University of Alberta

Reengineering Web Applications to Web-service Providers

by

Yingtao Jiang 

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science.**

Department of Computing Science

Edmonton, Alberta

Spring 2005

Abstract

Web services are the latest technology to integrate applications through Internet. Many B2C services backed by web applications could be reused in an application integration scenario. Correctly and effectively migrating those web applications that sit behind a presentation layer poses a challenging problem for researchers of software engineering.

ServiceBuilder is a tool that can automatically generate a web-service compliant wrapper around web application without knowledge about its code base. To achieve this ServiceBuilder collects a set of HTML documents by feeding the web application with input data. Then it applies pattern-mining techniques on the collected response documents and with little user involvement, generates data extraction rules for data around predefined labels of interests. Finally, ServiceBuilder generates a wrapper that at run time forwards the service requester's input data to the web application and extracts the output data from the responding document and returns it to the service requester.

Acknowledgements

I would like to extend my heartfelt gratitude to my supervisor, Dr. Eleni Stroulia, for her guidance, insightful feedback and support throughout this work. Without her vision, guidance and encouragement, this work would not have been possible.

I would also like to extend my thanks to my colleagues and friends for their help and encouragements during my graduate studies. My special thanks goes to Edward Zadrozny and Sze-Lai Mok for their contributions towards the implementation of the prototype system. Special thanks also goes to Dr. Mohammad El-Ramly for providing me with the original implementation of the IPM algorithm.

Finally, I would like to thank my family and my wife for all their love and support throughout the years.

Table of Contents

List of Tables

List of Figures

Chapter 1 Introduction and Motivation

The World Wide Web is changing rapidly from an application-to-human interaction medium to an application-to-application interaction and integration medium. Web services are the latest standard that dictates the application interaction and integration through the Web. In this chapter, we first give a short introduction to web services. Then, we establish the motivation of this research and briefly describe our proposed approach to the research problem. Finally, we summarize the main contributions of this research and outline the thesis organization.

1.1 Introduction

Since its invention, the World Wide Web has been expanding very fast. The major driving force behind this growth is the businesses' use of WWW as a service delivery channel. As the business demands change from supporting B2C interactions to also supporting B2B collaborations as well, the technologies behind the delivery of services change accordingly. In this section, we review the traditional web application technologies that only support human-machine interaction and the newest web services technology that support machine-machine interaction.

1.1.1 Traditional web applications

In its early days, the World Wide Web consisted of static HTML (Hyper Text Mark-up Language) documents weaved together by hyperlinks and was mainly used in academia as a media of information sharing. As the Web evolved, it quickly attracted the attention of businesses and was used as an information-publishing and service-delivery channel. The software systems responsible for delivering services through the web are called web applications. By definition, a web application is "a software program that uses HTTP for its core communication protocol and delivers Web-based information to the user in the HTML language."[MWSD] A typical web application is usually a three-tiered system as shown in Figure 1.

We can see that the whole system is divided into three separate tiers: presentation, business logic, and data storage and access. This architecture makes it possible to develop and maintain each tier separately. In a working web application, the presentation tier tends to be changed more frequently than the other two tiers.

As the web application uses a browser as its client side, it is obvious that it intends to deliver service to an end-user, i.e. a person. Therefore, web applications are usually used in a B2C business model. Nowadays, a large number of online businesses provide services to their customers in this fashion (i.e. a machine to human interaction style). According to [EMR02], a report published by the United States Census Bureau, in 2002 the total sales value of this kind of E-commerce in the United States alone is $85 billion. It is interesting to note that many of the services delivered in this way could be reused in a B2B (Business to Business) scenario. For example, the services provided by online hotel reservation, car rental, plan ticket booking etc. could be reused by an online travel planning service. In other words, a new travel planning service could be created by integrating those services currently provided to users online.

