國 立 台 灣 大 學

資 訊 工 程 學 研 究 所

碩 士 學 位 論 文

指導教授：傅立成 博士

全球資訊網資訊整合
可程式化包覆程式的
設計與實作

Design and Implementation of a
Configurable Wrapper for
Web Information Integration

研 究 生：黃宏軒 撰

學　　號：R87526019

中 華 民 國 八 十 九 年 七 月

誌謝

在進入研二下學期之後，每次遇到友人、學長姊時總是會被問道「論文開時寫了沒？」，而自己也只能苦笑著回答「還沒耶」。時間一晃而過，如今論文即將付梓，很慶幸自己能夠得到許多人的幫忙與支持，才得以在短時間內將論文完成。

這篇論文的完成首先必須感謝三位老師。中研院許鈞南老師一年來毫不吝惜的指導，在系統發展的過程中跟我討論及解答各種問題，在最後論文寫作期間，百忙之中還花費許多時間逐字訂正我充滿文法錯誤的稚拙英文及提供許多寶貴的意見，即使在週日還專程到實驗室指導我論文的寫作，非常感謝許老師的大力幫忙，論文才能夠及時地完成。也很感謝許永真老師在我研一一年的指導，口試的時候還特別為我變更回國行程，以及在口試的時候提供的許多建議，使得我能夠將原本考慮不周的部分加以補充，老師嚴謹的研究態度也帶給我許多的啟發。以及傅立成老師在我需要幫忙的時候都能夠毫不遲疑地伸出援手，也幫忙我解決了許多問題。

在系統發展的過程中，還必須感謝中研院兩位學弟的幫忙。翊嵐幫我寫作的程式節省了我許多的時間，與我的討論中提出的許多很有創意的點子也給了我莫大的助益，多次一起熬夜寫程式也成為珍貴的回憶。智海陪我渡過最忙碌的一段日子，連續幾個月每天中午、晚上一起用餐的中研院餐廳也成為充滿回憶的地方。在實驗進行的時候更幫了我很多忙，口試前還徹夜未眠地幫我準備系統展示。還有感謝建智即時地提供資料擷取程式的支援與系統發展的建議，隔著網路遠從交大與我一起除錯程式直到深夜兩點半也是一段難得的經歷。此外，台大的恩娥、漢申、婉容、光龍、士睿等學長姊及叡立對於口試的投影片提供了許多有用的建議，口試的時候也幫了我許多忙。

最後是謝謝父母與姊姊們一直以來精神上的支持、鼓勵與物質上的支援，以及對於我壓力下任性行為的容忍與體諒。這一篇論文謹獻給以上各位以及其他曾經幫助過我的人，謝謝。

摘要

在全球資訊網的環境中，如果可以將來自各個網站的資訊加以整合，這些資訊將可以更有效地被利用。資訊整合系統藉著系統本體與資料源中間的一層稱為包覆程式(wrapper)的軟體來存取資料源。為了可以包容其所連接的各個資料源之間的差異性，這些包覆程式經常是根據個別資料源的特性而以人工撰寫而成。然而，由於全球資訊網的互動性、不可靠性、及 HTML 文件的鬆散定義和結構，以人工為了單一網站而特別撰寫其包覆程式是一個很容易出錯並且花費大量時間的工作。本論文提供一套方法，使得網站的包覆程式可以被快速地建構。這些包覆程式不僅可以用在資訊整合系統中，並且也可以被用在其他需要自動化網站存取的應用系統之中。

我們定義了一套對於一組網頁、這些網頁上的資料、資料的位置、擷取方法以及資料組合方式的描述語言。並且實做了這個描述語言的解譯/執行器來實際執行以這個語言撰寫的程式。這一個執行器可以使幾乎所有種類的網站瀏覽動作自動化，並且可以處理動態產生的 HTTP 要求的參數，cookie，以及許多不合乎標準語法的 HTML 文件。我們並且設計和執行了一系列的實驗來驗證我們所採行的方法的可行性。實驗結果也顯示我們的系統可以不分領域且有效地快速建構全球資訊網網站包覆程式。

本論文的組織如下：第一章中對於資訊整合系統作一簡單地介紹，並且說明了全球資訊網中資料存取所會遭遇到的問題。第二章詳細地說明我們所定義的語言 WNDL 以及其如何使得網站的存取能夠自動化。第三章說明 WNDL 程式執行的過程以及整套系統實作的方法。第四章說明我們為了評估系統所做的實驗。在第五章中簡單地介紹了與本論文相關的研究。最後，我們在第六章中討論我們的系統目前還不能解決的問題以及未來可能的研究方向。

National Taiwan University
Department of Computer Science
and Information Engineering

M.S. Thesis

Design and Implementation of a Configurable Wrapper for Web Information Integration

Author: *Hung-Hsuan Huang*
Supervisor: *Li-Chen Fu*

July 2000

Acknowledgements

First at all, I wish to thank three professors. My advisor, Dr. Chun-Nan Hsu helped me a lot during the whole process of system design. He also spent plenty of valuable time to correct and refine my poorly written thesis draft. Prof. Yung-Jen Hsu taught me the conscientious attitude for being a graduate student and gave me many useful comments. Prof. Li-Chen Fu always gives me a hand immediately without any further consideration.

Particular thanks are owed to my colleagues at the Adaptive Internet Intelligent Agents Lab. of IIS, Academia Sinica. My project team members, Elan Hung and Chien-Chi Chang made a great contribution to system development while Harianto Siek helped me to conduct the experiments.

Then there are my colleagues at the Intelligent Mobile Robot Lab. of CSIE, National Taiwan University. They gave me many helpful suggestions and ideas on the slides and representation of my oral examination: Euna Jeong, Han-Shen Huang, Wan-Rong Jih, Kong-Lung Lin, Shih-Jui Lin, and Ray-Li Chen.

Finally I am grateful to my parents and sisters, they always support me and forgive my willful behaviors.

Contents

List of Figures iv

List of Tables v

List of Codes vi

List of Algorithms vii

Abstract 1

Chapter 1 Introduction 3

1.1 Information Integration………………………………………………………. 3

1.2 Web Information Integration…………………………………………………. 5

1.2.1 Web Navigation with HTML Documents………………………….. 5

1.2.2 Web Site Navigation Problem……………………………………… 7

1.2.3 Data Extraction Problem…………………………………………… 9

1.2.5 Our Approach………… 10

1.3 Organization of this Thesis………… 13

Chapter 2 Web Navigation Description Language (WNDL) 15

2.1 Preliminary………… 15

2.1.1 A Brief Introduction to XML………… 16

2.1.2 Term Definitions………… 18

2.1.3 Organization of WNDL………… 19

2.2 Data Web Map (DWM)………… 20

2.2.1 Data Web Map Node………… 21

2.2.2 Data Web Map Edge………… 23

2.3 Navigation Program………… 26

2.3.1 WNDL Session………… 26

2.3.2 WNDL Request………… 27

2.3.3 WNDL Loop………… 29

2.3.4 WNDL Episode………… 30

2.4 Limitations and Issues………… 32

Chapter 3 WNDL Executor Implementation 35

3.1 Architecture and Implementation of WNDL Executor………… 35

3.2 WNDL Program Evaluation Procedure………… 37

3.2.1 A Brief Introduction to Datalog………… 37

3.2.2 Executor Variable Binding and Propagation………… 40

3.2.3 WNDL Program Evaluation Procedure………… 41

3.3 Exception Handling……………………………………………………………... 46

Chapter 4 Experiments and Evaluations **47**

4.1 Expressiveness of WNDL………………………………………………………. 47

4.1.1 Chain and Loop……………………………………………………….. 48

4.1.2 Arbitrary Number of Data Extraction Passes………………………. 50

4.2 Utility of WNDL Wrapper……………………………………………………… 52

4.2.1 Configuring a New Wrapper………………………………………….. 52

4.2.2 Generality Across Application Domains…………………………… 54

Chapter 5 Related Work **55**

5.1 Web Automation Languages……………………………………………………. 55

5.1.1 Web Interface Definition Language (WIDL)………………………. 55

5.1.2 Web Language (WebL) and Hippo Core Language (HCL)………… 56

5.2 Work that Addresses the Web Navigation Problem……………………….. 59

Chapter 6 Conclusion **61**

6.1 Summary…………………………………………………………………………... 61

6.2 Critiques and Future Work……………………………………………………… 62

Bibliography **63**

List of Figures

1 . Information Integration System Architecture Diagram………………………... 4

2 . Web page structure of *amazon.com* in DWM representation………………….. 21

3 . Execution of a WNDL request…………………………………………………. 27

4 . Complex Loop Structure……………………………………………………….. 30

5 . Dependency graph between episodes………………………………………….. 32

6 . WNDL Executor Architecture…………………………………………………. 35

7 . Dependency graph of the predicates in Code 7………………………………... 38

8 . A next page link that can be found in a return page from *Yahoo!*…………….. 45

9 . DWM diagram of *TaipeiCity* and *TTimes*……………………………………… 49

10 . DWM diagram of *NYC*………………………………………………………… 49

11 . DWM diagram of *CTCareer*…………………………………………………… 50

12 . DWM diagram of *JobsDB*……………………………………………………... 50

13 . DWM diagram of *104Bank*…………………………………………………….. 51

List of Tables

1 . Queries to *amazon.com* and *BuyBooks.com* with book title, "*Core Java*"…….. 31

2 . Primitive Functions Used in Algorithm 3……………………………………… 42

3 . Web Sites in the WNDL Experiment…………………………………………... 48

4 . Twenty-seven Web sites that were used in the NCTU experiment…………….. 53

List of Codes

1 . An Example Subplan for Software Lego Executor……………………………. 10

2 . Nodes in DWM specification of *amazon.com*…………………………………. 22

3 . Edges in WNDL specification of *amazon.com*………………………………… 24

4 . `<VarDeclr>` block in a Session……………………………………………... 27

5 . Example of Request element…………………………………………………... 28

6 . Example of a self-looping edge………………………………………………... 29

7 . An Example Datalog Program………………………………………………… 38

8 . WNDL program in datalog representation…………………………………….. 40

9 . A WebL function excerpted from [14], which shops in *amazon.com*.………… 58

List of Algorithms

1 . Simple evaluation algorithm of datalog………………………………………... 39

2 . An algorithm that incrementally evaluates datalog programs…………………. 40

3 . WNDL Evaluation Procedure………………………………………………….. 44

Abstract

Utilization of the World Wide Web can be boosted if we can integrate information from various Web sites together. Information integrating systems rely on an intermediate software layer called *wrappers* to access connected Web information sources. Wrapper construction is often specially hand coded to accommodate the differences between each Web site. However, due to the interactive nature of Web browsing, unreliability of the Web, and the looseness of HTML, to program a Web wrapper by hand is error-prone and time-consuming. This thesis provides a solution for rapidly building Web wrappers for information integration systems and other applications that may be benefited by automated Web access. We defined a meta-language providing a representation of a set of Web pages and data on these pages, as well as how to locate the data, extract the data and combine the data. An interpreter/executor of this language was then implemented for running scripts written in this language. This executor can automate almost all types of Web browsing behaviors and tolerate dynamically generated HTTP request parameters, cookies, and illy-structured HTML. A series of experiments was conducted to evaluate the feasibility of our approach. The result shows that base on our meta-language, practical Web information integration systems can be rapidly constructed for a variety of application domains.

Chapter 1

Introduction

1.1 Information Integration

The purpose of building an information integration system is a three fold. First, we want to provide its users a uniform query interface to multiple connected information sources, which may be heterogeneous. Second, we want to keep the users from the detailed knowledge of individual query language and information locating procedure of each source. Third, we want to automate the task of integrating pieces of data from various information sources.

Current information integration systems are generally built with the architecture depicted in Figure 1 and work in the manner described below. A set of *domain relations* and *source descriptions* are defined according to the application domain where this system will be used and the information sources that are accessible to this system. Domain relations represent concepts of the application domain while source descriptions include *source relations* representing the contents of sources and configuration for locating and obtaining data from corresponding source.

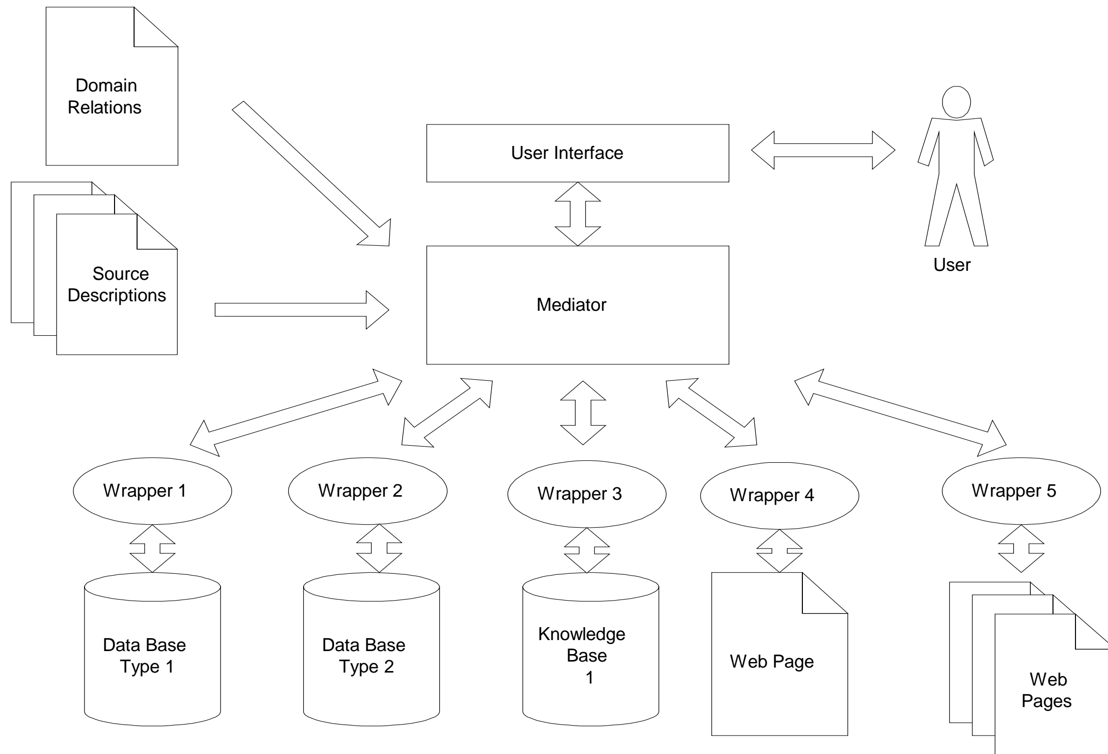


Figure 1. Information Integration System Architecture Diagram

Users of these systems formulate their queries in domain relation terms. In order to answer these queries, it is necessary to specify a set of relationships and mappings between domain relations and source relations. A query *planner* is required to transform user queries to a sequence of source queries. These newly generated queries are called *query plans,* which are telling which sources to query, how to query the sources, and how to integrate the data together.

To let the mediators be able to accommodate the differences between various types of externally connected information sources, each has its own access mechanism, query language, and data format, *wrapper* programs are built. They serve as the translators between mediators and the information sources. Wrappers provide a uniform access interface for each type of information sources to the mediator. They translate mediator query language to the query language of each source, obtain query result from the connected source, transform the query result to a uniform data format and send it back to the mediator. Each type of information sources requires a wrapper program to handle. In the example shown in the diagram, this mediator utilizes five different wrappers to integrate data from five types of information sources: two different types of database systems, one knowledge base, and two types of Web pages.

Information integration can add value to each individual piece of data. For instance, a demonstrating system named *TheaterLoc* [4] integrates data from five different information sources, one for restaurant information, one for movie show time information, one for movie preview videos, and two for map information. At first, users of *TheaterLoc* choose where they want to gather information of restaurants or theaters via the Web interface of *TheaterLoc*, an interactive map will immediately pump up. Users can then choose highlighted spots on that map to obtain more detailed information such as reviews, pricing information, and rating for restaurants, movie show time, and preview of movies of theaters. This integrated system provides its user a single cohesive application with seamless integrated information that cannot easily be done by a single information provider and shows the benefits of information integration systems.

1.2 Web Information Integration

Nowadays the World Wide Web has become an important and abundant information source. The value of the information on the Web can be even boosted if the relevant data from various Web sites can be integrated together. However, this information[1] integration task is not easy because the Web environment is designed for human users rather than for software agents. In this section we will discuss the problems we will encounter in building a wrapper for Web information integration systems.

1.2.1 Web Navigation with HTML Documents

This section briefly introduces HTTP (Hyper Text Transfer Protocol) and HTML (Hypertext Markup Language) features related to Web wrapper construction.

Currently, most of Web pages are written in HTML. The number of XML documents is increasing but still very few compared to HTML. Hence, we will focus on handling of HTML documents in this thesis. HTML pages use single directed *hyper links* to connect each other. Hyper links are specified by `<A>` tags in HTML. The target URL of the target page is specified as the value of the `HREF` attribute of `<A>` tags. The other typical way to make two Web pages "linked" is via `<Form>` tags. HTML forms allow Web users to send information or queries to Web servers. The

---

[1] In our discussion, we consider only integration of "text-only" data in Web environment. Multimedia information integration can be realized by extending our approach but in general is beyond the scope of this thesis.

information is then processed by a CGI (Common Gateway Interface) program hosted on the Web server, which will generate a response page to the user's browser. These CGI programs are specified as the *action* attribute value of `<Form>`'s. Although there are many alternative ways to produce interactive and server-side dynamically generated Web pages, such as ASP (Active Server Pages), Java Servlet, JSP (Java Server Page), etc, their interface to clients still follows the standard `<Form>` syntax.

The behavior of CGI programs is then determined by a sequence of name-value pairs sent by a Web client to the server. These name-value pairs are obtained from the various *controls* in the `<Form>` block. There are several types of form controls, including text areas, buttons, check boxes, radio buttons, etc. However, their underlying interface protocols are basically the same. That is, by name-value pairs parameter passing. Two methods are defined in HTTP to send these name-value pairs: get and post. When CGI parameters are sent by get method, they are appended to the end of the URL of the CGI program. The syntax of parameter passing starts by a question mark "*?*" followed by the name-value pairs, which are separated with each other by an ampersand "*&*". Suppose the URL of a CGI program is *http://site1.nyc.gov.tw/db/index.asp*, with parameters sss=1, and ttt=2. Then the complete URL to invoke that CGI program will look like: *http://site1.nyc.gov.tw/db/index.asp?sss=1&ttt=2*.

Post sends CGI parameters as a separate section following the HTTP headers and can send a larger amount of information than get. When a Web user clicks a button within a HTML form, these parameters will be sent by the browser to the Web server and trigger the operation of the specified CGI program. When the query result is too large, it usually will be rendered in multiple pages. Each page contains a certain number of data records and a link to *next page* is provided. Again this link to next page can be either a static link or another CGI query form.

Another frequently used mechanism for Web server/client interaction is *cookie*. Cookie is a piece of information for storing state of interaction between server side and client side. When responding a HTTP request, server attaches a *Set-Cookie* header containing cookie information with the returning document. The cookie will then be stored in the client side. When the client side continues to issue HTTP requests, it matches the stored cookies with the issuing request's URL and sends corresponding cookies with the request. Cookies are widely used by on-line merchants. Cookies can store currently selected items, payment options, registration information, and free the users from retyping the same information on following connections. Sites can also

store user preferences on the client side, when the client connects to those sites again. Servers can then present the Web site corresponding to the users' preference.

