of the 5th international conference on Soft computing as transdisciplinary science and technology*, pages 166-172, New York, USA, 2008. ACM.

[KC00] John Hughes Koen Claessen. Quickcheck: a lightweight tool for random testing of haskell programs. In *In International Conference on Functional Programming*, pages 268/–279, Montreal, Canada, September 2000. Association for Computing Machinery.

[Läm01] Ralf Lämmel. Grammar testing. In *Proc. of Fundamental Approaches to Software Engineering (FASE) 2001*, volume 2029 of *LNCS*, pages 201-216. Springer-Verlag, 2001.

[LC94] W. Li and C. Clifton. Semanticinte gration in heterogeneous databases using neural networks. In *Proc 20th Int Conf On Very Large Data Bases*, pages 1-12, 1994.

[LC00] W. Li and C. Clifton. *SemInt: a tool for identifying attribute correspondences in heterogeneous databases using neural network*, pages 49-84. Data Knowl Eng 33(1), 2000.

[LCL00] W. Li, C. Clifton, and S. Liu. *Database Integration Using Neural Network: Implementation and Experiences*, pages 73-86. Knowl Inf Syst 2(1), 2000.

[LCN06] The library of congress network development. http://www.loc.gov/index.html, 2006.

[Lev99] Alon Levy. More on data management for xml. http://www.cs.washington.edu/homes/alon/widom response.html, May 1999. University of Washington.

[LM05] Jian Bing Li and James Miller. *Testing the Semantics of W3C XML Schema*, pages 443 - 448. COMPSAC 2005, 2005.

[mar01] Marc to dublin core crosswalk. http://www.loc.gov/marc/marc2dc.html, February 2001.

[MAR05] Xslt of marc21slim2srwdc. http://www.loc.gov/standards/marcxml/xslt/ MARC21slim2SRWDC.xsl, 2005.

[mar06] Marc standard office. http://www.loc.gov/standards/marcxml/, July 2006.

[MAR07a] Marc 21 bibliographic of “controlfield”. http://www.loc.gov/marc/bibliographic/bd00x.html, 2007.

[MAR07b] Marc 21 bibliographic of “datafield”. http://www.loc.gov/marc/bibliographic/ bdheading.html, 2007.

[MAR07c] Marc 21 bibliographic of “leader”. http://www.loc.gov/marc/bibliographic /bdleader.html, 2007.

[MAR07d] Marc standards office. http://www.loc.gov/marc/, 2007.

[Mei02] Wolfgang Meier. exist: An open source native xml database. In *NODe 2002 Web-and Database-Related Workshops*, Springer LNCS Series, 2593, pages 169-183, Erfurt, Germany, October 2002. Springer-Verlag.

[MF94] Dorothy Graham Mark Fewster. *Software Testing Automation Effective use of test execution tools*. ACM Press, New York, 1994.

[MJC93] J. F. Naughton M. J. Carey, D. J. DeWitt. The oo7 benchmark. In *ACM SIGMOD Int. Conf. On Management of Data*, pages 12-21, Washington, United States, 1993.

[MLL] Zo Lacroix Mong-Li Lee, Ying Guang Li. The xoo7 benchmark. In *Proceedings of the VLDB 2002 Workshop EEXTT and CAiSE 2002 Workshop DTWeb on Efficiency and Effectiveness of XML Tools and Techniques and Data Integration over the Web-Revised Papers table of contents*, pages 146-147. Springer-Verlag London, UK.

[Mye04] Glenford J. Myers. *The art of software testing*. John Wiley & Sons, Inc., 2nd edition edition, 2004.

[myx06] Xml-database mapper (myxdm). http://sourceforge.net/projects/myxdm/, February 2006.

[OB88] T.J. Ostrand and M.J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of ACM*, 31(6), 1988.

[Obj04] Objectmodelgenerator. http://sourceforge.net/projects/omgen, 2004.

[Ora] Oracle. Xml-sql utility (xsu). http://www.oracle.com/technology/tech/xml/xdk/doc/production/plsql/doc/plsql/xsu/xsu_user.

[OX04] Jeff Offutt and Wuzhi Xu. Generating test cases for web services using data perturbation workshop on testing, analysis and verification of web services. Boston Mass, July 2004.

[oxy07] oxygen xml editor. http://www.oxygenxml.com/, 2007.

[Pat05] Ron Patton. *Software Testing*. Sams Publishing, July 2005.

[RB01] Erhard Rahm and Philip A. Bernstein. Survey of approaches to automatic schema matching. VLDB *Journal: Very Large Data Bases*, 10:334-350, 2001.

[Rei97] S.C Reid. An empirical analysis of equivalence partitioning, boundary valueanalysis and random testing. In *Software Metrics Symposium, 1997. Proceedings., Fourth International*, November 1997.

[RML05] Miguel A. Sanz-Bobi Rocío Martínez-López. *Lecture Notes in Computer Science*, chapter Divisible Rough Sets Based on Self-organizing Maps, pages 708-713. Springer Berlin / Heidelberg, 2005.

[RTTnd] RTTS: Proven XML testing strategy. http://www.rttsweb.com/services/index.cfm, nd.

[Rub06] Chris Rubsamen. Ibm transforms database market with introduction of db2. http://www-03.ibm.com/press/us/en/pressrelease/19781.wss, 2006.

[SB01] Z Lacroix-M Lee Y Li U Nambiar B Wadhwa S Bressan, G Dobbie. X007: Applying 007 benchmark to xml query processing tool. In *Proceedings of CIKM*, 2001.

[SB04] Tobias Nipkow Stefan Berghofer. Random testing in isabelle/hol. In *2nd International Conference on Software Engineering and Formal Methods*, pages 230/–239, Beijing, China, September 2004. IEEE Computer Society.