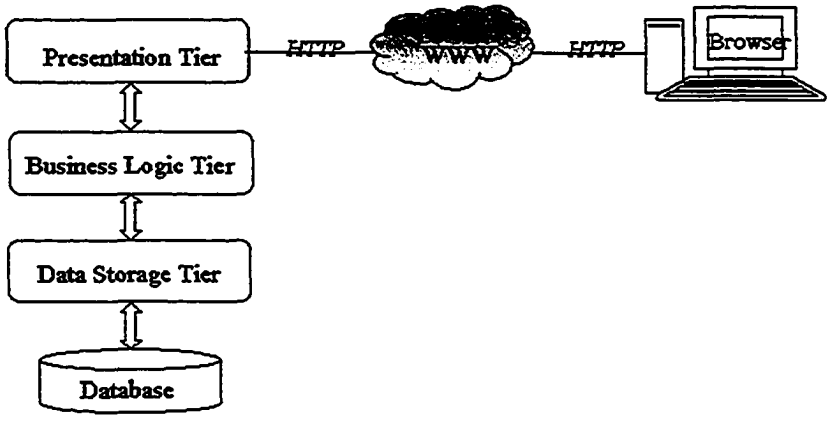

Figure 1: Three-tiered architecture of web application.

However, this is not an easy task. If we look back at the three-tier web application architecture, we can see that the actual business functionality we want to integrate lies in the business logic tier, which is hidden behind a presentation layer and not directly available through a programmatic API. This kind of web application architecture is not an

arbitrary choice. It conforms to the traditional web architecture, which essentially is defined in terms of a URL, as the unique location from where to access the application, HTTP, as the protocol to communicate with the application, and HTML, as the language of the interface of the Web. HTML is a human-oriented interface. But to integrate functionality through the Web, we need a machine-oriented interface, i.e. a programming interface.

The effort to integrate web applications started in the early days of web applications. Limited by the old web architecture, integrations were usually formed in an ad hoc manner since there were no standards to guide the way how to integrate those applications and how to define the programming interface through which an application should exposed to the web.

1.1.2 Web Services

The web-services stack of standards constitutes the latest technology in support of a standard way for applications to expose their functionalities on the web and interoperate with other applications. Web services are based on a new architecture paradigm called Service-Oriented Architecture (SOA). Figure 2 shows a basic service-oriented architecture. In this architecture, there are three roles: a service provider, a service requester and a service broker.

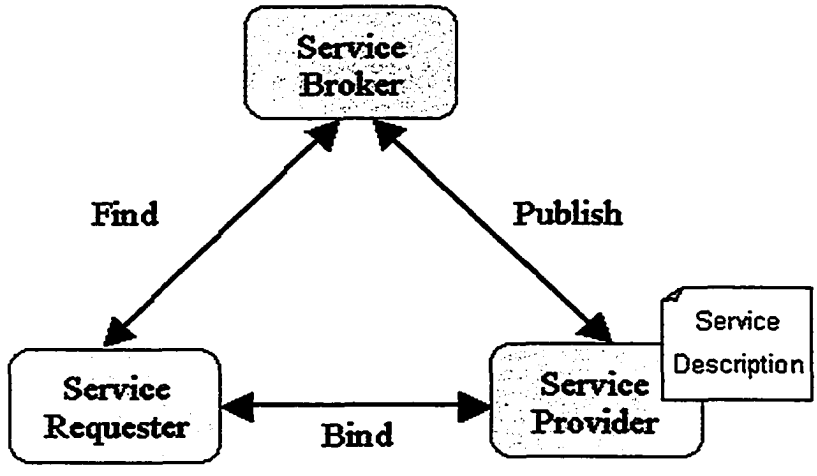

Figure 2: Basic Service-Oriented Architecture

The *service provider* is the host of a service and its implementation. A *service requester* is an entity that uses the service provided by the service provider. The service provider and the service requester communicate with each other through message exchange. The syntax of the message exchange is defined by the interface of the service, which is documented in a file called service description file. In order for the message exchange between service provider and service requesters to be successful, they must make an agreement in advance on both the syntax and the semantics of the message to be exchanged.