Another feature related to automatic Web navigation is relative URL, which is widely used in HTML documents. To access a Web page specified in relative URL, the browser will combine the relative URL with the base URL specified in `<Base>` tag or the URL of previous page to produce a complete URL.

1.2.2 Web Site Navigation Problem

Web information integration problem is different from information integration with databases because of the nature of Web structures. Data are contained in interlinked Web pages, that is, a web, rather than well-defined relations in database systems. Most of previous Web information integration systems such as *Ariadne* [1][2][15] simply model a Web page as a data source relation and ignore the necessity of navigation between Web pages. They did not capture the relationships between the linked data and Web pages and assumes that all Web pages can be obtained with a single HTTP request. The other Web information integrator called *ShopBot* [9] did not address this problem explicitly.

Web navigation mechanism is designed based on *interactivity* between human users and Web servers. Human users interact with Web servers by actions such as requesting a URL via locating the input text area, filling query string to a query form and clicking a button, etc.

As we discussed in the last section, to fetch a Web page from the Web server requires many kinds of interaction with the server. Request a Web page pointed by certain URL via get method is only the most basic one. In addition to URL's, other information such as *base URL*, *CGI parameters*, *cookie object*, *user ID/password*, etc, may be required. Because these values may only be obtainable from a sequence of Web accesses with the Web server, to specify these parameters with constant values in advance cannot solve the problem.

For example, considering a news site providing hourly-update news, links on their news index page to the detailed story page change every hour. Obviously, these detailed stories cannot be obtained via constant URL's. A more promising way is to fetch the index page, extract the detailed-story links, and obtain detailed story pages by following these links. In this example, the index page is an *intermediate page* that

does not make sense to the application. In an extreme case, visiting many intermediate pages may be required to obtain the target information. For example, some pages are necessary only for obtaining a *cookie* object to continue the navigation, and another pages only for obtaining the *user session ID* or triggering a user session ID mechanism inside the Web server. There are other examples that also prove the necessity of navigating Web pages to obtain target information. For example, the "*next pages*" of query results from a search engine. Links to those *next pages* are necessarily generated during run-time.

These examples all shows that modeling each page as a source and then each source can be accessed with a set of preset parameters is incorrect and not appropriate in wrapping Web information sources. Moreover, the Web is an *unreliable* information source. The quality of service of the Web varies with time and may be down.

In addition to the problems listed above. HTML itself is another source of problems in Web wrapper construction. The rapid growth of the World Wide Web comes from several reasons. Two important ones among all of them are the simplicity and loose specification of HTML. Even an amateur that has no experience in computer programming can learn how to use HTML to structure and style his/her documents in a very short time. However, from the view of a software agent developer, this property of HTML may cause problems. Illy-written HTML documents by unskilled HTML programmers are everywhere on the Web. Commercial browsers are laboriously programmed to tolerate these "*invalid*" HTML documents so that they can render them without causing fatal problems. But these documents did cause serious problems for Web agent developers.

Another problem comes from the limited restriction on syntax of HTML. For example, some end tags can be missing while other tags have no end tags. Ambiguity may arise in determining the scope of those tags. Besides, HTML documents are not agent-friendly because HTML tags reveal layout information only and cannot present sufficient semantic meaning. As a result, numerous data extraction techniques were developed to conquer this problem.

1.2.3 Data Extraction Problem

In the last section, we mentioned that navigation on the Web requires information extracted from Web pages during run-time. Data extraction problem is thus *how can we fetch the target data from a Web page*. Web pages are composed for human readers and are formatted in free texts. Hence, Web pages usually do not have a structured data format like a table in a relational database. Therefore, extracting data from Web pages is not so direct as selecting a row from a relational database.

Due to the daily increasing amount and format changing frequency of Web pages, it will be very tedious, time-consuming, and error-prone to use hand-coded programs for extracting data from Web pages. It is necessary to generate a Web page extractor *automatically* or *semi-automatically*. Several systems addressing this *wrapper induction* problem are initiated, for example, the works of Muslea [19], Kushmerick [16], and Hsu [11][12]. These works use machine-learning techniques to produce Web page data extractors with human labeled training examples. Instead of the methods taken by earlier work which need heavy domain specific and linguistic knowledge, the more recent research like the three listed above take advantage of the regularity of *semi-structured*[1] Web pages.

Online data sources frequently use tables or itemed lists to organize representation of their data, moreover, many of these pages are in fact generated by computer programs with data come from certain database(s). Hence, these data pages usually have some kind of regularity. Unlike normal plain text files, Web pages come with useful additional information such as tags for formatting appearance of pages. Therefore, Web page data extractors rely on regularity of the contents of data pages can show their strength in extracting data from these pages with high success rate.

Our work equipped with *SoftMealy* [11][12] extractor of Hsu as our Web page data extraction subsystem. Compared with previous works, SoftMealy addresses the following problems that cannot be handled by the previous works to cover a larger class of documents.

- Missing attributes
- Multiple attribute values
- Variant attribute permutations
- Exceptions and typos

[1] A semi-structured page means the desired information contained in this page can be located using a concise and formal grammar. This definition is found in [19].

1.2.5 Our Approach

To overcome the problems mentioned above and to provide a solution for automating Web interactions, we propose our approach as the following description: We would like to develop a configurable wrapper that wraps a set of Web pages rather than wraps only one Web page. The main purpose is to leave the upper layer mediator from handling the details of Web page locating and data extraction tasks. From the view of mediator, it can treat the set of Web pages wrapped by this wrapper as a relation. Whenever the structure of these pages changes, the mediator can be kept untouched if the integrated meaning represented by these pages doesn't change. To achieve these requirements, we previously proposed *Software Lego* [13] system, which shares the same architecture and basic ideas with the work that will be presented in this thesis.

The *Software Lego* executor executes a *datalog*[24]-like navigation plan. The plan is a text file containing blocks of *subplans*. Each subplan is an execution unit and constitutes a *domain relation* that is tailed with one or more *source relations*. The domain relation represents a user query while a source relation represents the data content of a Web source that is connected in this system. The *attributes* in a source relation means what can be obtained from this source or what need to be sent to the source to get values of the other attributes back. A set of system built-in functions was defined for arithmetic operations and string operations. Each attribute name in a relation has a global scope. Source relations are executed as the order listed in the subplan. URL and variable value bindings for CGI parameters and extracted data of each source are explicitly specified in a separated source configuration file.

```
author_query(Detail, Title, Author, Format, Date, NT_Price) :-
    amazon_author(Title_Keyword, Detail, Title, Author, Format, Date,
        US_Price) &
    currency_table("TWD Taiwan Dollars", "USD", Value) &
    MULTIPLY(Value, US_Price, NT_Price).
```

Code 1. An Example Subplan for Software Lego Executor

In the example shown in Code 1, we assume that two information sources are connected in this system. One is a bookstore source named `amazon_author`, and `currency_table` is a Web site that provides information of exchange rate. In this example, source relation `amazon_author` means the data obtained by querying about authors through a CGI form. In this subplan, `Title_Keyword` is a variable that must be bound prior to the execution. Executor first queries the CGI form with

bound value of `Title_Keyword`. And then it gets values of the following variables: `Detail, Title, Author, Format, Date, US_Price` from the output pages for the value of `Title_Keyword`. After that, Executor gets the exchange rate between U.S. dollars and New Taiwan dollars from the source, `currency_table` by sending *"TWD Taiwan Dollars"* as a query to that source. Then Executor calculates NTD prices of each result data from exchange rate and USD prices. Finally it returns rows of data with following attributes: `Detail, Title, Author, Format, Date, NT_Price`.

Software Lego can visit a sequence of Web pages, extract data from these pages, use obtained data in further navigation, traverse next pages, join and project obtained data into a target data schema. However, it has some drawbacks. To address these problems, we developed a new version of our system. We defined an XML (eXtensible Markup Language [26])-based language, WNDL (Web Navigation Description Language), which will be introduced in the next chapter. WNDL has intuitive expressions in Web concepts and can represent complex Web client/server interactions. A WNDL executor was then implemented. When compared to *Software Lego*, the WNDL wrapper has the following benefits.

1. XML format configuration file rather than specific format can ease information interchange between applications.
2. More *flexible* variable value system. All variable names are global in *Software Lego* thus the same source relation cannot be executed twice. This seriously limits the expressiveness of navigation plans of *Software Lego*.
3. *Recursive* evaluations in plan execution.
4. More robust and reliable "*next pages*" handling.
5. *Simplified* and *unified* configuration file. Software Lego executor requires six different kinds of configuration files. Now all required information could be stored in only one configuration file with unified look and feel. This makes the system maintenance easier.

To sum up, the combination of the executor and the definition of WNDL can offer the following features:

- Visit a sequence of Web pages and accumulate data extracted from these pages.
- Insulate the upper layer application from changes in the page structure of the connected Web site and changes of the formats of documents.
- Insulate developers from concerning HTTP and HTML pages handling.
- Declaratively represent complex navigation and data evaluation on Web.
- Handle dynamically generated static links and CGI query HTML forms.
- Generate Web pages from values obtained during processing.
- Handle *user session* problem.
- Automatically determine the end of multiple *next pages*.
- Automatically handle *base URL* assignment when relative URL's are used in Web pages.
- Tolerant to many mal-formed HTML documents.

The features listed above plus the data extraction power offered by *SoftMealy* extractor, the WNDL solution is expected to cover a large percentage of Web pages and Web sites. Moreover, WNDL and its executor can not only serve as a wrapper in a Web information integration system, but also can be used in various applications like:

- Customizable Web crawler.
- Information gathering robot.
- Web site links validating robot.
- Comparison-shopping bot.
- Meta-search engine.
- Meta-newspaper.

And many other applications that can be benefited from automated Web access.

1.3 Organization of this Thesis

This thesis is primarily discussing the automation of information gathering on the Web. Chapter 2 presents the meta-language, WNDL defined for describing Web navigation behaviors to facilitate the automation. It is followed by Chapter 3 that explains the evaluation procedure of WNDL and the implementation of the whole system. In Chapter 4, we will evaluate WNDL in terms of expressiveness and utility by experiments. Several related work will then be introduced in Chapter 5. Finally, Chapter 6 summarizes the whole contents and discusses critiques and possible future works.

Chapter 2

Web Navigation Description Language (WNDL)

2.1 Preliminary

The language, WNDL (Web Navigation Description Language) is an XML-based (eXtensible Markup Language [26]) language for describing navigations for Web information gathering tasks. WNDL is the first part of our solution to automate information locating, extraction, and integration on Web. A WNDL script describes how the data is stored in target Web sites, how to get them and how to bind the values. Then the executor of WNDL will interpret the instructions in the script to complete the task and return the result. Currently, human writers are required to compose WNDL scripts. In the future, we plan to develop techniques to automate WNDL scripts generation.

WNDL is an application of XML. A set of XML elements is defined. The combination of XML and a WNDL executor implemented in Java introduced in the next chapter makes this solution platform independent. Information agents like comparison-shopping robot can be built easily and rapidly on nearly all-existing computer platforms. The Web site configuration files can also be interchanged between heterogeneous hosts.

In an information integration system, wrappers are specialized to each connected information source type. These wrappers provide a uniform interface for the information sources to the mediator. We assume that the data can be represented in relational data model. That is, the data is modeled as a set of relations (tables) with a fixed number of attributes, and each attribute has no more than one value. Our wrapper makes Web pages look like a data table from the view of the mediator and insulates the mediator from handling the details of Web navigation. Besides, we also assume that all attributes are in string type.

WNDL is capable of describing:

- how to locate data in a logical Web site;
- the contents in target pages;
- how to extract target data from each page;
- the information required in obtaining each page in the set via following a static link or a form query that can be dynamically generated;
- combining the data coming from each page data set into the target data set.

2.1.1 A Brief Introduction to XML

XML is a markup language for describing information. It provides a framework for creating new markup languages. The predecessor of XML is SGML (Standard Generalized Markup Language), which is also the predecessor of HTML. Different from HTML, XML does not have a predefined fixed set of tags. XML allows the developers to define their own set of tags or use the one defined by others. Unlike HTML, XML is not only for formulating the rendering of documents with tags, but also simplifies the transformation of information among XML documents. Therefore, the emergence of XML indeed supplements the drawbacks of HTML in the aspect of data interchanging.

The most attractive aspect of XML is that it allows XML designers to define their own tags. With this capability, semantic meaning can now be attached on the tags though the interpretations are left to XML processors. With the flexibility, XML is not only a new format of Web pages, but also a container of data and control instructions. For example, considering an on-line book shopping application. The current way for representing a record of book in HTML may look like the following code:

```
<Table>
    <TR>
        <TD>Title</TD>
        <TD>Java Network Programming</TD>
    </TR>
    <TR>
        <TD>Author</TD>
        <TD>Elliotte Rusty Harnold</TD>
    </TR>
</Table>
```

In contrast, if we represent the same data in an XML-based language, it may look like the follows:

```
<BookRecord>
    <Title>Java Network Programming</Title>
    <Author>Elliotte Rusty Harnold</Author>
</BookRecord>
```

In the HTML version, tags only show how the data will be rendered in the browser window, not the meaning of the data. This makes the use of HTML documents by a computer program difficult. On the other hand, the XML version makes processing of this document by a computer program easier because the program can infer the meaning based on the tags. (`<Title>`, `<Author>`, etc)

XML itself comes without any predefined tags. It leaves services such as display control and hypertext linking to subsidiary languages. For example, XML leaves display control to style sheet languages such as XSL (eXtensible Stylesheet Language) and CSS (Cascading Style Sheets), and hyper links to XLink [28] (XML Linking Language).

XML allows developers to define their own document structure with document definition languages such as DTD (Document Type Definitions). With DTD, XML can strictly constraint document structure. Whether an element is required and where it should be can be defined. For example, XML can enforce a documents type of book data to have title, author's name, publisher's name, and its ISBN code. The order and containing relationship can be defined, too. Continuing the example, books that relate to programming languages can be defined to be in the computer science category.

Primary building blocks of XML documents are *elements* with *attributes* and *contents*. Elements are formed with the application of tags. For example, as the following XML code indicates,

```
<BookRecord isbn="0-13-960162-7">
    <Title main="XML by Example"
              subtitle="Building E-Commerce Applications"/>
    <Author name="Sean McGrath"/>
</BookRecord>
```

there are four tags in this code segment, `<BookRecord>`, `<Title>`, `<Author>`, and `</BookRecord>`. They represent three elements, BookRecord, Title, and Author. BookRecord element has the contents, Title and Author. And these elements have their own attributes such as: isbn, main, subtitle, and name.

2.1.2 Term Definitions

The following terms are used frequently in the whole context with special meaning. Although the terminologies used in this text are primarily based on a working draft, *"Web Characterization Terminology & Definitions Sheet"* [27], from the World Wide Web Consortium (W3C). We reused some of these terms and endowed them with another meanings. Their specific meanings to this context are defined as below.

Definition 1 (Logical Web Site) *A cluster of Web pages that are related to each other, each page contains certain amount of data. The data distributed among these pages can be integrated together and have a logical meaning.*

Definition 2. (Episode) *A sequence of related Web requests for information gathering that could fulfill the need of certain purpose.*

For example, a Web user wants to know what books related to a keyword, "Java", are sold in the on-line bookstore, *amazon.com*. To fulfill this need, he first connects to the homepage of *amazon.com*, finds out the book search page, inputs the term, "Java", in the form text area, presses the "Go" button and gets search result page(s). In this episode, at least three Web requests are sent. If the Web user wants to get the full list of the books match the keyword, he may click the "More Results" button on the search result page(s). All of these Web requests are related and constitute an episode.

Definition 3. (Session) *(User) Session is a higher-level container of multiple discrete episodes and operations using the collected information as operands.*

To finish a complete Web information-gathering task, or to answer a user query within an information integration system, sometimes multiple episodes are necessary. Continuing the example in the definition of an episode, after getting the title list of the books related to "Java" as well as the price list in U.S. Dollars from the first episode, the Web user may then want to know the prices in New Taiwan Dollars so that he/she can determine which book to buy. Hence, he/she visits the Web site of Hwa-Nan bank, after several clicks he/she can find the exchange rate between U.S. Dollars and New Taiwan Dollars. This forms the second episode. At last, the user uses a calculator to compute the prices in New Taiwan Dollars of the books that he/she is interested in purchasing. In this example, there are two episodes and an additional operation, multiplication. All of these steps constitute a *User Session*.

Definition 4. (Web Page Class) *A set of Web pages that a (SoftMealy) extraction rule can be applied to parse and extract their contents.*

WNDL executor utilizes a SoftMealy data extractor to extract data from Web pages. This extractor extracts data according to an extraction rule. A Web page class means the set of pages that one extraction rule can be applied to. This usually denotes Web pages generated by one single CGI program or Web pages with identical layout format.

2.1.3 Organization of WNDL

The purpose of defining WNDL is to enable the automation of information gathering procedure on the Web. WNDL is a set of XML elements. Each represents a specific meaning, associated with a set of element attributes, and contains contents elements. A valid WNDL document for a logical Web site consists of two primary parts. The first part describes the page structure and the distribution of meaningful data of this site. We call it *Data Web Map* (DWM). The second part is a block containing the *Navigation Programs* for locating the desired data as well as the relationship between the data.

2.2 Data Web Map (DWM)

The DWM part is the primary data container in WNDL documents. The information stored here describes how to open target Web pages, and how to extract the data on the obtained pages.

After inspecting normal Web browsing and information gathering activities, we found the following modeling is quite reasonable and intuitive. WNDL models a logical Web site (or a set of Web pages) as a directed graph. The objects contained in the graph include *nodes* and *uni-directed edges* that connect nodes. Each node on this graph represents one Web page class, and an edge represents a possible mean to reach its destination node from its source node. Generally this includes following a static link or querying a CGI program. For a given CGI program, there may be more than one edge to represent different ways to query it. For example, we may have one edge for query author, one for query book title for the CGI program of an on-line bookstore.

In WNDL, the definitions of DWM objects are enclosed in the Map element. Contents of the Map element include an Entrance element and one or more Node elements. Contents of Entrance element are one or more Edge elements. The edges in the Entrance element represent the way to access the logical Web site directly outside the scope of the defined map without further interaction with the Web server. Typically, this leads to front page of a Web site. For example, the page you can get via the URL's, *"http://www.yahoo.com"*, *"http://www.altavista.com"*, etc.

In the following sections, we will go through a complete example for modeling the behavior of querying books on the site of well-known on-line merchant *amazon.com*. In this sample model, we are interested in two Web page classes among all of the documents from *amazon.com*. We model the procedure to send a keyword for querying books against *amazon.com*'s search engine. The resulting DWM has two nodes and three edges as Figure 2 depicts.