[SCL01] Jeff Offutt Suet Chun Lee. Generating test cases for xml-based web component interactions using mutation analysis. In *In Proceedings of the 12th International Symposium on Software Reliability Engineering*, pages 200-209, Hong Kong China, November 2001. IEEE Computer Society Press.

[Sin00] Russell Sinclair. *From Access to SQL Server*. Apres, 2000.

[SM] Inc. Sun Microsystems. Mysql. http://www.mysql.com.

[SQC01] Xml schema quality checker. http://www.alphaworks.ibm.com/tech/xmlsqc, 2001.

[Sri] Rahul Srivastava. Xml schema: Understanding structures. http://www.oracle.com/technology/pub/articles/srivastava_structures.html.

[ST02] M.H. Shafazand and A M. Tjoa. *A Levelized Schema Extraction for XML Document Using User-Defined Graphs*, pages 434-441. Number LNCS 2510. EurAsia-ICT 2002, 2002.

[STH+99] Jayavel Shanmugasundaran, Kristin Tufte, Gang He, Chun Zhang, David DeWit, and Jeffrey Naughton. Relational databases for querying xml documents: Limitations and opportunities. In *Proceedings of the 25th VLDB Conference*, 1999.

[Sue02] Steve Suehring. *MySQL-Bible*. Wiley Publishing, Inc., 2002.

[Sun03] SUN XML instance generator. http://wwws.sun.com/software/xml/developers/instancegenerator/index.html, 2003.

[sun08] Sun microsystems announces completion of mysql acquisition. Sun Microsystems Press release, February 2008.

[TB01] E. Rahm T. Böhme. Xmach-1: A benchmark for xml data management. In *Proceedings of German database conference BTW2001*, pages 264-273, 2001.

[TBSK03] Khoo Boon Tian, Sourav S. Bhowmick, and Madria Sanjay Kumar. *VACXENE: A User-Friendly Visual Synthetic XML Generator*. Object-Oriented and Entity-Relationship Modelling, 2003.

[THCS01] Ronald L. Rivst Thomas H. Cormen, Charies E. Leiserson and Clifford Stein. *Introduction to Algorithms*. The MIT Press, second edition edition, 2001.

[tox05] Toxgene - the tox xml data generator. http://www.cs.toronto.edu/tox/toxgene/index.html, February 2005.

[val04] Java XML Validation. http://java.sun.com/j2se/1.5.0/docs/api /javax/xml/validation/package-summary.html, 2004.

[vdV02] Eric van der Vlist. *XML Schema*. O'Reilly, first edition edition, June 2002.

[W3C01] W3C Validator for xml schema. http://www.w3.org/2001/03/webdata/xsv, 2001.

[W3C05a] Extensible Markup Language (xml) conformance test suites. http://www.w3.org/XML/Test/, 2005.

[W3C05b] W3C World Wide Web Consortium. http://www.w3.org, 2005.

[W3S05] XML Schema reference. http://www.w3schools.com/schema/schema_elements_ref.asp, 2005.

[web05] Marc 21 concise bibliographic: Leader and directory. http://www.loc.gov/marc/bibliographic/ecbdldrd.html#mrc blea, 2005.

[WG06] Frank Neven Wouter Gelade, Wim Martens. Optimizing schema languages for xml: Numerical constraints and interleaving. *LECTURE NOTES IN COMPUTER SCIENCE*, (4353):269-283, 2006.

[Wid99] Jennifer Widom. *Data Management for XML*, volume 22(3), pages 44-52. IEEE Data Engineering Bulletin, Special Issue on XML, September 1999. Working Document, initial draft appeared April 1999.

[wik] Benchmark (surveying). http://en.wikipedia.org/wiki/Benchmark_(surveying).

[WYW00] Q. Wang, J. Yu, and K. Wong. Approximate graph schema extraction for semi-structured data. In *Proc Extending DataBase Technologies, Lecture Notes in Computer Science*, volume 1777, pages 302-316, Berlin Heidelberg NewYork, 2000. Springer.

[XBe] Xbench. http://www.xbench.com/.

[xer05] Xerces. http://xerces.apache.org/xerces-j/, 2005.

[XMa] Xmark. http://monetdb.cwi.nl/xml/.

[XML96] W3CXML. http://www.w3.org/XML/, 1996.

[XML98] W3C XMLSchema. http://www.w3.org/XML/Schema, 1998.

[XML99] XML Generator. http://www.alphaworks.ibm.com/tech/xmlgenerator, 1999.

[XML02] XMLTester. http://www.xmltester.org/_html_out/main/index.html, 2002.

[XML03] XMLUnit-junit and nunit testing for xml. http://xmlunit.sourceforge.net/, 2003.

[XML04] XMLXIG. http://sourceforge.net/projects/xmlxig, 2004.

[XML05a] XML Validator. http://www.elcel.com/products/xmlvalid.html, 2005.

[XML05b] XMLSpy. http://www.altova.com/products _ide.html, 2005.

[xmlnda] XML Judge. http://www.topologi.com/products/utilities/xmljudge.html, nd.

[xmlndb] XMLBuddy. http://xmlbuddy.com/2.0/index.php, nd.

[XOM02] XOMA. http://sourceforge.net/projects/xoma, 2002.

[XSL99] Xsl transformations (xslt). http://www.w3.org/TR/xslt, November 1999.

[XTY07] Benoit Eynard Xiu-Tian Yan, William J. Ion. *RoHS Compliance Declaration Based on RCP and XML Database*. Springer London, July 2007.

[YL02] K. C. Tai Yu Lei. A test generation strategy for pairwise testing. *IEEE Transactions on Software Engineering archive*, 28:109-111, January 2002.