It is the third role in the architecture, i.e., the *service broker,* which is responsible for the establishment of the syntactic and semantic agreement between the service provider and service requester. In a typical scenario, the three roles interact as follows:

- The service provider *publishes* its services description and service semantics to the service broker.
- A service requester *finds* the service provider by submitting criteria of the expected service provider to the service broker.
- The service broker use the criteria provided by the service requester to locate a service that meets the criteria and returns the service description of the found service to the service requester.
- Based on the information provided in the service description, the service requester initiates a message "conversation" with the service provider. In other words, the service requester *binds* with the service provider.

The web-services stack of standards defines the basic elements of the implementation of the above-discussed service-oriented architecture. More specifically, web services includes the following basic standards:

- WSDL (Web Service Description Language). The metadata language developed by World Wide Web Consortium (W3C) as the service description language. A WSDL description file is the contract a service promised to the outside world. It specifies four major aspects of the service described:

- The data format a service consumes and produces in terms of a data type system.
- The messages format exchanged between a service and its client.
- The high level functionality a service provides in terms of operations.
- The actual access point on the web of this service.

- SOAP (Simple Object Access Protocol). SOAP is a lightweight XML-based protocol for messaging and Remote Procedure Calls (RPCs). It is the recommended messaging protocol for web services by W3C.
- UDDI (Universal Description Discovery & Integration). UDDI is an OASIS (Organization for the Advancement of Structured Information Standards) specification that defines a registry framework to implement the service broker services as described in SOA.

WSDL, SOAP and UDDI together constitute the basic set of web services standards that implement the basic service-oriented architecture. WSDL defines the way to describe the service, SOAP defines the communication protocol between service provider and service requester, and UDDI defines a standard registry service and the API to interact with it. So far those web service standards receive a universal support from vendors. There are other web services standardizations efforts out there to address issues such as service choreography (composition), security, quality of service (QoS) etc. But most of those efforts are not as mature and stable as the three standard we mentioned above.

1.2 The Research Problem: Motivation and Background

With the business demand for the Web to support new kinds of services, especially the support for machine-to-machine interaction, the architecture of the Web slowly evolves. Figure 3 shows the vision of the future web architecture by W3C. In this new architecture, the web services we introduced in the previous section are the standard way for an application to expose its functionality as a programmatic interface to the web. If an online application provides a web service interface, it is ready to be integrated by other services through the Web.

As the functionality of many legacy web applications already available on the web could be reused in a business-to-business integration scenario, reengineering existing web applications into web-services providers becomes a compelling research problem.

One possible way to reengineer a legacy web application into web services is by code-based migration, which usually involves writing a new component that reuses the old code and packages it in a novel way to expose the desired functionalities as web services. This approach is time consuming and error prone. In order to write the new code, the programmer needs to fully understand the existing code base, which is not an easy task especially when the legacy web application is large with sparse documentation.