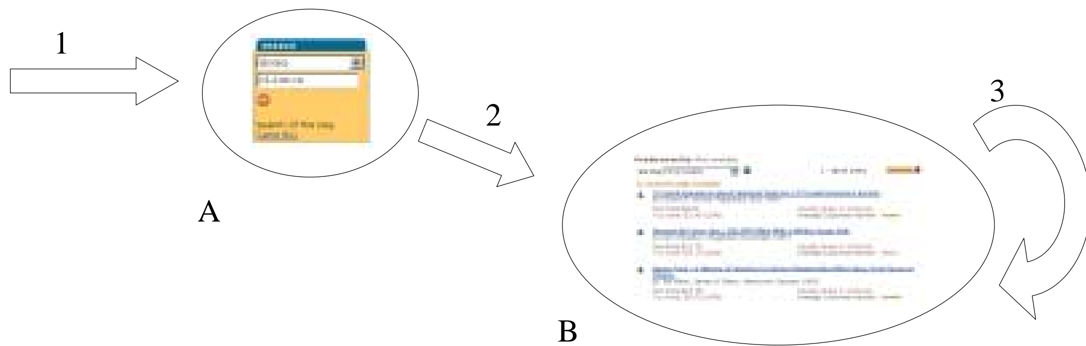


Figure 2. Web page structure of *amazon.com* in DWM representation

2.2.1 Data Web Map Node

A node represents one page class in the described logical Web site. Defined again here, a page class is the set of Web pages that one single extraction rule can be successfully applied to. A Web page class may contain an arbitrary number of Web pages when the class represents pages that are generated by a CGI program. Obviously, the number of Web pages that a CGI program can generate is innumerable. For example, considering a query program in a search engine like *Yahoo!*. Virtually any kind of query string can be placed for the processing of that program. The page(s) it returns is determined not only by the query string but also by the time when the query is placed.

In our model, each node is also considered as a container of data that can be found in the pages belong to corresponding Web page class. The data in a node is represented as a relation in the evaluation procedure taken by the executor. It is a table associated with a *schema,* which is an attribute name list of the target data that will be extracted from pages of that page class. Each attribute has an option for HTML tags filtering. This determines the behavior of a built-in HTML tag filterer of data extractor. Three modes are available:

1. Keep all HTML tags.
2. Keep no HTML tags at all.
3. Filter out all HTML tags but keep the value of HREF attribute of <A> tags.

HTML tag filtering is necessary because a lot of redundant HTML tags exist in Web pages for formatting purpose only, for example <BR> for new line, <IMG> for displaying an image, etc. These tags interfere with real data and are better to be filtered out to make better use of real data. On the other hand, HTML links play important role in Web navigation, so the default behavior of our extractor is to keep

the URL content of links and filter out all tags. Each attribute can be set separately with respect to this option.

Another essential element of a DWM node is the ExtractRule element. The content of this element is raw text of the extractor's extraction rule for this node. An external file can also be specified as the File attribute of this element. When the two both present, rule specified in the content will be used.

At last, Node element also has a key element, Edge, as its contents. Edges that are contained in a node mean these edges start from this node and link this node to another node.

There are two nodes in the example model as showed in Figure 2. Node *A* denotes the front page of *amazon.com*, and node *B* denotes query result pages returned from query answering program of *amazon.com*. (Edge elements are removed for abbreviation purpose.)

```
<Node Name="A">
    <Schema>
        <Attr Name="form" TagFilter="KeepAll"/>
    </Schema>
    <ExtractRule File="amazon_home_rule.txt"/>
</Node>
<Node Name="B">
    <Schema>
        <Attr Name="title"/>
        <Attr Name="author"/>
        <Attr Name="year"/>
        <Attr Name="price"/>
        <Attr Name="url_next"/>
    </Schema>
    <ExtractRule File="amazon_rule.txt"/>
</Node>
```

Code 2. Nodes in DWM specification of *amazon.com*

In node *A*, the information we are interested is the `<Form>` HTML tag block in this page. It is the only data that will be extracted from this page. This attribute is named as `form` in this example. Some Web sites use *user session id* mechanism to

recognize HTTP requests from identical user to keep track of a user session. This helps Web servers to determine the contents of Web pages they should response according to each user. User session id is usually implemented by the following mechanism: When a user first connect one of this sort of Web site, the front page returned by the Web server is in fact generated by a program, it assigns an id embedded in this Web page every time. When the user clicks links, query forms, and other interaction actions within this Web site, the assigned user session id will be carried with this server session. In some Web sites, HTTP clients need this id to continue navigation. Some Web sites optionally use this id. Although *amazon.com* belongs to the later category, we demonstrate the use of dynamically generated *HTML form text* here. Because the query form text is extracted during run time, we can obtain the user session id information that is generated dynamically.

Node *B* represents the query result returned by *amazon.com*'s search engine, the information we are interested include title, author, year, price, and a segment of HTML text (may be a static link or a query form) that links to the *next page* of the query result. Multiple result pages can be handled by WNDL executor during run-time with several heuristic rules. The detailed mechanism will be explained in the next chapter. *Next pages* will be fetched and the data will be extracted automatically during the evaluation process of WNDL document to make all data appearing to contain in a table of a relational database.

2.2.2 Data Web Map Edge

An Edge in a DWM represents a possible mean to obtain a page that belongs to a Web page class denoted by the destination node of this edge. DWM edges serve as containers of the necessary information of actual HTTP requests. Purpose of this part of WNDL is very close to another Web interface definition language, WIDL (Web Interface Definition Language) [25], that is, to define a HTTP session to automate Web accesses.

DWM edge can be used to represent the interfaces for both static HTML documents and dynamically generated HTML documents. This interface is defined by a set of parameters. Values of these parameters can be either *constants* or *variables* with run-time generated values.

```
<!-- This is an entrance edge. -->
<Edge ID="1" Type="Static" Method="Get" Dest="A"
       URL="http://www.amazon.com"/>

<!-- This is an edge within node A -->
<Edge ID="2" URL="$query" Method="Post" Type="Dynamic" Dest="B">
    <EdgeInput>
        <EdgeParam Name="query"/>
        <EdgeParam Name="keyword"/>
    </EdgeInput>
    <Query>
        <QueryParam FormInput="index" Value="books"/>
        <QueryParam FormInput="Go" Value="Go"/>
        <QueryParam FormInput="field-keywords" Value="$keyword"/>
    </Query>
</Edge>

<!-- This is an edge within node B -->
<Edge ID="3" URL="$next" Dest="B">
    <EdgeInput>
        <EdgeParam Name="next"/>
    </EdgeInput>
</Edge>
```

Code 3. Edges in WNDL specification of *amazon.com*

Edge *1* is an entrance edge of this map. It opens the front page of the target Web site. In this case, it is the destination node of edge *1*, node *A*. The action of edge *1* takes is quite trivial. Let's examine the more complex edge *2*. In some sense, WNDL can be considered as another programming language with a specific interpreter, and edges can be viewed as subroutine definitions in WNDL programs. An `<EdgeInput>` block can be found in the definition of edge *2* and means to invoke the operation of edge *2*. Values of two variables are required, `query` and `keyword`. So the EdgeInput element is actually a local variable declaration block, and two variables are declared here. The value of URL attribute of edge *2* is a variable, "`$query`". It is a variable reference[1] in WNDL. This means the URL of this HTTP

[1] In the cases of constant values with dollar sign prefixes, they are required to be escaped by duplicating the number of prefixing dollar sign. Hence, values with odd number prefixing dollar sign(s) are variable references and values with even number prefixing dollar signs are constant values.

interface is a variable and will be bound during run-time. In WNDL, HTML forms are treated as a parameterized URL, which will be explained later. URL of edge *2* in this example is the form text extracted from node *A*. Because this edge represents a query to a CGI program via a form, its type is `Dynamic`, in contrast to `Static` type of edge *1*.

The other noticeable block of edge *2* is the Query element. In this example, this means to query *amazon.com*, at least three parameters are required to be set. There may be some form parameters in hidden input type with constant values. In this example, since the whole form text will be extracted, these hidden constants can be found in the form text, it is not necessary to set them. The three input parameters `field-keywords`, `Go`, and `index` in this case are chosen by human Web users. This code segment means when the HTTP request following edge *2* is issued during run-time, `field-keywords` will be bound to a WNDL local variable, `keyword`, which is the query string to this CGI from. Parameter `index` is bound to a constant value, "books". When a human Web user browses this page, this value is specified by manually selecting an entry from the pull down menu. By inspecting HTML source of that page, we can find out the value of that parameter for querying books is "books". Similarly, the parameter `Go` is bound to value "Go". In this example, the human users fill the query string, and choose the query category. At last when the user wants to issue the query, he/she clicks the "Go" image button. Form parameter `Go` will be bound to "Go" at that time.

Let us turn to edge *3*. Edge *3* is an edge that has identical source and destination node just depicted in Figure 2. Therefore, it is a self-looping edge. Like edge *2*, edge *3* also has a URL as a variable that refers to the link underlying "More Results" image button that is in node *B*. During run-time, node *B* will form a self-loop. We call how to access pages requring iterations and how to terminate the iteration as the "*next-page problem*". As described above, next-page problem can be handled by WNDL.

The other element available as the contents of *Edge* element is the Timeout element. It contains the control information of timeout event handling. Because Web sites may not always available and network traffic may be jammed, to prevent infinite waiting of network connection, controlling timeout event is necessary in networking applications. In WNDL, number of retry attempts and time interval between each attempt can be specified. This time interval is equal to the time bound of timeout event. If all attempts fail, the executor will throw an exception signal to its invocator.

2.3 Navigation Program

Navigation program part of a WNDL script specifies the programs (or plans) to navigate the DWM described in the same document. It works like a computer program that utilizes the predefined DWM objects. WNDL navigation programs are made to represent sequences of HTTP requests that constitute particular Web information-gathering tasks. The sequence can be considered as paths on a defined DWM.

In WNDL, a `<Program>` block encloses navigation programs. Primary elements contained in the Program element are Session, Episode, Loop, and Request. WNDL navigation programs are divided into several hierarchies. The atomic operation is an individual HTTP request. A Request element that represents a HTTP request is the atomic operation in WNDL. Session, Episode, and Loop are higher-level logical structures of Request's.

2.3.1 WNDL Session

A Session element represents a user session, which is the highest-level structure in Navigation Program. Each session represents a unit of the execution of a WNDL specification and has a set of session level variables. These variables are required to be declared first before they can be use. They are value holders during the whole process of execution. Therefore, they can be considered as global variables. Value substitution of these session level variables is the core part of WNDL document evaluation and execution process that will be explained in the next chapter.

A Session element contains two parts: a variable declaration part (VarDeclr) and an actual Web navigation operation description part (Access). In VarDeclr element, the content is a list of session level variable declarations (SessionVar). Each variable's type and label are defined in SessionVar. Four session level variable types are available.

In The value will be given at run-time before the execution of this session.
Out This variable will be presented as a column at the final result data set.
InOut This variable is both an *In* and an *Out*.
Tmp This variable will be used in the following navigation program, but it is neither an *In* nor an *Out*.

The output of a WNDL execution is a relational table. The schema of this table is the session variables with type *Out* or *InOut*. Label is an attribute of SessionVar for renaming the column names at last. Constants that will be used in the program are also need to be declared here with an additional Value attribute. If this attribute presents, this variable will serve as a constant in the following evaluation process with the value given by the value of Value attribute.

Multiple user sessions can be defined in a single WNDL document. They are totally independent to each other but use the same definition of Data Web Map. All of them are contained in a Program element in a WNDL document. In each execution of this document, one user session is selected and executed by the calling program.

```
<VarDeclr>
    <SessionVar Name="X"  Type="Tmp"/>
    <SessionVar Name="Y"  Type="Tmp" Label="Query String"/>
    <SessionVar Name="Z1" Type="Out" Label="Title"/>
    <SessionVar Name="Z2" Type="Out" Label="Author"/>
    <SessionVar Name="Z3" Type="Out" Label="Date"/>
    <SessionVar Name="Z4" Type="Out" Label="Price"/>
    <SessionVar Name="Z5" Type="Tmp"/>
</VarDeclr>
```

Code 4. <VarDeclr> block in a Session

2.3.2 WNDL Request

WNDL requests represent HTTP requests. In WNDL, this means to follow an edge on DWM, enter into a node and fetch data from that node. The execution of a WNDL program can be conceptually considered as traversing across a finite number of program states, which correspond to the nodes where the program position indicator is pointing.

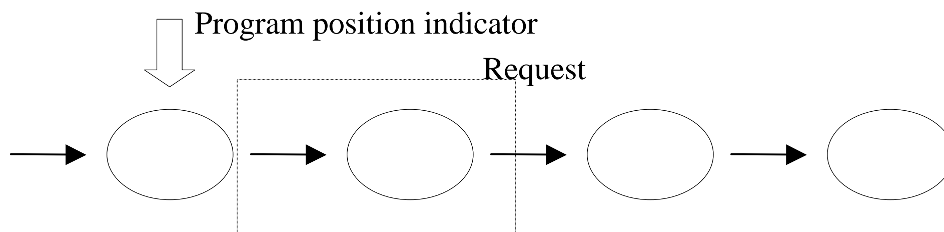


Figure 3. Execution of a WNDL request

Executing a request means a transaction of a node. In addition to specifying which edge to follow, the value binding relationships between local variables of DWM nodes and edges and global variables defined in session level are also required. The following code segment shows a request following edge *2* in Figure 2.

```
<Request Edge="2">
    <VarBind Name="query"       Type="Edge" Value="$X"/>
    <VarBind Name="author_name" Type="Edge" Value="$Y"/>
    <VarBind Name="title"       Type="Node" Value="$Z1"/>
    <VarBind Name="author"      Type="Node" Value="$Z2"/>
    <VarBind Name="year"        Type="Node" Value="$Z3"/>
    <VarBind Name="price"       Type="Node" Value="$Z4"/>
    <VarBind Name="url_next"    Type="Node" Value="$Z5"/>
</Request>
```

Code 5. Example of Request element

The Edge attribute of Request element indicates which edge will be followed in this step. Inside the Request element are all VarBind elements. Each VariBind element represents a variable binding. Two types of variable bindings are available: *edge* variable bindings and *node* variable bindings. As illustrated by Figure 2, Code 2, and Code 3. Code 5 represents a state transition from node *A* to node *B*. That is, to send a CGI query HTTP request to the Web server via the setting in edge *2* and to obtain pages belong to the page class represented by node *B*. By inspecting Code 2 and Code 3, the local variables contained in edge *2* are `query` and `author_name`, and the variables belong to node *B* are `title`, `author`, `year`, `price`, and `url_next`. Those are the variables need to be bound in this request. They are specified to be bound to `X`, `Y`, `Z1`, `Z2`, `Z3`, `Z4`, and `Z5` respectively when the request is being executed. But the direction of value binding is different between these two kinds of variable binding. Before execution of this request, session variables `X` and `Y` are supposed to already have some valid value tuples. These valid value tuples will be bound to edge *2* local variables according to the specification in edge *2*'s definition block. Then a sequence of corresponding HTTP requests will be sent to obtain node *B* pages. After these edges are fetched, the extractor rule(s) specified in node *B*'s definition block will be applied to these pages and data will be extracted from these pages according to the applied rule(s). A data table representing these data will be generated. The initial schema of that table is the one declared in the Schema element of node *B*. The data will then be bound to session variables as the specification that can be found in Code 4. These data will be propagated to following execution.

2.3.3 WNDL Loop

Loop is a higher-level logical structure than requests. It represents looping Web navigations. An example of looping behavior during practical Web browsing is clicking the next page on the result pages returned by a search engine. The pages returned by a search engine usually constitute a Web page class. That is, a node on the DWM. Following next page links is actually a self-loop on one single node. It is the most frequently encountered loop structure in actual Web navigation.

```
<!-- The original position of this edge is within the block of node B -->
<Edge   ID="3"
        URL="$next"
        Dest="B">
    <EdgeInput>
        <EdgeParam Name="next"/>
    </EdgeInput>
</Edge>
<Loop Max="100">
    <Request Edge="3">
        <VarBind Name="next" Type="Edge" Value="$Z5"/>
    </Request>
</Loop>
```

Code 6. Example of a self-looping edge

Continuing the *amazon.com* example, Code 6 represents the self-looping structure of the query result. Edge *3* adjacent to node *B* is a self-looping edge that start from node *B* and point back to node *B* itself. The sequence of HTTP requests is chained via the information from variable `next` in edge *3* and the session variable, `Z5`. Recall that the request code segment in Code 5, session variable `Z5` is bound to node variable `url_next`. `url_next`'s value is the URL underlying the "More Results" image of *amazon.com*'s query result pages. This link is extracted from node *B* pages and is bound to `Z5` of result data set. When issuing HTTP requests following edge *3*, values of `Z5` is bound to `next` in edge *3*. Hence, a loop is formed. The termination of iterations in a loop is automatically determined by several heuristic rules, which will be explained in the next chapter. An upper bound of Loop iteration number can be specified. This is particularly useful when we are only interested in the most recent data. In that case, extracting all data from the returned pages is time-consuming and unnecessary.

Though the most frequently used loop structure in practical Web navigation is to handle next pages. The definition of loop within WNDL can be more general. Multiple requests can exist in a single loop. Loop structure definition can even be nested, that is, a loop can be defined within another loop. These more generic loop structures can be defined and evaluated in WNDL, though these cases are rarely seen in real Web sites.

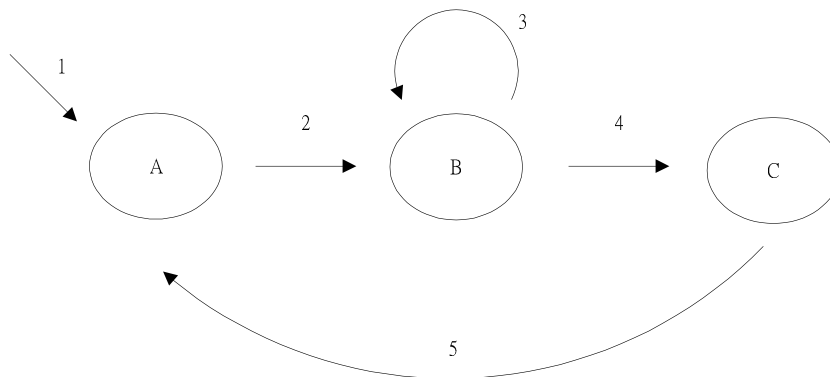


Figure 4. Complex Loop Structure

Figure 4 shows a complex loop structure that can be represented by WNDL. An example of this circumstance is: there is a CGI form in node *A*. Edge *2* is a CGI query. Edge *5* connect node *C* to node *A* when data extracted from node *C* is then used to query the CGI program contained in node *A*.

2.3.4 WNDL Episode

An episode in WNDL is a chain of HTTP requests. The representations of the visited DWM objects by an episode are similar to the one shown in Figure 3. It is a path on DWM starts with an entrance edge. An episode is represented by Episode element in WNDL. The contents of an Episode are Request and Loop elements.

To complete the information-gathering task of a user session, multiple episodes can be defined. They are basically chains of HTTP requests. They may or may not have identical component objects. There is no limitation on the component DWM objects between each episode. But they are supposed to form a complete logical unit and produce a data set. Continuing the *amazon.com* example, the process of Figure 2 is a complete episode. There may be another episode defined in the same user session integrally to complete an application. For example, in an on-line comparison-shopping application, another episode for querying the same keywords to *BuyBooks.com*, can be defined. The episodes share the same set of session variables.

Then the results may look like this table:

Merchant	Title	Author	Year	Price
amazon.com	Core Java 2, Volume 2: Advanced Features	Cay S. Horstmann, Gary Cornell	December 27, 1999	31.49
amazon.com	Core Java 2 , Volume 1: Fundamentals	Cay S. Horstmann, Gary Cornell	December 15, 1998	30.09
BuyBooks.com	Core Java 1.2; Vol. 2	Cornell, Gary & Cornell	1/1/2000	33.99
BuyBooks.com	Core Java 1.2; Fundamentals With CDROM	Horstmann, Cay S. & Cornell, Gary & Horstmann & Cornell	12/1/1998	32.99

Table 1. Queries to *amazon.com* and *BuyBooks.com* with book title, “*Core Java*”

WNDL not only allow independent episodes within one session but also allow existence of dependency between episodes. Evaluation of one episode can rely on data gathered from another episode(s). To achieve this, episode dependency can be specified in Dependency sub-element of Episode element. A list of prerequisite episode names is given there for dependency specification. Dependency between episodes can be recursively defined in WNDL. For example, in Figure 5, episode *E2*, *E3*, and *E4* form a cycle. On the other hand, *E1* has no prerequisite and is independent to *E2*, *E3*, and *E4*.