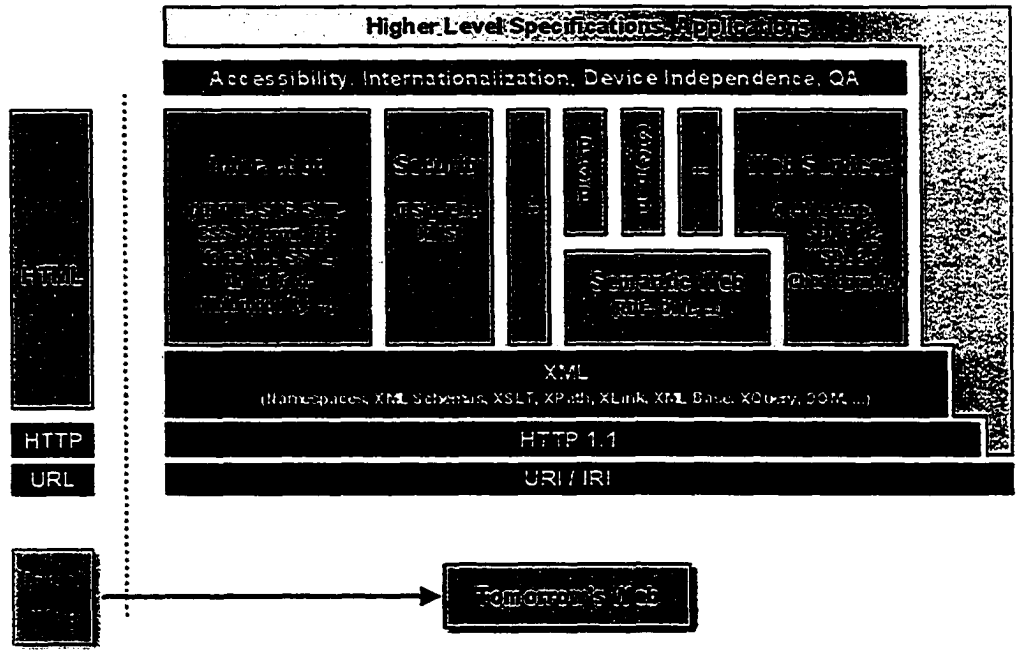

Figure 3: The architecture of future Web. (Reproduced from W3C website)

Another way is to use a middleware product such as [Artix] from IONA to migrate existing code base to web services. Those kinds of products provide tool support for migrating existing code bases to web services. Compared to the first approach, this approach is faster and less error prone. But it usually requires that the existing code base already conform to a certain kind of middleware standard or a specific object model.

Furthermore, it requires the purchase of new pieces of middleware that is usually too costly to afford for small companies, especially since the business model for using and providing web services is not completely clear yet.

Both of the above mentioned migration approaches assume the modification of the source code of the web application being migrated. However, that is not always a viable assumption: for example, the company that runs the web application does not own the source code or does not want to expose the source code to a third party for security reasons. Another example would be a newly started online travel assistant service that wants to migrate the publicly available online weather forecast services into web services so their can reuse it as part of the service their provide to their customer. In both cases, a challenging migration problem exists: how to migrate a web application into web service without access to its code base? This thesis provides a novel solution to address this problem.

1.3 The Proposed Solution

We propose a migration approach based on interaction reengineering [ERSS02], which does not require modifications to the source code. This approach is based on the observation of how a web application interacts with a user. The web application receives input as a user hits a submit button after filling a form in a web document. As a result, the web application executes its business logic and possibly contacts its underlying database and returns a dynamically generated web document containing the data the user expects. Note that, in this interaction, both the input and the output data are embedded in some HTML documents. If we write a piece of code to simulate the user input to the web application and to extract the user expected data out of the HTML document returned by the web application, then we could expose this piece of code as the programmatic interface to the original web application. Essentially this piece of new code is a wrapper around the user interface of the legacy web application. The major challenge of building such a wrapper is to determine where in the returned documents to extract the data a user expects. In other words, the wrapper needs to have the appropriate output-data extraction rule.

ServiceBuilder is a prototype system built to facilitate the automatic generation of such a wrapper. The generated wrapper conforms to the web service standards and is ready to be deployed as a web service. The basic idea behind ServiceBuilder's wrapper-generation process is that, even though each individual document returned by the web application might be different from one another in its contents, there are, however partial structural invariants among most returned documents that contain the actual returned data. Those invariant structural features – called *valid patterns* in ServiceBuilder - could be used as a stencil to extract data out in the future returned documents. There might be other invariant structures among returned documents that do not contain data of interest to the user. In order to distinguish those structures from patterns containing data, we introduce the concept of "landmarks". Landmarks are those words or symbols in the returned documents that are located in close proximity to data of interest. If a pattern contains landmarks, it is most likely also to contain data of interest. Intuitively, the data of interest is usually close to the meaningful domain-specific labels contained in the returned response document of the web application.

To learn the data-extraction rule, the ServiceBuilder first takes as input a set of "landmarks" and a set of example input data from the user. Then it encapsulates the example input data into properly formed of HTTP requests and sends those requests to the target web application. Once it has received all the returned documents from the target web application, ServiceBuilder uses sequential pattern mining techniques to mine through the collected documents and generates a set of valid patterns. The ServiceBuilder displays those patterns with whatever data they happen to contain in the HTML documents collected from the web-application responses highlighted to the user and lets the user decide which pattern actually contains data the user interested in. After the user selects the valid patterns, the ServiceBuilder presents a data *type editor* window, and supports the user in defining the desired output data format. The ServiceBuilder automatically establishes the mapping between the output data format and the rule used to extract data. Based on this mapping information, the ServiceBuilder automatically generates a Java implementation conforming to the web service standard that wraps around the target web site.