Besides the dependency specification, *HTTP authentication* information can be specified in Episode element if authentication is required. Some Web sites are not for public access and require user ID/password pair to protect sensitive information inside the Web site. A single pair of user ID/password authentication may protect more than one class of Web pages. They are called “realms” in HTTP authentication terminologies. Hence, the authentication information is supplied at episode level, not request level. A separate episode can be defined so as to access a realm of password-protected pages.

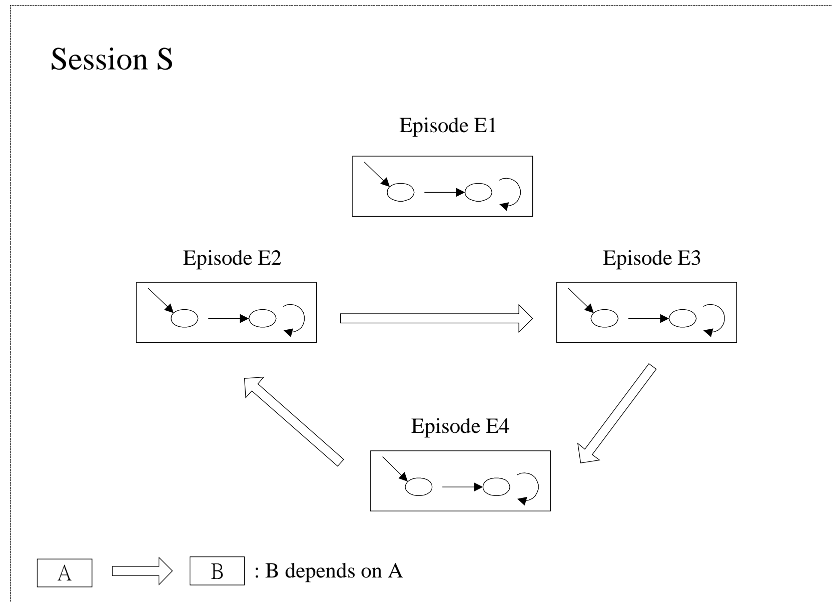


Figure 5. Dependency graph between episodes

2.4 Limitations and Issues

WNDL is designed based on some assumptions that may be violate in some cases.

WNDL assumes all target data can be extracted from fetched Web pages.

WNDL executor relies on a *SoftMealy* extractor to extract data from fetched Web pages. Unfortunately, *SoftMealy* extractor has its limitation and certainly cannot guarantee to achieve this. In some cases, multiple rules can be applied on the same Web page to jointly produce data that cannot be extracted by one single extraction pass. WNDL executor takes advantage of this trick to conquer the one-pass[1] nature of *SoftMealy* extractor. By using different extraction rule in each pass, and then concatenating or joining the result from each result data set, we can make the output data look like coming from a multiple-pass extractor.

In other cases, if the extractor fails to extract key information for execution to continue, for example, a URL or a `<Form>` block, WNDL wrapper will fail to wrap this site. WNDL executor expects that all required information such as parameter values for CGI query could be extracted directly from Web pages. However, there are cases where the extractor cannot achieve this because some values are determined

[1] The current SoftMealy extractor scans the extracting document one pass only so that it cannot produce overlapped data records.

outside the form block. For example, for pages with *JavaScript* functions to dynamically generate the required parameter values. These values will be determined after the execution of that function during run-time when certain event occurs such as when the user clicks a button on the Web page. Without bundling a *JavaScript* interpreter[1], the current system cannot handle this case. Since the definition of WNDL is not implementation dependent, it is not difficult to upgrade the whole system to a more powerful extractor, for example, a next generation *SoftMealy* extractor or an extracting subsystem with *JavaScript* interpreting capability.

WNDL assumes the values of the variables referring to the same real world object are equal to each other.

The data contained in Table 1 exemplifies the inconsistency of string values referring to the same real world object in practical applications. The same books have different listed titles from those two information sources even though each book has only one title printed on its cover. The style of publish date representation is also different in these two Web sites. Moreover, the same book has two publish date in these two Web sites.

Normalizing various representations of the same real world object to a canonical form can solve the problem partially but not completely. As discussed in previous work [6], "Determining if two name constants should be considered identical can require detailed knowledge of the world, the purpose of the user's query, or both." In that work, they proposed a mechanism using statistical information retrieval techniques to explicitly measure the similarity between terms in information integration domain and got impressive results. But the techniques they adopted are only applicable to English documents and are not suitable for environments that are full of documents in other languages.

In *Soft Lego* system, we experimentally adopted an immediate and direct way to aid this problem. In addition to a configuration file, another file that specifies aliases of terms can be optionally fed to the executor. Terms in an equivalent class defined in that file will be treated as identical during the evaluation process if string comparison operations are taken. However, we found that defining the various aliases of terms found in pages belong to an application domain exhaustively is too tedious and impractical for larger domains with a large amount of term inconsistencies. So we remove this feature in the current version. Handling this problem is by itself a difficult

---

[1] Or interpreters for other script languages that can be used in Web pages, such as VBScript.

research topic. We decide not to address this problem in this thesis. Handling process is left to the upper layer application that calls our system.

WNDL assumes that each data cell has at most one value.

In some cases, multiple values appeared in one data cell. Consider the book data in Table 1 again. These books have multiple authors. Though the current extracting subsystem is capable for identifying multi-value attributes and extracting them, when the value of a cell is no longer a string and become a set of strings, these values can have many kinds of relationship between each other. Hence, comparison between multi-value attributes is very ambiguous. We decide to assume that every attribute have at most one value and keep the system works as a relation evaluation system. When multi-value is necessary, defining additional attributes may be required.

WNDL may not be naturally extended to fully support XML.

The number of XML documents is believed to increase gradually in the future. Although when we designed our system, we focused on Web navigations with HTML documents, our system can still cover part of XML documents with a replaced data extractor. However, adapting our system to include the capability of fully processing XML documents may require substantial modification.

First of all, XML documents have a tree-structured data model that may not be expressed in relational data model that we adopted, as the above point indicated. Second, links in XML document has larger freedom than HTML links. XML allows its users to define their own tags. Therefore, any kind of linking element may be defined. Interpretation of these linking elements is left to the processors of these XML-based languages. Hence, navigation of XML documents cannot be totally generalized. We can only extend our system to include the processing of generally acknowledged standards such as *XLink* [28], which is suggested by W3C.

However, XLink links can be bi-directional, involve multiple resources, other complex links can also be created. The current *one-pass* WNDL executor cannot handle these cases. The relationships between multiple involved documents cannot be specified by WNDL. As a result, the processing of XLink links does not conform to the evaluation of WNDL programs and cannot be incorporated into our system naturally.

Chapter 3

WNDL Executor Implementation

3.1 Architecture and Implementation of WNDL Executor

The same as *Software Lego* executor discussed previously, a WNDL wrapper is composed of three subsystems: *executor* kernel, *SoftMealy extractor*, and *page fetcher*. Figure 6 shows the relationship between them and the order of execution steps.

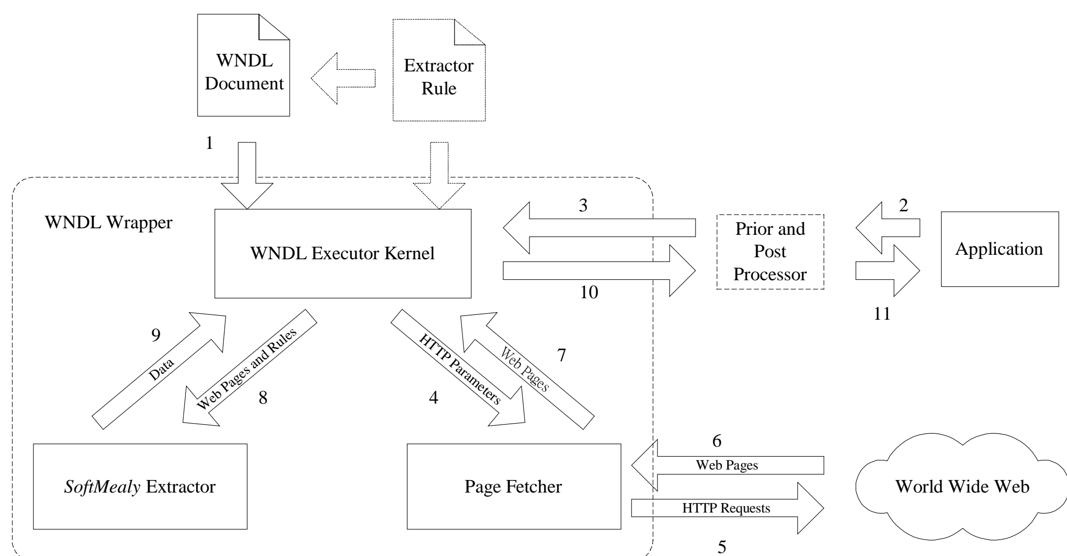


Figure 6. WNDL Executor Architecture

Every WNDL wrapper takes a WNDL document as a configuration file to wrap a logical Web site. This file defines the behavior of this wrapper. The executor handles data flows in this system and evaluates relations of data according to the algorithms listed in the next section. During the execution, it reads static information and variable binding information specified in the configuration file to complete HTTP requests.

The page fetcher abstracts HTTP connections to higher level interfaces to the executor. The page fetcher can handle HTML and HTTP features including *form* element parsing, *get* and *post* HTTP connections, *cookies*, *timeout*, HTTP *authentication* and *mal-formed* URL handling. The page fetcher transforms the parameters received from the executor to low level, executable HTTP requests. After actually obtaining a Web page from a target Web server, it sends this page back to the executor directly.

After obtaining a Web page, the executor will feed this page and the corresponding extraction rule to the extractor for data extraction. One page may go through this process multiple times if there are more than one extraction rule required for this page. Extracted data will be returned to the executor for further processing.

The other noticeable block in Figure 6 is an optional *prior and post processor* block. WNDL executor is a general-purpose wrapper and does not take operations specific to data from one information source. In cases when the upper layer application accesses Web with more than one WNDL wrappers, it may be necessary to insert a piece of code between WNDL wrappers and the application. This processor is responsible to transform queries from application to the form appropriate to the specific source and perform post processing over the data into a canonical form to fit the need of the application.

Our system is implemented in Java language as a class library. An API is defined to maximize its utility. A set of exceptions is defined and leaves most of exception handling task to be determined by the application. In our implementation, we take advantage of the following Java packages: *Java 2 SDK* [22] as the basic Java development kit, *JAXP* [21] for XML parsing, *HTTPClient* [23] for HTTP connection, and *HTMLStreamTokenizer* [3] in HTML form processing.

3.2 WNDL Program Evaluation Procedure

3.2.1 A Brief Introduction to Datalog

The evaluation procedure of WNDL programs is closely related to *datalog* [24] evaluation procedure. We present a brief introduction to datalog here. Datalog is a version of Prolog that is suitable for database systems. It differs from Prolog with the following two respects.

- Datalog allows only variables and constants as arguments of predicates and does not allow function symbols as arguments.
- Viewpoint to meaning of predicates in datalog is different to Prolog and in some cases it deviates from that in Prolog.

Datalog programs are composed of atomic *formulas* that are *predicate* symbols with a list of arguments. For example: $P(A_1, A_2,\ldots, A_n)$ where P is the predicate symbol. Every *attribute* A_i can be either a variable or a constant. Each predicate denotes a relation. Arithmetic comparison predicates such as >, <, and so on, can also be used to construct atomic formulas, they are called built-in predicates in datalog context. A predicate whose relation is stored in a database is called an *extensional database* (EDB) relation; the one defined by logical rules is called an *intentional database* (IDB) relation.

A *literal* is an atomic formula or a negated atomic formula. A *clause* is a logical *OR* of literals. A *Horn clause* is a clause with at most one positive literal. Hence, a Horn clause is in one of the three cases.

- fact – a single positive literal
- integrity constraint – negative literal(s) without positive literal
- rule – a positive literal with one or more negative literal[1]

[1] Horn clauses in this category are logically equivalent to logical implication with one positive literal as the antecedent.

A datalog program is a collection of rules. In datalog, rules are written in Prolog style such as this example datalog program.

```
Chain(X, Y) :- Link(X, Y).
Chain(X, Z) :- Chain(X, Y) & Chain(Y, Z).
```

Code 7. An Example Datalog Program

In this short example datalog program, there is an IDB relation, `Chain`, and an EDB relation, `Link`. Predicate at left hand side of the "if" symbol, ":-", in a rule is called the *head* of that rule, and predicates at the right hand side of a rule is called the *body* of that rule. In datalog, head is the logical consequence of body and each atomic formula in body is called as a *subgoal*. The dependency relationship between predicates can be depicted by dependency graphs.

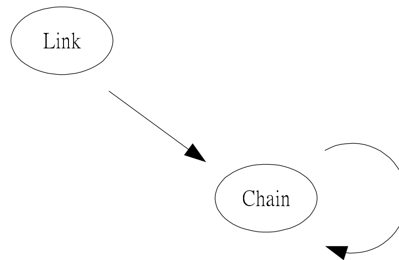


Figure 7. Dependency graph of the predicates in Code 7

When cycles exist in dependency graph, the involved predicates are recursive. In the example above, `Chain` is such a recursive predicate. Recursion is the main strength of datalog over other languages. Suppose we have an EDB with the following contents. *{(A, B), (B, C), (C, D), (E, F)}* Via the program in Code 7 and several iterations of evaluation, we can finally obtain the conclusion that `Chain(A, D)` holds.

Consider computing a datalog program with EDB relations $R_1, R_2, \ldots, R_k$ and IDB relations $P_1, P_2, \ldots, P_m$. For each i, $1 \leq i \leq m$, the set of provable facts for predicate p_i that corresponds to IDB relation P_i can be expressed by this assignment.

$$P_i := \texttt{Eval}(p_i, R_1, \ldots, R_k, P_1, \ldots, P_m)$$

In this assignment, `Eval` is the union of the evaluation result from each rule with predicate p_i as head. Then there is an immediate algorithm that solves recursive datalog programs.

```
begin
    for i = 1 to m
        Pi := Ø;
    end of for
    do
        for i = 1 to m
            Qi := Pi;  // save values of Pi
        end of for
        for i = 1 to m
            Pi := Eval(pi, R1,…, Rk, Q1,…, Qm);
        end of for
    while Pi != Qi for all i, 1 ≤ i ≤ m
    return Pi's;
end
```

Algorithm 1. Simple evaluation algorithm of datalog

This algorithm considers all tuples of all relations in all iterations and is obviously inefficient. It can be proved that newly generated tuples during each iteration only depend on the tuples that were generated in last iteration [24]. The following algorithm incrementally evaluates a datalog program and is more efficient than the previous one. In this algorithm, `EvalIncr` generates tuples in this way: when evaluating each rule with P_i as head, the evaluation will run in a loop with m iterations. In each iteration, one full relation is replaced by an incremental relation ΔQ_j.

```
begin
    for i = 1 to m
        ΔPi := Eval(pi, R1, …, Rk, Ø, …, Ø);// the same as the one in
        Pi := ΔPi;                          // Algorithm 1.
    end of for
    while Δpi ≠ Ø for all i
        for i := 1 to m
            ΔQi := Δpi;
        end of for
        for i := 1 to m
            Δpi := EvalIncr(Pi, R1, …, Rk, P1, …, Pm, ΔQ1, …, ΔQm);
            ΔPi := ΔPi - Pi;
        end of for
        for i := 1 to m
```

```
            Pi := Pi ∪ ΔPi;
    end of while
    return Pi's;
end
```

Algorithm 2. An algorithm that incrementally evaluates datalog programs

These two algorithms work rely on two assumptions.

1. EDB relations are *constants*. That is, contents in each EDB relation will not change during evaluation procedure.
2. All operations are *monotone*. This means no negated atomic formulas are allowed.

WNDL executor evaluates WNDL programs with a procedure based on Algorithm 2. It also relies on the two assumptions listed above.

3.2.2 Executor Variable Binding and Propagation

WNDL program evaluation procedure uses *relations* to represent the data found on Web pages. In this section, we will explain the variable binding and the relation evaluation of WNDL.

In this discussion, we adopted the following notation: $\bar{X}$ denotes a tuple of one or more relation attributes, X_1, X_2,…,X_n.

A WNDL program can be conceptually represented as the following general datalog program.

1. $Output(\bar{S}_{IO}, \bar{S}_O)$:- $Session(\bar{S}_I, \bar{S}_{IO}, \bar{S}_O, \bar{S}_{Tmp})$. where $\bar{S}_I \cup \bar{S}_{IO} \cup \bar{S}_O \cup \bar{S}_{Tmp} = \cup_i \bar{E}_i$
2. $Session(\bar{S}_I, \bar{S}_{IO}, \bar{S}_O, \bar{S}_{Tmp})$:- $E_1(\bar{E}_1)\&E_2(\bar{E}_2)\&\ldots\&E_i(\bar{E}_i)$.
3. {$Session(\bar{S}_I, \bar{S}_{IO}, \bar{S}_O, \bar{S}_{Tmp})$:- $E_1(\bar{E}_1)\&E_2(\bar{E}_2)\&\ldots\&E_i(\bar{E}_i)\&$
   $Session(\bar{S}_I, \bar{S}_{IO}, \bar{S}_O, \bar{S}_{Tmp})$.} // *optional*
4. $E_i(\bar{E})$:- $R_1(\bar{e}_1, \bar{n}_1)\&R_2(\bar{e}_2, \bar{n}_2)\&\ldots\&R_j(\bar{e}_j, \bar{n}_j)$. where $\bar{E} = \cup_j(\bar{e}_j \cup \bar{n}_j)$
5. $R_j(\bar{e}_j, \bar{n}_j)$:- $Fetcher(\bar{e}_j, WebPage, \bar{P}_j)\&$
   $Extract(WebPage, \bar{n}_j, Rule)$.

Code 8. WNDL program in datalog representation

In clause 5 of Code 8, request *j* is represented as a *relation* R_j that can be denoted as $R_j(\bar{e}_j, \bar{n}_j)$ where $\bar{e}_j$ represents the variables that are defined in the EdgeInput block of the edge that is involved in request *j*, and $\bar{n}_j$ denotes the variables that are defined in the Schema block of the node that is involved in request *j*. During the execution, $\bar{e}_j$ and constant parameters that are set in WNDL script will be passed to *page fetcher*, a *WebPage* will then be returned. This *WebPage* and its corresponding extraction rule will be passed to extractor, and then $\bar{n}_j$ will be returned. In actual execution, attributes in $\bar{e}_j$ and $\bar{n}_j$ will be renamed to session variables according to the binding specifications by the VarBind elements in request *j*, as expressed in the clause 4.

In clause 2 and 3, $\bar{s}_I$, $\bar{s}_{IO}$, $\bar{s}_O$, and $\bar{s}_{Tmp}$ represent session variables with the types, *In*, *InOut*, *Out*, and *Tmp*, respectively. These session variables are declared in SessionVar elements in WNDL scripts. The optional datalog clause 3 presents when there are dependency specifications between episodes in WNDL scripts. Finally, the output will be instantiated to relation `Output` as expressed in clause 1.

3.2.3 WNDL Program Evaluation Procedure

WNDL programs are closely related to datalog programs without negation as introduced in section 3.2.1. They have the following corresponding relationship: A WNDL *session* corresponds to a datalog program. A WNDL *episode* corresponds to an IDB relation, and *requests* correspond to EDB relations. *Loop* is a special structure that does not exist in datalog. However, there are functional overlaps between loops and episodes that mutually depend on each other. A set of such episodes can be interchangeable with a loop structure. To keep the logic cleanness of an episode's chain nature and help to organize a WNDL program well, both of these two features are provided by WNDL.

Algorithm 3 is the actual evaluation procedure for a WNDL program that is defined by a Session element. In this pseudo code, data are denoted by italic names and relations are named with a capital character prefix. Relation E_i represents current data set of episode e_i. Relation *I* represents the value of session variables with type *In* or *InOut*. Relation *O* represents session variables with type *Out* or *InOut*. There are several primitive functions that are denoted with bold face. Their functionalities are listed in Table 2.

join	Join the argument relations with *natural join*.
projection(*E*, *s*)	Project relation *E* to relation scheme denoted by *s*.
intersection(E_1, E_2)	Generate a relation with tuples come from E_1 so that when this tuple is projected to E_2's scheme, it is a tuple in E_2. (Scheme of E_2 is a proper subset of the scheme of E_1)
fetch_data	Issue a sequence of HTTP requests, get the pages, and extract data from these pages.
generate_HTTP_request	According to tuples of edge parameter relation and other information to generate full information for HTTP requests

Table 2. Primitive Functions Used in Algorithm 3

The execution of Algorithm 3 starts with the `execute_session` function, which executes a WNDL program defined in a session with *m* episodes. It calls `execute_episode` to execute episodes in an infinite loop until no more new data can be found. If there is no mutually dependent episode exists, only one iteration will be performed.

The execution of episode e_i starts with a *reference relation*. It is formed by the common attributes between the episode that is being executed, relation *I*, and its prerequisite episodes. Further extracted data during the execution of episode e_i will be joined with this reference relation to form the final E_i in certain iteration.

`generate_request_and_filter` generates a temporary relation of tuples of edge variables, that is $\bar{e}$ in the last section, each tuple will be used to generate a HTTP request later by `generate_HTTP_request`. *Filter* is a relation that will be used to filter out invalid tuples that are extracted from each page.

```
execute_session(E1, E2, …, Em) // Execute a session with m episodes.
begin
    do
        for i = 1 to m
            ΔRefi := generate_ref(i, E1, E2, …, Em);
            ΔRefi := ΔRefi – Refi;
            Refi := Refi + ΔRefi;
            if ΔRefi = Ø and Ei ≠ Ø
                continue;
            end of if
            execute_episode(Ei, ΔRefi);
        end of for
    while for i to m ΔEi ≠ Ø;
    return projection(join(E1, E2, …, Em), O);
```

```
end
genenerate_ref(i, E1, E2, …, Em) // Generate the reference for episode i
begin                            // from all of the m episodes.
    Ref := join(I with Ej's from j = 1 to m
                if ej is a prerequisite of ei);
    projection(Ref, Ei);
    return Ref;
end
```

```
execute_episode(E, Ref) // Execute episode e that is represented by
Begin                   // relation E with reference relation Ref.
    E := join(E, Ref);
    for each content element s defined in e
        if s is a loop
            execute_loop(s, E)
        end of if
        if s is a request
            execute_request(s, E)
        end of if
    end of for
end
```

```
execute_loop(l, E) // Execute loop l with data relation E.
begin              // This function does not handle self-loop.
    if l is a self-loop
        return;
    endif
    for i = 0 to l.max
        ΔE := ΔE - E;
        if ΔE = Ø
            break;
        end of if
        E := ΔE
        for each step s defined in l
            if s is a loop
                execute_loop(s, ΔE);
            end of if
            if s is a request
                execute_request(s, ΔE);
            end of if
        end of for
        i := i + 1;
    end of for
end
```

```
execute_request(r, E) // Execute request r with data relation E.
begin
    Requests, Filter := generate_request_and_filter(r, E);
    Result := fetch_data(Requests);
    Result := intersection(Result, Filter);
    E := join(E, Result);
end
```

```
generate_request_and_filter(r, E) // Generate HTTP request parameters
begin                             // for request r with relation E.
    Req := projection(E, r.edge);
    Requests := generate_HTTP_request(Req);
    Filter := projection(E, r.edge ∪ r.node);
    return Requests, Filter
end
```

Algorithm 3. WNDL Evaluation Procedure

The whole process of evaluation of a WNDL program follows the algorithm listed above with an exception when a page with “next pages” or a loop on DWM involves only one node. “Next pages” appears frequently when issuing a query to a CGI form. It is necessary to follow the chain of these inter-linked pages to obtain all of the query results. Executor will concatenate the results rather than join the results, which is the procedure for other kinds of loop. WNDL executor handles this kind of pages with several heuristics to terminate this loop. These heuristics work rely on a *link* to next page as shown in Figure 8.

Figure 8. A next page link that can be found in a return page from *Yahoo!*.

Our method works no matter it is a button, an image, or a link text as long as it has an underlying *static link* or a HTML *form*[1]. Executor requires schema of this node to include this link. The self-looping edge has to specify this link with its URL. Recall that the example in chapter 2, edge 3 is such an edge. The variable `url_next` will be extracted in node B and will be used as the URL of edge 3 to obtain the next page, which also belongs to node B page class. The process will repeat as a loop until one of the following conditions occurs.

1. Maximum iteration number specified in the WNDL document is reached.
2. No next-page link can be found in current iteration.
3. The next-page link extracted in current iteration has been visited in previous iterations.
4. No more new data can be found in current iteration.
5. A failure occurs when the executor applies this node’s extraction rule(s) to the page obtained in current iteration.

Condition 2 is applicable to most of the cases in our experiments. Condition 3 and 4 are designed to prevent the executor from visiting the “*previous-page link*” that can be found in many *last pages*. Condition 5 is for the cases where the last page returned by the CGI is in a format different from data pages and says something like “Sorry, we cannot find more results”. These heuristics are induced from dozens of Web sites of our case studies and work fine in our experiments.

[1] The other limitation is when the parameters of the form can only be determined by running a program like *JavaScript* at browsing time.

3.3 Exception Handling

The World Wide Web is an unreliable and unpredictable environment. Many exceptions can happen. For example, network traffic may be jammed, Web servers may be temporarily down, page structure of a site may be changed, and Web page layout may be modified. All Web applications are required to handle some or all of these exceptions or they may fail frequently.

WNDL allows its users to specify the number of retry attempts and the time interval between each attempt. The time intervals are equal to the time bounds of timeout events. These settings can prevent the executor from waiting forever for a stalled request to complete. When other failures occur, we simply throw a *Java* language exception to the upper layer application indicating what error happened and leave the remaining error-processing task to the application. The exceptions that the executor can report to the application are classified into four categories.

- Invalid WNDL scripts. For example, a required attribute is missing, referring to an undefined variable, etc.
- HTTP failures.
- Failed extraction passes.
- General Java language exceptions, which is not supposed to happen.

We made such a design decision that the executor will indicate the error without further reporting actions because generally, there is no way to save the execution if one of these exceptions occurs. So we leave the handling to be determined by the application. The upper layer application may then terminate immediately or try another navigation plan.

Chapter 4

Experiments and Evaluations

In the previous two chapters we described the design and implementation of our WNDL wrapper. Since our claim is to develop a way for rapidly building wrappers for information mediators on the Web, in this chapter, we conducted a series of experiments to evaluate two critical aspects of the WNDL wrapper. First is to show the *coverage* of our WNDL wrappers in section 4.1. Second, we will show the *simplicity* of configuring WNDL wrappers for a set of Web pages and the *generality across domains* of our approach in section 4.2.

4.1 Expressiveness of WNDL

In this part of our experiments, we want to show that WNDL is sufficiently expressive to cover Web sites in a wide variety. We decided to use WNDL wrappers to practically develop an application of integration of information from multiple Web sites. We picked a topic and build an experimental meta-search Web site in a specific domain. The topic of our domain is *job finding*. We chose our target Web sites from category lists of *Yahoo!.com* (5 sites) and *yam.com* (14 sites) without deliberate intention in picking sites that are especially suitable for WNDL wrappers. These web sites are listed in Table 3.

During the development of *Software Lego* and *WNDL* wrappers, we have conducted the experiments mentioned above and the experiment that will be discussed in next section. We already tested our wrappers and showed that they are capable of successfully wrapping these 40+ Web sites. These sites can be classified into several types. We will show the expressiveness of WNDL with example cases.

Name	Entrance Edge URL	Node	Edge
104Bank	http://www.104.com.tw/cfdocs/job1qry1.dbm	3	3[1]
37Job	http://www.37.com.tw/scripts/asp/fj_rs_new.asp	1	2
9999	http://www.9999.com.tw/m1201.asp	1	2
Career Builder	http://www.careerbuilder.com	2	3
Career Path	http://www.careerpath.com/service/cp/FindJob?	2	3
CTCareer	http://www.ctcareer.com.tw/findjob.exe	2	3
Career Web	http://www.cweb.com/jobsearch/jobsearch.cfm	1	2
Horse	http://www.job.com.tw/Horse/newHorse-right.asp	1	1
IlanJob	http://104.ilhg.gov.tw/jobdetail.asp	1	1
Job 4 free	http://www.job4free.com/findjob2.cfm	1	1
JobsDB	http://www.jobsdb.com.tw/TW/B5/default.asp	1	2
Kauhsiung	http://www.07job.com.tw/m1201.asp	1	2
Monster	http://jobsearch.monster.com/jobsearch.asp	1	2
NYC	http://site1.nyc.gov.tw/db/index.asp	3	4
Raritan	http://job.raritan.com.tw/fjobmain.cfm	1	1
Taichung	http://www.04job.com.tw/m1201.asp	1	2
TaipeiCity	http://163.29.128.6/OKwork/Data_Com/Company2.asp	2	2
WSJ	http://wsj.careercast.com/texis/wsj/wsjjobsearch/dosearch.html	1	2
YouCool	http://job.youcool.com.tw/career_c/search_jobs/s_by_disc_search.asp	1	2

Table 3. Web Sites in the WNDL Experiment

4.1.1 Chain and Loop

WNDL is capable of representing and fetching data from Web sites from as simple as simplex node chains to those as complex as long node chains with loop structures.

[1] The next page link of 104Bank is determined by a JavaScript function and cannot be preset. Thus our configuration in fact lacks the edge *4* in Figure 13.

A Simplex Chain of Nodes

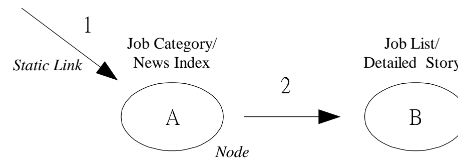


Figure 9. DWM diagram of *TaipeiCity* and *TTimes*

The page structure of *TaipeiCity* looks like a chain. WNDL executor will follow a static link via edge *1* to node *A*. It can find dozens of static links there and use these URL's on edge *2* to move on to node *B*. At node *B*, it can obtain the target data of this navigation episode. Data found on each node in the path can be carried on by the executor until the end of the execution of this episode. For example, in *TTimes*[1] that has a similar structure to TaipeiCity, some of our target data are stored in node *A*. Information found in node *A* and node *B* will all appear in the result data set.

A Long Chain with Loop

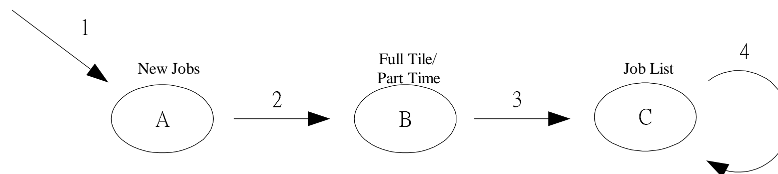


Figure 10. DWM diagram of *NYC*

The page structure of *NYC* is similar to the one shown in Figure 9 with an additional looping edge, edge *4*. Though edge *3* is not a form query, NYC distributes its data into multiple pages. Edge *1* is particularly remarkable because the purpose of sending a HTTP request via edge *1* is in fact only to trigger NYC's CGI program to start a user session[2] inside their Web server.

---

[1] TTimes (http://www.ttimes.com.tw/index_sub1.html) is a site providing on-line news, which is one of our test Web site but is not in our job-finding corpus list.

[2] NYC's CGI program responds the same static URL request with different results according to this user session mechanism. This user session state will be terminated if the Web user has no interaction with the server longer than three minutes.

4.1.2 Arbitrary Number of Data Extraction Passes

The design of WNDL makes it possible to take arbitrary number of data extraction passes on a single page. Generally there is exactly one extraction pass over a single Web page, but to extend the number of pass to an arbitrary number we can provide the following functionalities that will be explained by examples.

A Zero Extraction Pass

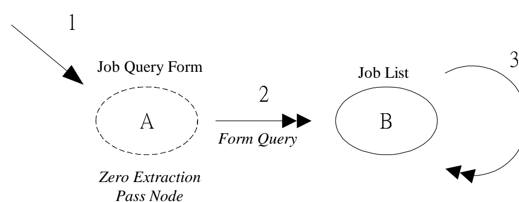


Figure 11. DWM diagram of *CTCareer*

CTCareer is an example with zero extraction pass in our corpus. What we want to do is placing queries via edge *2* to fetch the target information on a multiple page node *B*. But in this case, edge *2* cannot be an entrance edge since HTTP clients need to obtain a cookie object when request pages, which belong to node *A*. Thus, though we do not need any information on node *A*, we still need to visit node *A* once in order to be able to send our queries. Hence, let node *A* to be an *empty node* without any data and extraction pass can make the process more efficient. The CGI query form is in fact on node *A*, but since we need to bind CGI query parameters during run-time, we configured the form in the definition of edge *2* completely with full set of input variables and do not require to extract the HTML form on node *A*.

A Two Extraction Passes

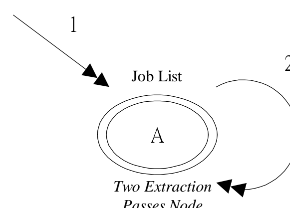


Figure 12. DWM diagram of *JobsDB*

JobsDB is a site that requires two passes of data extraction over a single node. In this site we plan to send a query to a CGI program to get result pages in page class *A*, we configured edge *1* as an entrance edge to send queries to that CGI program. A problem occurred in extracting data from node *A* with *SoftMealy* extractor. The contents of page class *A* are wholly contained within a HTML form block that includes all of the query results as well as the *next-page link*[1]. This is another bad HTML programming practice that can be found on the Web. The scope of the data area is overlapped with the scope of next-page link in this page. This situation causes the extractor fail to extract query results and the next-page information at the same time, which is required by the operation of executor. Allowing multiple extraction passes over one node makes it possible to solve this problem. We can make following configuration in the WNDL script so that in the first pass, the extractor will extract query results and will extract the next-page link in the second pass. The executor will then combine the information extracted by the two passes to make it appear as produced by only one pass.

A Three Extraction Passes Case

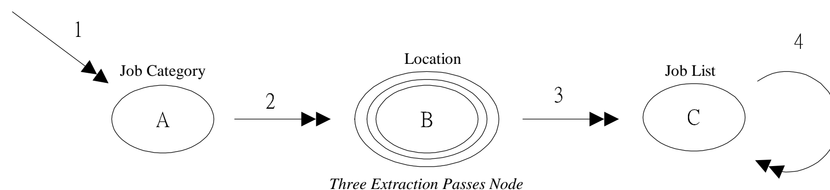


Figure 13. DWM diagram of *104Bank*

There is a node that needs three passes of extraction in *104Bank*, as illustrated by node *B* in Figure 13. This node requires multiple extraction passes for another reason. This is a page with sixty-eight HTML buttons with similar format. Each one is contained in a form. Our purpose of this navigation is to pick only three buttons from all the sixty-eight buttons and send the corresponding CGI queries. The problem is the CGI query parameters within the form change everyday so that we cannot identify one or several buttons out of all of these buttons with a preset form text for matching. However, the flexibility provided by WNDL makes it possible to take three passes of extraction over the node *B* pages. Each one selects one button for further navigation.

[1] In this case, the next-page link is nearly the whole page.

4.2 Utility of WNDL Wrapper

In this section, we show the utility of our WNDL wrapper with empirical results. First, we show how easy it will be to wrap a Web site with WNDL. Second, we will show the generality of WNDL wrappers.

4.2.1 Configuring a New Wrapper

To assess the simplicity of configuring a wrapper of a new Web site, we released the previous version of our work, *Software Lego* software package to students who took the course, "Principles of Internet Agents" at the National Chiao-Tung University as their term project material in Jan. 2000.

These students were divided into eleven groups. Each was formed with two students. Every group was assigned an application domain and was asked to build a shopping bot application that integrates several Web sites based on *Software Lego*. Eight days after the release date of that software package, only one group failed to complete the task. All of the remaining ten groups successfully wrapped three to five Web sites. Moreover, there was one group completed a cellular phone comparison-shopping site with four connected Web sites within only one night. They defined a canonical schema according to their application domain and use *Software Lego* to fetch data from the connected Web sites corresponding to that data schema. All of these groups built a Web site interface for their shopping bot agent. Users of their system enter a query and then the Software Lego wrapper will be called to fetch data, combine data and return the result to their system. Those data may then be post processed in their program. Finally search results from multiple Web sites are integrated in a uniform outlook and are presented to the users. Table 4 lists the Web sites used by these ten groups[1].

[1] There were only twenty-seven different sites because multiple groups may use the same Web site.

Site Homepage URL	Domain
http://www.books.com.tw	Book
http://www.kingstone.com.tw	Book
http://www.silkboook.net	Book
http://www.longshine.com.tw	Book
http://www.ylib.com.tw/home.asp	Book
http://www.myjob.com.tw	Job
http://www.jobs.com.tw	Job
http://www.adhot.com.tw	Job
http://www.jobbank.com.tw	Job
http://www.job4free.com	Job
http://www.kingnet.com.tw	Job
http://job.Raritan.com.tw	Job
http://www.moea.gov.tw	Job
http://www.ezjob.com.tw	Job
http://www.ctcareer.com.tw	Job
http://www.funny.com.tw	DVD/VCD Software
http://www.dvdmall.com.tw	DVD/VCD Software
http://www.3fdvd.com	DVD/VCD Software
http://www.hantop.com.tw	Computer Hardware
http://leapnet.jeya.com.tw	Computer Hardware
	Home Appliances
	Cellular Phone
http://www.shoppingguide.com.tw	Computer Hardware
http://www.answer.net.tw	Cellular Phone
http://www2.seeder.net	Cellular Phone
http://www.taconet.com.tw	Cellular Phone
http://store.acer.net	Home Appliances
http://www2.any.com.tw	Home Appliances
http://dmd.pros.com.tw	Home Appliances

Table 4. Twenty-seven Web sites that were used in the NCTU experiment

After that, we asked one colleague in our lab who did not participate in this project and was not familiar with the WNDL wrapper to configure wrappers of job-finding Web sites. He successfully configured wrappers of thirteen Web sites as well as trained dozens of *SoftMealy* extractor rules required for these sites in one week. He also set up a demo Web site that functions roughly the same as those NCTU term projects. These empirical results should provide evidences that WNDL wrappers can be configured in a short time and are capable of wrapping Web sites in a wide variety.