At run time, the wrapper takes input from its client application, appropriately formulates the client input and initiates a HTTP request to the target website. Upon receiving the web document from the target website, the wrapper uses the appropriate pattern to extract data from it and populates the extracted data into a Java Bean object, which is returned back as the result to the client.

1.4 The Contributions

The main contributions of this thesis are outlined below:

- It proposes a new methodology with a corresponding toolkit to migrate traditional web applications into web services providers.
- It introduces the use of sequential pattern mining techniques to solve the problem of extraction-rule learning in wrapper construction. This new approach has several advantages over traditional wrapper induction and tree structure based wrapper construction techniques.
    - First, it does not require the manual labeling of training examples.
    - Second, the produced wrapper is more robust to source changes than wrappers generated by other approaches.
    - Third, this approach can efficiently learn from a large amount of examples, thus providing more confidence for the resulting data-extraction rule.
- It provides a highly automatic, easy to use, wrapper construction toolkit. ServiceBuilder is highly automatic. With proper setup information, the user only needs to specify a set of landmarks and the tool automatically collects training examples from the web application and generates a small set of candidate extraction patterns. Those candidate patterns are visually presented to the user in the context of actual web documents collected. Once the user selects the pattern or patterns containing data of interest and defines the output data format with the help of a wizard, the tool automatically generates the final Java implementation. The whole wrapper construction process is just a matter of minutes.
- ServiceBuilder offers a set of heuristics that can efficiently and effectively eliminate most spurious patterns and greatly free the user from the burden of

wading through a large amount of candidate patterns to select the one that containing data of interest. Depending on the properties of the output web document of a web application, the user could select different heuristics to filter the candidate patterns.

- The generated wrapper is a Java implementation of web services and ready to be deployed on a variety of platforms. As a web service, the wrapper is accessible to a broad range of clients.
- This work provides a foundation for further service composition extensions. Now ServiceBuilder can only migrate one step, search engine like services into web services. With a state management component and a platform that support the future web service composition standards, it could be extended to migrate multi-step, complex web applications into web services.

1.5 Thesis Outline

This thesis is organized as follows. Chapter 2 provides an overview of some related research on wrapper construction in general. Some representative web wrapper construction tools are described and compared to ServiceBuilder. Chapter 3 describes ServiceBuilder in detail. A high-level architecture overview is given first. Then following the data flow, the implementation and the process of each individual component is discussed in detail. Experiments and evaluations are described in Chapter 4. Chapter 5 summarizes the thesis with major contributions of this research and points out some possible future research directions.

Chapter 2 Related Work

The web service implementation generated by the ServiceBuilder is essentially a wrapper around the web application. In this chapter, we will review some previous research works in the field of web wrapper construction and will compare them with ServiceBuilder.

A Web wrapper is a software component that can do the following:

- Retrieve web document;
- Identify data of interest from the web document;
- Extract the identified data and package it into a format desired by the user of the wrapper.

The major challenge in building a wrapper is to learn the extraction rule that can be used to correctly and efficiently extract the right data out of target web documents.

During the early days of the web, wrappers were usually manually constructed using different kinds of general programming languages such as Java, C++ or Perl. The major problem with these hand-coded wrapper-construction processes is that they are labor-intensive, error-prone and hard to maintain.

To alleviate this situation, researchers mainly in the database community started to develop languages [CM98][AM98] especially designed to write wrappers. Compared to general-purpose programming languages, these languages provide more expressive power and reduce the effort needed to write a wrapper. But major drawbacks still persist: wrappers are constructed manually; to write a wrapper, previous knowledge of the structure of the web documents is required.

To further help users to find the extraction rule, a lot of research work has been done [LPH00][SA01][BFG01][CMM01][Kus00a][HD98a][MMK01]. They either provide tool support in the process of extraction-rule discovery or they provide new techniques that can semi-automatically or automatically learn the extraction rule. Research work toward automating the process of wrapper construction can be loosely divided into two

categories. The first school of research is called wrapper induction [KWD97][Kus97]. Nicholas Kushmerick formulated the automatic wrapper construction problem as an inductive learning problem. By providing the learning algorithm with some labeled training examples, the learning algorithm generates delimiter-based extraction rules that can later be used to extract data from web document. The second group of wrapper construction research makes use of the structural information of HTML documents. They usually represent the HTML document as some kind of tree structure in memory and then interact with user to generate extraction rules. The generated rules usually contain path information of the tree.

Following, we will review some of the major works of each category.

2.1 Wrapper Induction

2.1.1 Kushmerick's work

Kushmerick [KWD97][Kus97][Kus00a] formalized the wrapper construction problem as an inductive learning problem enabling the automatic generation of extraction rules. He described six different classes of wrappers each appropriate to a certain kind of web documents (or other documents). For each of those classes of wrappers, a machine-learning algorithm is provided to learn the extraction rule.

The general idea of this work is as follows:

- A document is viewed as a sequence of characters.
- One or more tuples with fixed number of attributes exists in each document. A tuple is roughly corresponding to a record in the backend database.
- The training example is labeled. A labeled example is represented by a matrix. Each row of the matrix is a K element vector. Each element of this vector is a <start, end> pair, representing the start and end position within the example document of the corresponding attribute value. The labeling process basically identifies two things:
    - It distinguishes each tuple from one another;

- It distinguishes each attribute from one another within a tuple.

- With a set of labeled training examples, the algorithm tries to find a vector of delimiter string that could be used to extract each tuple out of the document.

Consider for example the following example document from [Kus00a]:

```
<HTML><TITLE>Some Country Codes</TITLE><BODY>
<B>Congo</B> <I>242</I><BR>
<B>Egypt </B> <I>20  </I><BR>
<B>Belize </B> <I>501</I><BR>
<B>Spain  </B> <I>34  </I><BR>
</BODY></HTML>
```

Figure 4: Example document.

There are four tuples in this example document, each tuple with 2 attributes i.e. "country name" and "country code". This document could be wrapped by the simplest kind of wrapper – LR wrapper. "L" and "R" represent "left" and "right" respectively. The task of the LR wrapper-learning algorithm is to find two delimiters (a "left" delimiter and a "right" delimiter) for each attribute in a tuple. In this example, the learning algorithm needs to find 4 delimiters. One possible learning result will be l1=<B>, r1=</B>, l2=<I>, r2=</I>. Using those four delimiters and two operation Skip(li) and Extract(ri) the wrapper can extract the *ith* attribute out of a web document.

The WIEN (Wrapper Induction Environment) tool implemented some of Kushmerick's wrapper learning algorithm. By providing the tool a set of labeled examples and selecting which kind of wrapper is appropriate for these examples, the tool can generate a specific kind of wrapper that can be used to extract data. The wrapper generated by WIEN is essentially a linear finite-state transducer (FST). Each attribute is a state with two possible out-going edges, one for extracting texts and the other for skipping to the next state. The input to the Skip() operation is the delimiter.

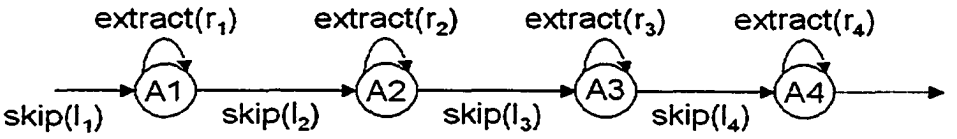

Figure 5: WIEN wrapper.

Kushmerick's work is seminal in that it is the first one to propose the use of inductive-learning techniques to solve the automatic wrapper construction problem. Furthermore, it identified six types of wrapper classes and corresponding wrapper-learning algorithms. The major limitations of this work are as follows:

- It needs manually labeled training examples.
- It only deals with tuples with fixed number of attributes. If there are missing attributes or the order of the attributes varies the technique fails.
- It can only deal with flat tuples and cannot deal with data with nested structures.