4.2.2 Generality Across Application Domains

When WNDL and its executor were designed, we did not take any advantage of domain related information. We expected our solution is domain independent. The NCTU experiment also coincides with this presumption.

The Web sites used by those students come from six different domains. On-line booksellers, job search engines, video software stores, computer hardware stores, cellular phone shops and home appliance stores. These domains differ a lot to each other without common background knowledge or structure/format of Web sites. This experiment suggests that the generality of WNDL can be across various application domains.

Chapter 5

Related Work

In this chapter, we will introduce previous work related to this thesis. Section 5.1 introduces three languages that can also be used to describe automation of Web interactions. Another work that also addresses Web navigation problem is included in section 5.2.

5.1 Web Automation Languages

Because of the practical need of automation of Web access, several other researches were also initiated to try to develop a way to automate interactions with the Web. Coincidentally, they all result in defining a language that describes Web interactions.

5.1.1 Web Interface Definition Language (WIDL)

In terminology of WIDL [25], a *service* is a request/respond interaction with the server. WIDL was designed to capture the details of the interfaces to services. Like WNDL, WIDL is also an application of XML and includes definitions of a set of XML elements. It is also a script language that will be interpreted during run-time.

WIDL is a language that is similar to edge definitions of WNDL. The details of obtaining a Web page such as URL, form parameter setting, and HTTP methods are

also represented by XML elements/attributes in WIDL. Unlike WNDL, WIDL does not define or determine a mechanism for accessing document data. It relies on an object model referencing mechanism to access data on Web pages. WIDL also comes with a variable system and variable value binding definitions such that WIDL can also handles dynamically generated URL and CGI program parameter values. Moreover, Web page contents matching mechanism is used to determine the *condition* of the binding result. When binding failure occurs, rebinding is possible. On the contrary, WNDL use the result of extraction to determine whether a binding attempt succeeds.

The primary difference between WIDL and WNDL is the aims they were designed to achieve. WNDL aims to capture *navigation* on the Web while WIDL focuses on defining an *interface* to Web services. Moreover, WIDL does not combine data, it only use data to facilitate further service interface access. Though there is a mechanism in WIDL called *service chain*, it cannot be compared to *Navigation Program* in WNDL. Service chain is a chain of services that works as the following description. Data obtained from each service are sent to next service as the input binding of that service. It can be used with systems when it is necessary to invoke multiple services in sequence to complete a transaction. But obtained data from certain service cannot be used to facilitate a service other than the *immediate successor* on the service chain. More than that, each service of WIDL associates at most one input binding and at most one output binding, which have global scope names, so recursive execution such as *next page* handling cannot be achieved by service chain of WIDL, which greatly minimizes the utility of their language in practice.

5.1.2 Web Language (WebL) and Hippo Core Language (HCL)

WebL [14][20] is another language that was designed for the automating of Web-related tasks. The authors tried to figure out a general programming model of the computation on the Web. This language features two distinguishing features, *service combinators* [5] and *markup algebra*. A *service* in WebL is an HTTP information provider wrapped in error-detection and handling code while a service combinator is an operator for composing services. On the other hand, markup algebra is for structured text searching on Web pages.

Several service combinators are available in WebL:

- *Sequential Execution*, `S ? T` – secondary service *T* will be consulted if *S* fails, otherwise result of *S* will be returned.
- *Concurrent Execution*, `S | T` – the result of the one that succeeds first will be returned.
- *Time Limit*, `timeout(t, S)` – acts as *S* but timeouts after *t* seconds.
- *Repetition*, `repeat(S)` – repeats S until it succeeds.
- *Non-termination*, `stall` – dummy combinator that does nothing[1].

These service combinators are applied on basic service `url` that represents a static link and *gateways* like `gateway get` and `gateway post` that represent CGI gateways that need arguments passing. These service combinators can also be used to combine with each other to produce joint effects.

Two types of text search can be done in *markup algebra* of WebL, *structured search* that can be used in searching with element names and *pattern search* that extracts all occurrences of a *regular expression* in the text of a page. Besides, *set operators*, *position operators*, and *hierarchical operators* are also defined for operations on page texts.

```
shopAmazon := fun(title, autorfirst, authorlast)
    books := [];
    params := [. .];
    params["author"] := authorfirst + " " + authorlast;
    params["authormode"] := "full";
    params["title"] := title;
    params["titlemode"] := "word";
    params["subject"] := "";
    params["subjectmode"] := "word";
    page := postpage("http://www.amazon.com/exec/obidos/atsquery/",
                    params);
    items := page.Elem("dd");
    every book in items do
        info1 := substring(book.Text(),
                '\w*([\^{}/]*)(/([\^{}/]*))?(/[\^{}\d]*(\d*))?')[0];
        info2 := substirng(book.Text(), 'Our Price: \$(\d*.\d*)');
```

[1] This combinator can be used in operations such as "***wait** and retry later*".

```
        if(size(info) > 0) and (info1[3] != "Audio Cassette") then
            books = books + [[.
            title = (page.Elem("a") directlybefore book)[0].Text(),
            link = (page.Elem("a") directlybefore book)[0].href,
            author = info1[1],
            type = info1[3],
            year = (select(info1[5], 2, 1) ? "N/A"),
            price = info2[0][1]
            .]]
        end
    end;
    books
end
```

Code 9. A WebL function excerpted from [14], which shops in *amazon.com*.

The code listed in Code 9 reveals the imperative nature of WebL that is similar to traditional computer languages. The authors of WebL focused on defining a *programming language* that encapsulates normal Web interaction operations and deals with error handling between several information sources that have identical contents. It was designed under the basic premise that providing programmers an easier way to write robust programs for automating Web access. Developers use the explicit language structures, service combinators, to compose their Web programs.

In contrast to WebL, WNDL serves as a behavior definition language for our WNDL wrapper, which is supposed to be used in an application requiring automated Web access as a Web data gathering subsystem. The problems addressed by WebL such as sequential execution, concurrent execution, and other error handling operations are left to be handled by the applications that take advantage of WNDL wrappers and are beyond the scope of this thesis.

The other noticeable point of WebL is their method for extracting data from Web pages, *markup algebra*. The *structured search* uses element names to identify data, though this mechanism is applicable to XML documents, it will be hard to be applied on HTML documents. The *pattern search* is another risky part, as we can see in Code 9, the regular expression fails easily even a minor change of page format occurs. Unfortunately, format of Web pages changes frequently and even pages in the same format are not consistent to each other. Code the regular expressions by hand is also a tedious and error-prone task.

HCL [7] is a language similar to WebL and addresses some drawbacks figured out by the authors of HCL and provides the following features:

- the unification of universal resource, text file and string types
- implicit URL fetching.
- parallel execution
- alternation execution

However, this work is still in an early stage, and their methodology is not clearly stated in their publication.

5.2 Work that Addresses the Web Navigation Problem

Marc Friedman, Alon Levy, and Todd Millstein [10] figured out two distinguishing characteristics of data on the Web.

1. Linked pairs of pages contain related data.
2. Obtaining the data from the site may require *navigation* through a particular path in the site.

In [10], they focused on introducing their data integration approach, GLAV (global-local-as-view) and its complexity analysis, which is shown to be as efficient as LAV (local-as-view). They took a pure *datalog* approach and suggest an algorithm that basically can be taken prior to user queries for generating a *navigation plan*.

Their work is focused on query plan generating and leaves the Web page fetching details in vain. They ignored the need of additional parameters for HTTP access such as *base* URL's for relative URL's, HTML form parameters in *hidden* type as well as other issues such as HTTP connection *timeout*, *retry*, *failed service*, etc. They leave these tasks completely to their datalog executor that operates only rely on parameters passed from datalog programs. According to our research, we found this approach is not only impractical but also makes adding features such as *fuzzy matching* or *string concatenation* to the system awkward.

Moreover, in their approach, it is necessary to put the whole navigational details such as "*next-page information*" or "*query keywords*", which are irrelevant to user queries, into their high-level *logic plan* and may overwhelm the cleanness of the meaning of logical plans. On the other hand, we focus on a wrapper that wraps a web site with a *user session* concept and frees high-level query plan from handling navigational details, which may be a better level of abstraction and serves as a better conceptual unit.

Chapter 6

Conclusion

6.1 Summary

In this thesis we presented a configurable information gathering software robot called WNDL executor that crawls the Web and its configuration language, WNDL that captures the navigation behavior on the Web.

The behavior of WNDL executor is determined by an XML-based script language called WNDL. This language was designed to capture the process of *navigation* and *data gathering* during a *user session*. WNDL executor can traverse both static and dynamic links on the Web while extracts and accumulates text data from the visited Web pages at the same time. Those data can then be used to facilitate further Web navigation if the information for navigation must be obtained during run-time.

The system was implemented in Java language as a class library. Applications call functions in this library and can be insulted from handling the detail and repeated tasks such as *HTTP access*, *data extraction*, and *data combination*. Our system is supposed to be able to benefit applications that need automated Web interactions. The feasibility of our idea was experimented on realistic Web applications. We also compared our approach with some contemporary related work at last.

6.2 Critiques and Future Work

In section 2.4, we already discussed several assumptions of WNDL that may be violated in real cases. However, there are some other problems we did not address in this thesis.

Web pages that require *client-side computations* like *Java Applet*, *JavaScript*, *VBScript* and the likes to determine URL's or CGI program parameters exist on the Web. Web pages in this category require an interpreter or executor on client side to process them. However, our system is not equipped with these interpreters. Therefore, in most of cases we cannot handle this kind of Web interaction. In other cases, since they are still using standard HTTP, we may manually determine the values of CGI parameters and set them as constants in WNDL scripts though this method does not really solve the problem. Although the unsolvable cases do not appear frequently[1], we need to find a better way to handle this problem. For example, we plan to incorporate a *JavaScript* interpreter into our WNDL executor in the future. This should help a lot since *JavaScript* is the most frequently encountered case.

HTTP failure handling is not completely done in the current work. WNDL executor and WNDL currently only provide *timeout* and *retry* mechanism. When other failures happen, we simply throw a Java language exception to the upper layer application indicating what error happened and leave the error-processing task to the application. However, it helps if *exception handling*, *alternative execution* or *conditional execution* can also be specified in WNDL. It is possible for WNDL to be extended to accommodate these features in the future.

Extension to WNDL executor in *join* operations to allow *fuzzy term matching* based on statistical measurement or some other techniques is also a possible future work. This feature is supposed to make WNDL solution to cover a much larger scope of cases.

Currently, WNDL scripts are composed by human programmers. Although its syntax is quite intuitive and is easy to learn, composing a WNDL script is still a laborious task and may require some level of familiarity with HTTP and HTML. Better Web automation can be achieved if we can develop a way to generate WNDL scripts from human Web browsing.

---

[1] In our experiments, we encounter only one unsolvable case, the *next-page link* of *104Bank*.

Bibliography

[1] Jose-Luis Ambite, Yigal Arens, Naveen Ashish, Craig A. Knoblock, Steven Minton, Jay Modi, Maria Muslea, Andrew Philpot, Wei-Min Shen, Sheila Tejada, and Weixiong Zhang. *The SIMS Manual Version 2.0,* December 22, 1997.

[2] Yigal Arens, Chin Y. Chee, Chun-Nan Hsu, and Craig A. Knoblock. Retrieving and Integrating Data from Multiple Information Sources. In *International Journal on Intelligent and Cooperative Information Systems* Vol. 2, No. 2. pp. 127-158, 1993.

[3] Arthur Do Consulting. *HTMLStreamTokenizer,* 1998.
http://www.do.org/products/parser/

[4] Greg Barish, Craig A. Knoblock, Yi-Shin Chen, Steven Minton, Andrew Philpot, and Cyrus Shahabi. TheaterLoc: A Case Study In Building An Information Integration Application. In *Proceedings of the IJCAI-99 Workshop on Intelligent Information Integration,* 1999.

[5] Luca Cardelli and Rowan Davies. Service Combinators for Web Computing. *DEC SRC Research Report* 148, June 1997.

[6] William W. Cohen. *Integration of Heterogeneous Databases Without Common Domains Using Queries Based on Textual Similarity*. SIGMOD 98, 1998.

[7] Richard Connor and Keith Sibson. *HCL – a language for Internet Data Acquisition*. Workshop on Internet Programming. ICCL '98, 1998.

[8] Rick Darnell. *Html 4 Unleashed: Professional Reference Edition.* Sams.net, December 1, 1997.

[9] Robert B. Doorenbos, Oren Etzioni, and Daniel S. Weld. A Scalable Comparison-Shopping Agent for the World-Wide Web. In *Proceedings of Agents '97 Conference*, 1997.

[10] Marc Friedman, Alon Levy, and Todd Millstein. Navigational Plans for Data Integration. In *Proceedings of the IJCAI-99 Workshop on Intelligent Information Integration,* 1999.

[11] Chun-Nan Hsu and Chien-Chi Chang. Finite-State Transducers for Semi-Structured Text Mining. In *Proceedings of IJCAI-99 Workshop on Text Mining: Foundations, Techniques and Applications*, 1999.

[12] Chun-Nan Hsu and Ming-Tzung Dung. Generating Finite-State Transducers for Semi-Structured Data Extraction from the Web. In *Journal of Information Systems* Vol. 23, No. 8, pp.521-538, 1998.

[13] Chun-Nan Hsu, Hung-Hsuan Huang, Elan Hung, and Chian-Chi Chang. *Wrapper Agent Kernel(Shopbot Toolkit) Version 0.1 the Manual*. Technical Report IM-IIS-00-001, Academia Sinica Institute of Information Science, Jan. 2000.

[14] Thomas Kistler, and Hannes Marais. WebL – a programming language for the Web. In *Computer Networks and ISDN Systems* (WWW7), vol. 30, April 1998.

[15] Craig A. Knoblock, Steven Minton, Jose Luis Ambite, Naveen Ashish, Pragnesh Jay Modi, Ion Musela, Andrew G. Philpot, and Sheila Tejada. Modeling Web Sources for Information Integration. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, 1998.

[16] Nicholas Kushmerick, Daniel S. Weld, and Robert Doorenbos. Wrapper Induction for Information Extraction. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, 1997.

[17] Simon St. Laurent. *XML A Primer.* MIS Press, 1998.

[18] Sean McGrath. *XML By Example: Building E-commerce Applications.* Prentice Hall PTR, 1998.

[19] Ion Musela, Steve Minton, and Craig Knoblock. STALKER: Learning Extraction Rules for Semistructured, Web-based Information Sources. In *Proceedings of AAAI-98 Workshop on AI and Information Integration*, 1998.

[20] Keith Sibson. *Service Combinators and WebL.* *http://www.hippo.cs.strath.ac.uk/papers/service_combinators_and_webl.html*

[21] Sun Microsystems, Inc. Java™ API for XML Parsing (JAXP), version 1.0.1, May 2, 2000.
http://www.javasoft.com/products/xml/index.html

[22] Sun Microsystems, Inc. *Java™ 2 SDK, Standard Edition, version 1.3.0,* May 8, 2000.
http://java.sun.com/j2se/1.3/

[23] Ronald Tschalär. *HTTPClient*, version 0.3-2, March 19, 2000.
http://www.innovation.ch/java/classes.html

[24] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems.* Computer Science Press, 1988.

[25] WebMethods, Inc. Phillip Merrick and Charles Allen. W3C Working Note. *Web Interface Definition Language (WIDL),* September 22, 1997.
http://www.w3.org/TR/NOTE-widl-970922

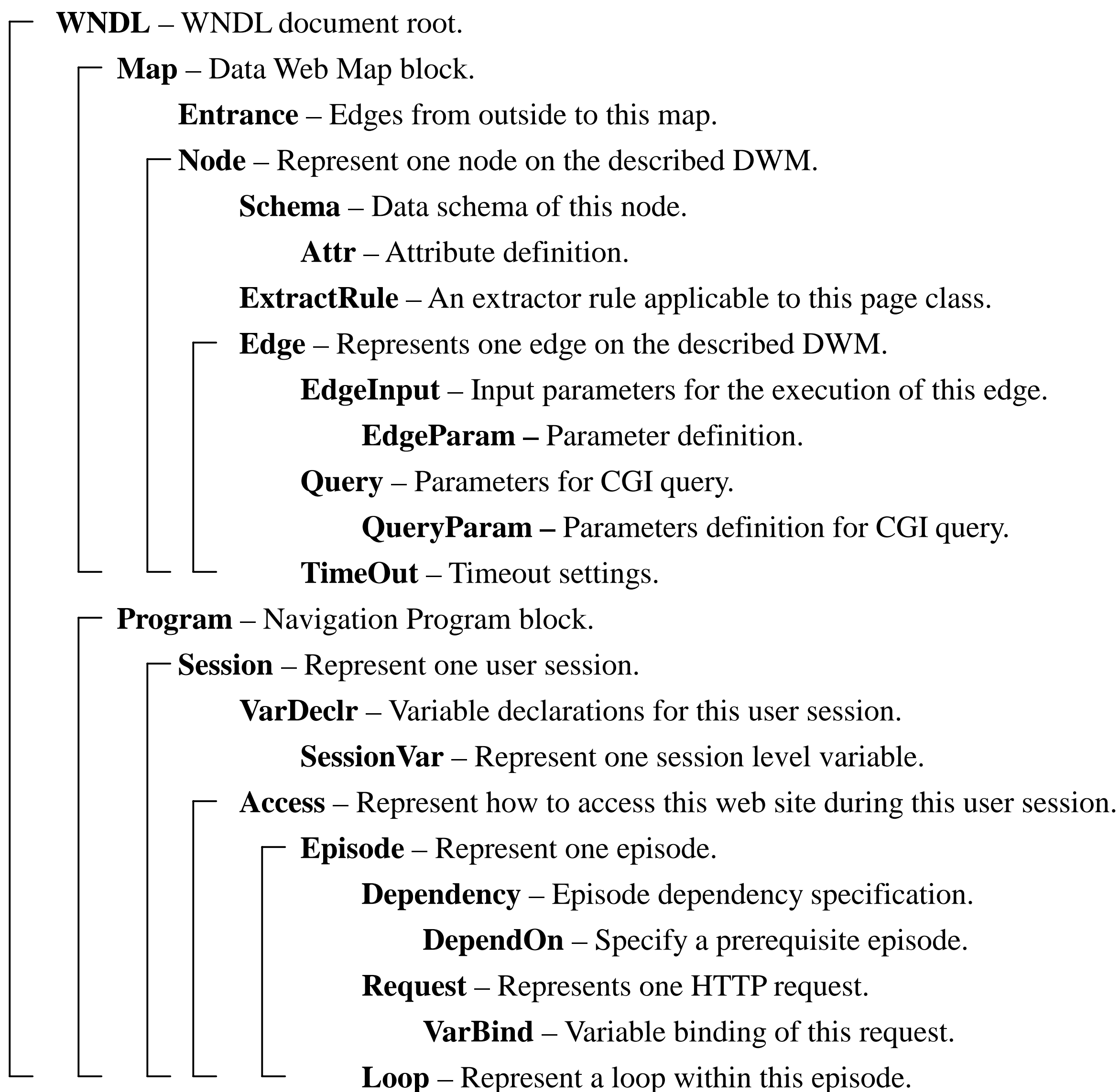
[26] World Wide Web Consortium (W3C) Recommendation. *Extensible Markup Language (XML) 1.0*, February 10, 1998.