If we compare ServiceBuilder to WIEN, we can see that the ServiceBuilder in general solves the above mentioned problems: it does not need labeled training examples and it can deal with nested structure and missing attributes as well. This is not to say ServiceBuilder is always "better" than WIEN, as those two tools have different assumptions. WIEN assumes the flat fixed attributes tuple structure of the target web documents, where ServiceBuilder assumes the existence of the "landmark" word in the target web document. If the target web documents happen to satisfy the first assumption and not the second, then ServiceBuilder will fail where WIEN will succeed. Another difference between ServiceBuilder and WIEN is that the "atomic" building block of delimiter is different. ServiceBuilder uses "token" i.e. HTML tag or "word" as the basic elements of delimiter while WIEN use "character" as the basic element, as it is learned the delimiter "character" by "character." Thus, in extreme cases, it is possible that a "perfect" (100% support) wrapper cannot be found with ServiceBuilder but could be found with WIEN.

2.1.2 SoftMealy

Similar to WIEN, SoftMealy [HD98a] is a wrapper induction tool that can generate data extraction wrappers represented as a finite-state transducers. But unlike WIEN that only generates linear FSTs, SoftMealy can generate non-linear FSTs. In other words, in the FSTs generated by SoftMealy, it is possible to have more than one successor states. As each state represents an attribute, this means that SoftMealy can deal with missing attributes and varying order of attributes in the training samples. Another difference between SoftMealy and WIEN is that, SoftMealy replaced literal "delimiters" used in WIEN with a more abstract artifact called "contextual rules" to locate the attribute values of interest.

The basic concepts of SoftMealy is as follows:

- Token: a segment of input string. HTML tags, numbers, words, punctuation marks are all tokens. A token is denoted as t (v), where t is a token class and v is a string. For instance, a string "123" in a web document is denoted as Num (123), which means this is a number token, and its value is 123.
- Separator: the invisible borderline between two tokens.
- Dummy attribute: a sub-string we want to skip; denoted as –k if it following the k attribute.
- Contextual rule: a sequence of tokens t(v) and its generalized form t(-), which denotes any token of class t (e.g. Num (-) denotes any number). Contextual rules are used to characterize a set of individual separators that separate two adjacent attributes.

Take Figure 6 for an example. It contains two records of personal information of a certain department. Each record contains several attributes that are already marked out by rectangles with the attribute name. For example, the first record contains attributes like "URL", "Name", "Academic title" and "Admin title".

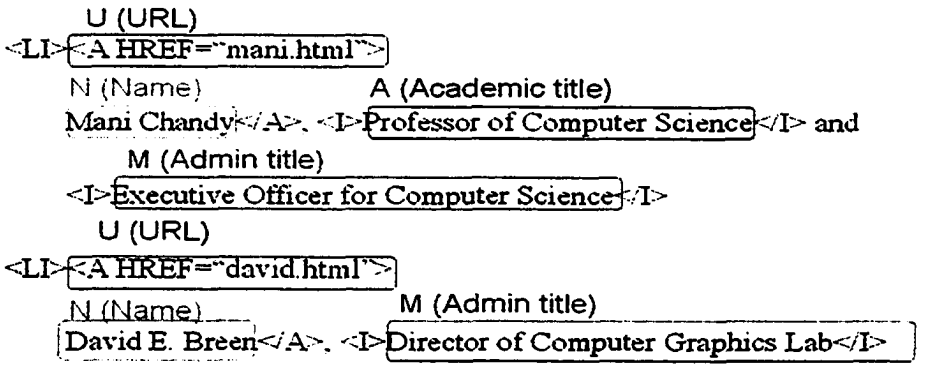

Figure 6: An example document (modified from [HD98b])

The separator between "<I>" and "Professor" can be characterized by the following two contextual rules:

- LeftRule: Html(<I>). This means before the invisible separator is a HTML tag <I>;
- RightRule: C1Alph(Professor) Spc(1) Oalph(of). Token class "C1Alph" represent a string starts with a capital letter; "Oalph" means a string starts with a lower case letter. Spc(x) suggest x number of space characters. So this right contextual rule suggests that after the invisible separator is a "Professor of" string.