[27] World Wide Web Consortium Working Draft. *Web Characterization Terminology & Definitions Sheet*, May 24, 1999.

[28] World Wide Web Consortium Candidate Recommendation. *XML Linking Language (XLink) Version 1.0*, July 3, 2000.

Appendix A. WNDL Reference

A.1 WNDL Element Structure



A.2 WNDL Elements Reference

Attribute Value Type:

Type	Description
String	A text string.
Number	A decimal number.
Enumeration	The value needs to be picked from a set of valid values.
URL	A universal resource locator.
Reference	A reference to a variable's value, the references are written as a dollar sign followed by a variable name. For example, "$variable_name". Because of the syntax of variable references, if the leading characters of one variable's value are dollar signs, each one of them needs to be replaced by double dollar signs, that is, "$$".

WNDL Element

This is the root element of a WNDL document tree. It contains the two primary elements of a complete WNDL description.

Attribute	Description	Type	Number	Default
Name	A name for this configured web site information source.	String	1	Null
Version	Identify the version of WNDL being used in this script.	String	0 or 1	Null

Contents	Type	Number
Map	Element	1
Program	Element	1

Map Element

This element represents a Data Web Map.

Contents	Type	Number
Entrance	Element	1
Node	Element	1 or more

Entrance Element

Specify the edges that do not belong to any node here. They are the entrance of this site, that is, they are directly accessible from outside the web site.

Contents	Description	Type	Number
Edge	Entrance edges of this web site. ※Only put the edges that have no source nodes here.	Element	1 or more

Node Element

The instances of this element represent one node on the WDM.

Attribute	Description	Type	Number	Default
Name	The name of this configured web information source. ※All node names need to be unique.	String	1	Null

Contents	Type	Number
Schema	Element	0 or 1
ExtractRule	Element	Unlimited
Edge	Element	Unlimited

Note: The valid number of ExtractRule depends on the existence of Schema. If there are no Schema element presents, no ExtractRule is allowed. If there is one and only one Schema element, at lease one ExtractRule element is required.

Schema Element

Data schema of this node is defined here. Although there is only one dimension of the data schema in this definition, the data schema can be multi-dimensional.

Contents	Type	Number
Attr	Element	1 or more

Attr Element

Attribute	Description	Type	Number	Default
Name	Name of this variable. ※All attribute names within one schema need to be different.	String	1	Null
TagFilter	HTML tag filter option: Valid values: *KeepAll* – Keep all HTML tags. *KeepLink* – Filter out all HTML tags except URL links of `<A>` tags. *NoTag* – Filter out all HTML tags.	String	0 or 1	*KeepLink*

ExtractRule Element

This element stores the extraction rule that will be used by *SoftMealy* extractor during execution. The extraction rule is generated by the rule learner program and is not meant to be readable for human. These rules are in plain text format and are necessarily contained within one CDATA element. The rule can also be contained in a separated plain text file. In this case, the path to that file is specified as the File attribute. If the File attribute and the contents of this element present at the same time, the rule contained in the CDATA element will be used.

Attribute	Description	Type	Number	Default
File	Path to the file that contains the rule	String	0 or 1	Null

Contents	Type	Number
CDATA	Text	0 or 1

Edge Element

This element stores the necessary settings for executor to get a web page.

Attribute	Description	Type	Number	Default
ID	Identifier of this edge, edge id is required to be unique.	String	1	Null
Type	Type of the target web page. Valid values: *Static* – A static link. *Dynamic* – A form.	Enumeration	0 or 1	*Static*
Base	Base URL of the target web page.	URL/Reference	0 or 1	Null
URL	URL of the target web page.	URL/Reference	1	Null
Method	HTTP method for getting the target page. Valid values: *Get*, *Post*	Enumeration	0 or 1	*Get*
Dest	Destination node name.	String	1	Null

Contents	Type	Number
EdgeInput	Element	0 or 1
Query	Element	0 or 1
Timeout	Element	0 or 1

EdgeInput Element

Parameters passed from outside to this edge that are necessary for opening the target URL.

Contents	Type	Number
EdgeParam	Element	1 or more

EdgeParam Element

This element represents one parameter.

Attribute	Description	Type	Number	Default
Name	Parameter name.	String	1	Null

Query Element

This element contains the parameters for CGI program query.

Contents	Type	Number
QueryParam	Element	1 or more

QueryParam Element

This element represents one parameter for this CGI query.

Attribute	Description	Type	Number	Default
FormInput	Parameter name for this CGI query	String	1	Null
Value	Parameter value, it can be a string constant or a variable reference.	String/Reference	1	Null

Timeout Element

Timeout setting during HTTP operations

Attribute	Description	Type	Number	Default
Wait	Time between each attempt of connection to the web server Unit: second	Number	0 or 1	∞
Retry	Times of retry.	Number	0 or 1	10

Program Element

This element contains possible user sessions for accessing this web site.

Contents	Type	Number
Session	Element	1 or more

Session Element

This element represents one user session.

Attribute	Description	Type	Number	Default
Name	Session name.	String	1	Null

Contents	Type	Number
VarDeclr	Element	1
Access	Element	1

VarDeclr Element

Contents	Type	Number
SessionVar	Element	1 or more

SessionVar Element

Variables those will be used during this user session's execution.

Attribute	Description	Type	Number	Default
Name	Variable name.	String	1	Null
Type	Type of this variable. Valid values: *In* – Input variables from application. *Out* – Output variables after execution. *InOut* – Appeared as both input and output. *Tmp* – Intermediate variables.	Enumeration	0 or 1	Out
Label	Another name of this variable. If label name presents, it will appear as the column name of the output.	String	0 or 1	Null
Value	Variable value. If this parameter is a constant, its value can be specified here. If there's a value attribute, this parameter is a constant. Otherwise, it is a variable.	String	0 or 1	Null

Access Element

This element describes the actual accessing part of this type of user session.

Contents	Type	Number
Episode	Element	1 or more

Episode Element

The element represents one episode contained in this type of user session.

Attribute	Description	Type	Number	Default
Name	Name of this episode.	String	1	Null
User	User name in the case that HTTP authentication is required.	String	0 or 1	Null
Passwd	Password for HTTP authentication. ※Required if User presents.	String	0 or 1	Null

Contents	Type	Number
Dependency	Element	0 or more
Request	Element	0 or more
Loop	Element	0 or more

Dependency Element

Specify that the data extracted from this episode need to depend on another episode.

Contents	Type	Number
DependOn	Element	1 or more

DependOn Element

Attribute	Description	Type	Number	Default
Name	Prerequisite episode name.	String	1 or more	Null

Request Element

This element represents a HTTP request contained in this episode. Variable value bindings for edge variables and node schema variables are required to be specified here.

Attribute	Description	Type	Number	Default
Edge	Edge ID for this HTTP request to follow.	String	1	Null

Contents	Type	Number
VarBind	Element	Unlimited

VarBind Element

Attribute	Description	Type	Number	Default
Name	Name of the variable that will be bound to some value.	String	1	Null
Type	Type of this binding. Valid values: *Edge*, *Node*	Enumeration	0 or 1	*Node*
Value	A value that will be bound to this variable. The value attribute can be a constant value or a variable reference.	String/Reference	1	Null

Loop Element

Attribute	Description	Type	Number	Default
Max	Maximum continuous repetition of this loop during execution. No specification means loop forever until no more data can be extracted. Unit: iteration	Number	0 or 1	100

Contents	Type	Number
Request	Element	0 or more
Loop	Element	0 or more

A.3 Complete WNDL Document Type Definition

```
<!ELEMENT WNDL (Map, Program)>
<!ATTLIST WNDL Name    CDATA #REQUIRED
               Version CDATA #REQUIRED>

<!ELEMENT Map (Entrance, Node+)>

<!ELEMENT Entrance (Edge+)>

<!ELEMENT Node (Schema?, ExtractRule*, Edge*)>
<!ATTLIST Node Name CDATA #REQUIRED>

<!ELEMENT Schema (Attr+)>

<!ELEMENT Attr EMPTY>
<!ATTLIST Attr Name CDATA #REQUIRED
               TagFilter (KeepAll|KeepLink|NoTag) "KeepLink">

<!ELEMENT ExtractRule (#PCDATA)*>
<!ATTLIST ExtractRule File CDATA #IMPLIED>

<!ELEMENT Edge    (EdgeInput?, Query?, Timeout?)>
<!ATTLIST Edge    ID     CDATA               #REQUIRED
                  Type   (Static|Dynamic)    "Static"
                  Base   CDATA               #IMPLIED
                  URL    CDATA               #REQUIRED
                  Method (Get|Post)          "Get"
                  Dest   CDATA               #REQUIRED>

<!ELEMENT EdgeInput (EdgeParam+)>

<!ELEMENT EdgeParam EMPTY>
<!ATTLIST EdgeParam Name CDATA #REQUIRED>

<!ELEMENT Query (QueryParam+)>
```

```
<!ELEMENT QueryParam EMPTY>
<!ATTLIST QueryParam FormInput   CDATA #REQUIRED
                     Value       CDATA #REQUIRED>

<!ELEMENT Timeout EMPTY>
<!ATTLIST Timeout Wait   CDATA #IMPLIED
                  Retry  CDATA #IMPLIED>

<!ELEMENT Program (Session+)>

<!ELEMENT Session (VarDeclr, Access)>
<!ATTLIST Session Name CDATA #REQUIRED>

<!ELEMENT VarDeclr (SessionVar+)>

<!ELEMENT SessionVar EMPTY>
<!ATTLIST SessionVar  Name   CDATA                  #REQUIRED
                      Type   (In|Out|InOut|Tmp)     "Out"
                      Label  CDATA                  #IMPLIED
                      Value  CDATA                  #IMPLIED>

<!ELEMENT Access (Episode+)>

<!ELEMENT Episode (Dependency?, Request*, Loop*)>
<!ATTLIST Episode Name    CDATA   #REQUIRED
                  User    CDATA   #IMPLIED
                  Passwd CDATA   #IMPLIED>

<!ELEMENT Dependency (DependOn+)>

<!ELEMENT DependOn EMPTY>
<!ATTLIST DependOn Name CDATA #REQUIRED>

<!ELEMENT Request (VarBind*)>
<!ATTLIST Request  Edge CDATA #REQUIRED>
```

```
<!ELEMENT VarBind EMPTY>
<!ATTLIST VarBind Name    CDATA          #REQUIRED
                  Type    (Edge|Node)    "Node"
                  Value   CDATA          #REQUIRED>

<!ELEMENT  Loop   (Loop*, Request*)>
<!ATTLIST  Loop   Max     CDATA          #IMPLIED>
```

Appendix B. WNDL Sample Codes

B.1 WNDL Code for *amazon.com* Books Keyword Query

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE WNDL SYSTEM "WNDL.dtd">

<WNDL Name="Amazon Book Search" Version="0.4">
     <Map>
          <Entrance>
               <Edge   ID="1"
                       Type="Static"
                       Dest="A"
                       URL="http://www.amazon.com">
               </Edge>
          </Entrance>
          <Node Name="A">
               <Schema>
                    <Attr Name="form" TagFilter="KeepAll"/>
               </Schema>
               <ExtractRule File="amazon_home_rule.txt"/>
               <Edge ID="2"
                     URL="$query"
                     Method="Post"
                     Type="Dynamic"
                     Dest="B">
                    <EdgeInput>
                         <EdgeParam Name="query"/>
                         <EdgeParam Name="keyword"/>
                    </EdgeInput>
                    <Query>
                         <QueryParam FormInput="index"           Value="books"/>
                         <QueryParam FormInput="field-keywords"  Value="$keyword"/>
                         <QueryParam FormInput="Go"              Value="Go"/>
                    </Query>
               </Edge>
          </Node>
          <Node Name="B">
               <Schema>
                    <Attr Name="title"/>
                    <Attr Name="author"/>
                    <Attr Name="year"/>
                    <Attr Name="price"/>
                    <Attr Name="url_next"/>
               </Schema>
               <ExtractRule File="amazon_rule.txt"/>
               <Edge ID="3"
                     URL="$next"
                     Dest="B">
                    <EdgeInput>
                         <EdgeParam Name="next"/>
                    </EdgeInput>
               </Edge>
          </Node>
     </Map>
```

```
    <Program>
        <Session Name="Test">
            <VarDeclr>
                <SessionVar Name="X"     Type="Tmp"/>
                <SessionVar Name="Y"     Type="In"/>
                <SessionVar Name="Z1"    Type="Out"     Label="Title"/>
                <SessionVar Name="Z2"    Type="Out"     Label="Author"/>
                <SessionVar Name="Z3"    Type="Out"     Label="Date"/>
                <SessionVar Name="Z4"    Type="Out"     Label="Price"/>
                <SessionVar Name="Z5"    Type="Tmp"/>
            </VarDeclr>
            <Access>
                <Episode Name="dummy">
                    <Request Edge="1">
                        <VarBind Name="form"     Type="Node" Value="$X"/>
                    </Request>
                    <Request Edge="2">
                        <VarBind Name="query"    Type="Edge" Value="$X"/>
                        <VarBind Name="keyword"  Type="Edge" Value="$Y"/>
                        <VarBind Name="title"    Type="Node" Value="$Z1"/>
                        <VarBind Name="author"   Type="Node" Value="$Z2"/>
                        <VarBind Name="year"     Type="Node" Value="$Z3"/>
                        <VarBind Name="price"    Type="Node" Value="$Z4"/>
                        <VarBind Name="url_next" Type="Node" Value="$Z5"/>
                    </Request>
                    <Loop Max="100">
                        <Request Edge="3">
                            <VarBind Name="next" Type="Edge" Value="Z5"/>
                        </Request>
                    </Loop>
                </Episode>
            </Access>
        </Session>
    </Program>
</WNDL>
```

B.2 WNDL Code for *TaipeiCity*

```
<?xml version="1.0" encoding="Big5" standalone="no"?>
<!DOCTYPE WNDL SYSTEM "WNDL.dtd">

<WNDL Name="TaipeiCity" Version="0.4">
     <Map>
          <Entrance>
               <Edge  ID="1"
                      Type="Static"
                      Dest="A"
                      Method="Get"
                      URL="http://163.29.128.6/OKwork/Data_Com/Company2.asp?haker=0&amp;employ_
                      type=1"/>
          </Entrance>
          <Node Name="A">
               <Schema>
                    <Attr Name="job_type_url"/>
                    <Attr Name="job_type"/>
                    <Attr Name="number"/>
                    <Attr Name="explanation"/>
               </Schema>
               <ExtractRule File="taipeicity-table_rule.txt"/>
               <Edge ID="2" URL="$url" Dest="B">
                    <EdgeInput>
                         <EdgeParam Name="url"/>
                    </EdgeInput>
               </Edge>
          </Node>
          <Node Name="B">
               <Schema>
                    <Attr Name="detail_url" TagFilter="KeepAll"/>
                    <Attr Name="company"/>
                    <Attr Name="title"/>
                    <Attr Name="contact"/>
                    <Attr Name="phone"/>
                    <Attr Name="post_date"/>
               </Schema>
               <ExtractRule File="taipeicity_rule.txt"/>
          </Node>
     </Map>
     <Program>
          <Session Name="Test">
               <VarDeclr>
                    <SessionVar Name="X1" Type="Tmp"/>
                    <SessionVar Name="X2" Type="Tmp"/>
                    <SessionVar Name="X3" Type="Tmp"/>
                    <SessionVar Name="X4" Type="Tmp"/>
                    <SessionVar Name="Y1" Type="Out" Label="DETAIL_URL"/>
                    <SessionVar Name="Y2" Type="Out" Label="COMPANY"/>
                    <SessionVar Name="Y3" Type="Out" Label="TITLE"/>
                    <SessionVar Name="Y4" Type="Out" Label="CONTACT_PERSON"/>
                    <SessionVar Name="Y5" Type="Out" Label="CONTACT_PHONE"/>
                    <SessionVar Name="Y6" Type="Out" Label="POST_DATE"/>
               </VarDeclr>
```

```
               <Access>
                    <Episode Name="dummy">
                         <Request Edge="1">
                              <VarBind Name="job_type_url"  Type="Node" Value="$X1"/>
                              <VarBind Name="job_type"      Type="Node" Value="$X2"/>
                              <VarBind Name="number"        Type="Node" Value="$X3"/>
                              <VarBind Name="explanation"   Type="Node" Value="$X4"/>
                         </Request>
                         <Request Edge="2">
                              <VarBind Name="url"           Type="Edge" Value="$X1"/>
                              <VarBind Name="detail_url"    Type="Node" Value="$Y1"/>
                              <VarBind Name="company"       Type="Node" Value="$Y2"/>
                              <VarBind Name="title"         Type="Node" Value="$Y3"/>
                              <VarBind Name="contact"       Type="Node" Value="$Y4"/>
                              <VarBind Name="phone"         Type="Node" Value="$Y5"/>
                              <VarBind Name="post_date"     Type="Node" Value="$Y6"/>
                         </Request>
                    </Episode>
               </Access>
          </Session>
     </Program>
</WNDL>
```

B.3 WNDL Code for *TTimes*

```
<?xml version="1.0" encoding="Big5" standalone="no"?>
<!DOCTYPE WNDL SYSTEM "WNDL.dtd">

<WNDL Name="TTimes" Version="0.3">
     <Map>
          <Entrance>
               <Edge   ID="1"
                       Type="Static"
                       Dest="A"
                       Method="Get"
                       URL="http://www.ttimes.com.tw/index_sub1.html">
               </Edge>
          </Entrance>
          <Node Name="A">
               <Schema>
                    <Attr Name="headline"/>
                    <Attr Name="url"/>
               </Schema>
               <ExtractRule>
                    <![CDATA[
                         R(GB,RB):Big5(_)Punc(_)Html(_);
                         L(GB,RB):;
                         R(RB,headline):;
                         L(RB,headline):Ptag(<b>);
                         R(headline,@headline):Html(_);
                         L(headline,@headline):;
                         R(@headline,url):Ptag(<a>);
                         L(@headline,url):;
                         R(url,RE):Ptag(<img>);
                         L(url,RE):;
                         R(RE,GE):Ttag(</a>)Ttag(</font>)Ptag(<!----end sub sub article highlight
                         bar 名人 PUB----->)Ptag(<!Sub_Category_Headline_11_Zone>);
                         L(RE,GE):;
                    ]]>
                </ExtractRule>
                    <Edge ID="2" URL="$url" Dest="B" Method="Get">
                         <EdgeInput>
                              <EdgeParam Name="url"/>
                         </EdgeInput>
                    </Edge>
          </Node>
          <Node Name="B">
               <Schema>
                    <Attr Name="path"/>
                    <Attr Name="title"/>
                    <Attr Name="subtitle"/>
                    <Attr Name="reporter"/>
                    <Attr Name="photo"/>
                    <Attr Name="source"/>
                    <Attr Name="dateTime"/>
                    <Attr Name="paragraph"/>
               </Schema>
               <ExtractRule File="ttimes_rule.txt"/>
          </Node>
     </Map>
     <Program>
          <Session Name="Test">
               <VarDeclr>
                    <SessionVar Name="X"     Type="Tmp"/>
                    <SessionVar Name="Y"     Type="Tmp"/>
                    <SessionVar Name="Z1"    Type="Out" Label="Path"/>
                    <SessionVar Name="Z2"    Type="Out" Label="Title"/>
                    <SessionVar Name="Z3"    Type="Out" Label="Subtitle"/>
                    <SessionVar Name="Z4"    Type="Out" Label="Reporter"/>
                    <SessionVar Name="Z45"   Type="Out" Label="PhotoSource"/>
                    <SessionVar Name="Z5"    Type="Out" Label="Source"/>
                    <SessionVar Name="Z6"    Type="Out" Label="Time"/>
                    <SessionVar Name="Z7"    Type="Out" Label="Article"/>
               </VarDeclr>
```

```
                <Access>
                     <Episode Name="dummy">
                          <Request Edge="1">
                               <VarBind Name="url"           Type="Node" Value="$X"/>
                               <VarBind Name="headline"      Type="Node" Value="$Y"/>
                          </Request>
                          <Request Edge="2">
                               <VarBind Name="url"           Type="Edge" Value="$X"/>
                               <VarBind Name="path"          Type="Node" Value="$Z1"/>
                               <VarBind Name="title"         Type="Node" Value="$Z2"/>
                               <VarBind Name="subtitle"      Type="Node" Value="$Z3"/>
                               <VarBind Name="reporter"      Type="Node" Value="$Z4"/>
                               <VarBind Name="photo"         Type="Node" Value="$Z45"/>
                               <VarBind Name="source"        Type="Node" Value="$Z5"/>
                               <VarBind Name="dateTime"      Type="Node" Value="$Z6"/>
                               <VarBind Name="paragraph"     Type="Node" Value="$Z7"/>
                          </Request>
                     </Episode>
                </Access>
           </Session>
      </Program>
</WNDL>
```

B.4 WNDL Code for *NYC*

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE WNDL SYSTEM "WNDL.dtd">

<WNDL Name="NYC" Version="0.4">
	<Map>
		<Entrance>
			<Edge	ID="1"
				Type="Static"
				Dest="1"
				Method="Get"
				URL="http://site1.nyc.gov.tw/db/index.asp?sss=1">
			</Edge>
		</Entrance>
		<Node Name="1">
			<Schema>
				<Attr Name="url"/>
				<Attr Name="title"/>
			</Schema>
			<ExtractRule File="node1_rule.txt"/>
			<Edge ID="2" URL="$url" Dest="2">
				<EdgeInput>
					<EdgeParam Name="url"/>
				</EdgeInput>
			</Edge>
		</Node>
		<Node Name="2">
			<Schema>
				<Attr Name="url"/>
				<Attr Name="title"/>
			</Schema>
			<ExtractRule File="node2_rule.txt"/>
			<Edge ID="3" URL="$url" Dest="3">
				<EdgeInput>
					<EdgeParam Name="url"/>
				</EdgeInput>
			</Edge>
		</Node>
		<Node Name="3">
			<Schema>
				<Attr Name="detail_url"/>
				<Attr Name="company"/>
				<Attr Name="location"/>
				<Attr Name="contact_person"/>
				<Attr Name="contact_phone"/>
				<Attr Name="contact_method"/>
				<Attr Name="requirement"/>
				<Attr Name="title"/>
				<Attr Name="salary"/>
				<Attr Name="post_date"/>
				<Attr Name="expire_date"/>
				<Attr Name="next"/>
			</Schema>
			<ExtractRule File="rule1.txt"/>
			<Edge ID="4" URL="$next" Dest="3">
				<EdgeInput>
					<EdgeParam Name="next"/>
				</EdgeInput>
			</Edge>
		</Node>
	</Map>
```

```
     <Program>
          <Session Name="Test">
               <VarDeclr>
                    <SessionVar Name="X"     Type="Tmp"/>
                    <SessionVar Name="Y"     Type="Tmp"/>
                    <SessionVar Name="M"     Type="Tmp"/>
                    <SessionVar Name="N"     Type="Tmp"/>
                    <SessionVar Name="Z1"    Type="Out" Label="DETAIL_URL"/>
                    <SessionVar Name="Z2"    Type="Out" Label="COMPANY"/>
                    <SessionVar Name="Z3"    Type="Out" Label="LOCATION"/>
                    <SessionVar Name="Z4"    Type="Tmp" Label="CONTACT_PERSON"/>
                    <SessionVar Name="Z5"    Type="Tmp" Label="CONTACT_PHONE"/>
                    <SessionVar Name="Z6"    Type="Tmp" Label="CONTACT_METHOD"/>
                    <SessionVar Name="Z7"    Type="Tmp" Label="REQUIREMENT"/>
                    <SessionVar Name="Z8"    Type="Out" Label="TITLE"/>
                    <SessionVar Name="Z9"    Type="Out" Label="SALARY"/>
                    <SessionVar Name="Z10"   Type="Out" Label="POST_DATE"/>
                    <SessionVar Name="Z11"   Type="Tmp" Label="EXPIRE_DATE"/>
                    <SessionVar Name="Z12"   Type="Tmp" Label="next_page"/>
               </VarDeclr>
               <Access>
                    <Episode Name="dummy">
                         <Request Edge="1">
                              <VarBind Name="url"                Type="Node" Value="$M"/>
                              <VarBind Name="title"              Type="Node" Value="$N"/>
                         </Request>
                         <Request Edge="2">
                              <VarBind Name="url"                Type="Edge" Value="$M"/>
                              <VarBind Name="url"                Type="Node" Value="$X"/>
                              <VarBind Name="title"              Type="Node" Value="$Y"/>
                         </Request>
                         <Request Edge="3">
                              <VarBind Name="url"                Type="Edge" Value="$X"/>
                              <VarBind Name="detail_url"         Type="Node" Value="$Z1"/>
                              <VarBind Name="company"            Type="Node" Value="$Z2"/>
                              <VarBind Name="location"           Type="Node" Value="$Z3"/>
                              <VarBind Name="contact_person"     Type="Node" Value="$Z4"/>
                              <VarBind Name="contact_phone"      Type="Node" Value="$Z5"/>
                              <VarBind Name="contact_method"     Type="Node" Value="$Z6"/>
                              <VarBind Name="requirement"        Type="Node" Value="$Z7"/>
                              <VarBind Name="title"              Type="Node" Value="$Z8"/>
                              <VarBind Name="salary"             Type="Node" Value="$Z9"/>
                              <VarBind Name="post_date"          Type="Node" Value="$Z10"/>
                              <VarBind Name="expire_date"        Type="Node" Value="$Z11"/>
                              <VarBind Name="next"               Type="Node" Value="$Z12"/>
                         </Request>
                         <Loop Max="5">
                              <Request Edge="4">
                                   <VarBind Name="next"          Type="Edge" Value="$Z12"/>
                              </Request>
                         </Loop>
                    </Episode>
               </Access>
          </Session>
     </Program>
</WNDL>
```

B.5 WNDL Code for *CTCareer*

```
<?xml version="1.0" encoding="Big5" standalone="no"?>
<!DOCTYPE WNDL SYSTEM "WNDL.dtd">

<WNDL Name="CTCareer" Version="0.4">
     <Map>
          <Entrance>
               <Edge ID="0"
                     Type="Static"
                     Dest="A"
                     URL="http://www.ctcareer.com.tw/findjob.asp?Action=1"/>
          </Entrance>
          <Node Name="A">
               <Edge ID="1"
                     Type="Dynamic"
                     Dest="B"
                     Method="Post"
                     URL="http://www.ctcareer.com.tw/findjob.exe">
                    <EdgeInput>
                         <EdgeParam Name="sJobMainCatID"/>
                    </EdgeInput>
                    <Query>
                         <QueryParam FormInput="sSex"            Value="1"/>
                         <QueryParam FormInput="sJobType"        Value="3"/>
                         <QueryParam FormInput="JobDate"         Value="all"/>
                         <QueryParam FormInput="iJobDiploma"     Value="0"/>
                         <QueryParam FormInput="sCityID"         Value="00"/>
                         <QueryParam FormInput="sCompCatID"      Value="00"/>
                         <QueryParam FormInput="sCompSubCatID"   Value="000"/>
                         <QueryParam FormInput="sJobMainCatID"   Value="$sJobMainCatID"/>
                         <QueryParam FormInput="sJobSubCatID"    Value="00"/>
                         <QueryParam FormInput="sCompanyType"    Value="0"/>
                         <QueryParam FormInput="sInSciencePark"  Value="2"/>
                         <QueryParam FormInput="sIsForeign"      Value="2"/>
                         <QueryParam FormInput="noperpage"       Value="50"/>
                         <QueryParam FormInput="Submit"          Value="開始查詢"/>
                         <QueryParam FormInput="index_root"      Value="d:\ctcareer2\index"/>
                    </Query>
               </Edge>
          </Node>
          <Node Name="B">
               <Schema>
                    <Attr Name="detail_url"/>
                    <Attr Name="company"/>
                    <Attr Name="title"/>
                    <Attr Name="degree"/>
                    <Attr Name="experience"/>
                    <Attr Name="age"/>
                    <Attr Name="salary"/>
                    <Attr Name="location"/>
                    <Attr Name="post_date"/>
                    <Attr Name="next" TagFilter="KeepAll"/>
               </Schema>
               <ExtractRule File="rule.txt"/>
               <Edge ID="2"
                     Type="Dynamic"
                     Dest="B"
                     Method="Post"
                     URL="$next">
                    <EdgeInput>
                         <EdgeParam Name="next"/>
                    </EdgeInput>
                    <Query>
                         <QueryParam FormInput="PageDown" Value=" 下一頁 "/>
                    </Query>
               </Edge>
          </Node>
     </Map>
```

```
    <Program>
        <Session Name="Test">
            <VarDeclr>
                <SessionVar Name="J"     Type="In"/>
                <SessionVar Name="X1"    Type="Out"  Label="DETAIL_URL"/>
                <SessionVar Name="X2"    Type="Out"  Label="COMPANY"/>
                <SessionVar Name="X3"    Type="Out"  Label="TITLE"/>
                <SessionVar Name="X4"    Type="Tmp"  Label="DEGREE"/>
                <SessionVar Name="X5"    Type="Tmp"  Label="EXPERIENCE"/>
                <SessionVar Name="X6"    Type="Tmp"  Label="AGE"/>
                <SessionVar Name="X7"    Type="Out"  Label="SALARY"/>
                <SessionVar Name="X8"    Type="Out"  Label="LOCATION"/>
                <SessionVar Name="X9"    Type="Out"  Label="POST_DATE"/>
                <SessionVar Name="X10"   Type="Tmp"/>
            </VarDeclr>
            <Access>
                <Episode Name="dummy">
                    <Request Edge="0"/>
                    <Request Edge="1">
                        <VarBind Name="sJobMainCatID" Type="Edge" Value="$J"/>
                        <VarBind Name="detail_url"    Type="Node" Value="$X1"/>
                        <VarBind Name="company"       Type="Node" Value="$X2"/>
                        <VarBind Name="title"         Type="Node" Value="$X3"/>
                        <VarBind Name="degree"        Type="Node" Value="$X4"/>
                        <VarBind Name="experience"    Type="Node" Value="$X5"/>
                        <VarBind Name="age"           Type="Node" Value="$X6"/>
                        <VarBind Name="salary"        Type="Node" Value="$X7"/>
                        <VarBind Name="location"      Type="Node" Value="$X8"/>
                        <VarBind Name="post_date"     Type="Node" Value="$X9"/>
                        <VarBind Name="next"          Type="Node" Value="$X10"/>
                    </Request>
                    <Loop Max="5">
                        <Request Edge="2">
                            <VarBind Name="next"      Type="Edge" Value="$X10"/>
                        </Request>
                    </Loop>
                </Episode>
            </Access>
        </Session>
    </Program>
</WNDL>
```

B.6 WNDL Code for *JobsDB*

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE WNDL SYSTEM "WNDL.dtd">

<WNDL Name="jobsdb" Version="0.4">
     <Map>
          <Entrance>
               <Edge    ID="1"
                        Type="Static"
                        Dest="A"
                        Method="Get”
                        URL="http://www.jobsdb.com.tw/TW/B5/default.asp?pagename=adslist&amp;Sor
                        tOpt=PostDate&amp;A=&amp;S=J&amp;ListURL=%2FTW%2FB5%2Fdefault.asp&amp;Bu
                        tton=Ads&amp;SearchText=&amp;SearchRef=&amp;JobArea=000&amp;area=&amp;di
                        strict=&amp;Find=%B7j%B4M">
               </Edge>
          </Entrance>
          <Node Name="A">
               <Schema>
                    <Attr Name="DETAIL_URL"/>
                    <Attr Name="TITLE"/>
                    <Attr Name="COMPANY"/>
                    <Attr Name="LOCATION"/>
                    <Attr Name="POST_DATE"/>
                    <Attr Name="NUMBER"/>
                    <Attr Name="form" TagFilter="KeepAll"/>
               </Schema>
               <ExtractRule File="rule_all.txt"/>
               <ExtractRule File="rule_form2.txt"/>
               <Edge ID="2"
                     URL="$nextLink"
                     Method="Get"
                     Type="Dynamic"
                     Dest="A">
                    <EdgeInput>
                         <EdgeParam Name="nextLink"/>
                    </EdgeInput>
                    <Query>
                         <QueryParam FormInput="Next" Value="下一頁"/>
                    </Query>
               </Edge>
          </Node>
     </Map>
     <Program>
          <Session Name="Test">
               <VarDeclr>
                    <SessionVar Name="X"     Type="Tmp"/>
                    <SessionVar Name="Z1"    Type="Tmp"/>
                    <SessionVar Name="Z2"    Type="Out" Label="TITLE"/>
                    <SessionVar Name="Z3"    Type="Out" Label="COMPANY"/>
                    <SessionVar Name="Z4"    Type="Out" Label="LOCATION"/>
                    <SessionVar Name="Z5"    Type="Out" Label="POST_DATE"/>
                    <SessionVar Name="Z6"    Type="Out" Label="DETAIL_URL"/>
                    <SessionVar Name="Z7"    Type="Tmp"/>
               </VarDeclr>
```

```
                <Access>
                     <Episode Name="dummy">
                          <Request Edge="1">
                               <VarBind Name="DETAIL_URL"    Type="Node" Value="$Z1"/>
                               <VarBind Name="TITLE"         Type="Node" Value="$Z2"/>
                               <VarBind Name="COMPANY"       Type="Node" Value="$Z3"/>
                               <VarBind Name="LOCATION"      Type="Node" Value="$Z4"/>
                               <VarBind Name="POST_DATE"     Type="Node" Value="$Z5"/>
                               <VarBind Name="NUMBER"        Type="Node" Value="$Z6"/>
                               <VarBind Name="form"          Type="Node" Value="$Z7"/>
                          </Request>
                          <Loop Max="5">
                               <Request Edge="2">
                                    <VarBind Name="nextLink" Type="Edge" Value="$Z7"/>
                               </Request>
                          </Loop>
                     </Episode>
                </Access>
           </Session>
      </Program>
</WNDL>
```

B.7 WNDL Code for *104Bank*

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE WNDL SYSTEM "WNDL.dtd">

<WNDL Name="104-test" Version="0.4">
     <Map>
          <Entrance>
               <Edge   ID="1"
                       Type="Dynamic"
                       Dest="A"
                       Method="Post"
                       URL="http://www.104.com.tw/cfdocs/job1qry1.dbm">
                    <Query>
                         <QueryParam FormInput="submit" Value="依職務類別及地點查詢"/>
                    </Query>
                    <Timeout Wait="60" Retry="2"/>
               </Edge>
          </Entrance>
          <Node Name="A">
               <Schema>
                    <Attr Name="button" TagFilter="KeepAll"/>
               </Schema>
               <ExtractRule File="rule_104job1qry.txt"/>
               <Edge   ID="2"
                       Type="Dynamic"
                       Dest="B"
                       Method="Get"
                       URL="$form">
                    <EdgeInput>
                         <EdgeParam Name="form"/>
                    </EdgeInput>
                    <Timeout Wait="60" Retry="2"/>
               </Edge>
          </Node>
          <Node Name="B">
               <Schema>
                    <Attr Name="button" TagFilter="KeepAll"/>
               </Schema>
                    <ExtractRule File="rule_104taipeicity.txt"/>
                    <ExtractRule File="rule_104taipeicounty.txt"/>
                    <ExtractRule File="rule_104others.txt"/>
               <Edge   ID="3"
                       Type="Dynamic"
                       Dest="C"
                       Method="Get"
                       URL="$form">
                    <EdgeInput>
                         <EdgeParam Name="form"/>
                    </EdgeInput>
                    <Timeout Wait="60" Retry="2"/>
               </Edge>
          </Node>
          <Node Name="C">
               <Schema>
                    <Attr Name="new"/>
                    <Attr Name="company"/>
                    <Attr Name="detail_url"/>
                    <Attr Name="title"/>
                    <Attr Name="job_type"/>
                    <Attr Name="gender"/>
                    <Attr Name="degree"/>
                    <Attr Name="experience"/>
                    <Attr Name="age"/>
                    <Attr Name="salary"/>
               </Schema>
               <ExtractRule File="104_ruleH.txt"/>
          </Node>
     </Map>
```

```
    <Program>
        <Session Name="Test">
            <VarDeclr>
                <SessionVar Name="X1"    Type="Tmp"     Label="COMPANY_TYPE"/>
                <SessionVar Name="X2"    Type="Out"     Label="LOCATION"/>
                <SessionVar Name="X3"    Type="Tmp"     Label="NEW"/>
                <SessionVar Name="X4"    Type="Out"     Label="COMPANY"/>
                <SessionVar Name="X5"    Type="Out"     Label="DETAIL_URL"/>
                <SessionVar Name="X6"    Type="Out"     Label="TITLE"/>
                <SessionVar Name="X7"    Type="Tmp"     Label="JOB_TYPE"/>
                <SessionVar Name="X8"    Type="Tmp"     Label="GENDER"/>
                <SessionVar Name="X9"    Type="Tmp"     Label="DEGREE"/>
                <SessionVar Name="X10"   Type="Tmp"     Label="EXPERIENCE"/>
                <SessionVar Name="X11"   Type="Tmp"     Label="AGE"/>
                <SessionVar Name="X12"   Type="Out"     Label="SALARY"/>
            </VarDeclr>
            <Access>
                <Episode Name="dummy">
                    <Request Edge="1">
                        <VarBind Name="button"   Type="Node" Value="$X1"/>
                    </Request>
                    <Request Edge="2">
                        <VarBind Name="form"     Type="Edge" Value="$X1"/>
                        <VarBind Name="button"   Type="Node" Value="$X2"/>
                    </Request>
                    <Request Edge="3">
                        <VarBind Name="form"          Type="Edge" Value="$X2"/>
                        <VarBind Name="new"           Type="Node" Value="$X3"/>
                        <VarBind Name="company"       Type="Node" Value="$X4"/>
                        <VarBind Name="detail_url"    Type="Node" Value="$X5"/>
                        <VarBind Name="title"         Type="Node" Value="$X6"/>
                        <VarBind Name="job_type"      Type="Node" Value="$X7"/>
                        <VarBind Name="gender"        Type="Node" Value="$X8"/>
                        <VarBind Name="degree"        Type="Node" Value="$X9"/>
                        <VarBind Name="experience"    Type="Node" Value="$X10"/>
                        <VarBind Name="age"           Type="Node" Value="$X11"/>
                        <VarBind Name="salary"        Type="Node" Value="$X12"/>
                    </Request>
                </Episode>
            </Access>
        </Session>
    </Program>
</WNDL>
```