Unlike WIEN-generated wrappers where each state represents an attribute, states in SoftMealy wrappers represent either an attribute or a "dummy attribute". A "dummy attribute" basically represents a state where the wrapper skips a series of strings until the input is a contextual rule that leads to a new "attribute" state. As the example documents are already labeled and the operations in each state are fixed (extraction in attribute states and skipping in dummy attribute states), the only thing that needs to be learned to construct a SoftMealy wrapper is to learn the contextual rules that characterize those separators of each individual attribute. SoftMealy uses a generalization algorithm to learn those contextual rules from the labeled examples. Here we do not present the details of the learning algorithm, instead we just demonstrate how SoftMealy wrappers can deal with missing attributes and different attributes situations. The FST wrapper shown in Figure 7 is learned from the example document in Figure 6.

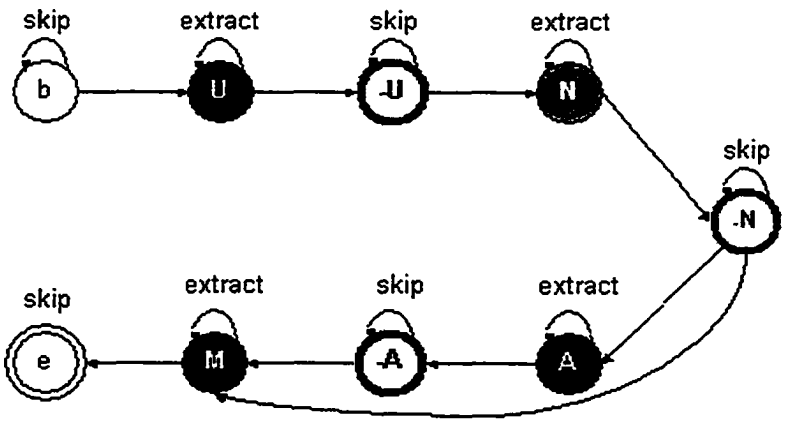

Figure 7: SoftMealy FST wrapper (modified from [HD98b]).

Comparing Figure 7 with Figure 6 we can see that the missing of attribute "Academic title" in the second record in the example document is simply reflected by a shortcut transition edge from state –N to M. That is to say, if a new instance of mutation of attributes appears in the example document, a new transition edge or new states (in case new attributes exist in the new record) will be added to the existing FST.

SoftMealy is a big improvement as compared to WIEN. It solves the missing and variant attribute order problem of WIEN by changing from linear FST to non-linear FST. But it still needs labeled training examples and assumes a flat tuple structure of the web document. Compared to ServiceBuilder, SoftMealy - like WIEN - does not need the appearance of "landmark" word that adjacent to the attributes to be extracted. ServiceBuilder basically sacrifices the generality of the learning approach in favor of removing the labeling process of the training examples. This trade off is well justified since in dynamically generated web documents those kinds of "landmark" words are usually available. Furthermore, nested structure is not a problem to ServiceBuilder.

2.1.3 Stalker

Stalker [MMK01] is a wrapper induction algorithm that was originally developed in the information agent framework Ariadne [KMA+98]. It further developed the techniques used in WIEN and SoftMealy and the generated wrapper is more general and can deal with hierarchical data extraction.

In Stalker, a new kind of formalism called embedded catalog tree (ECT) is developed, which can describe the structure of a wide-range of semi-structured documents. The ECT description of a web document is a tree-like structure in which leaves are the relevant data of interest to the user. The internal nodes of the ECT represent lists of tuples. Each of those tuples can be either a leaf node or another embedded list. To learn the wrapper, Stalker requires the following two inputs:

- A ECT description of the structure of the target documents;
- A set of training examples in the form of sequence of tokens representing the surroundings of the data to be extracted.

The Stalker wrapper induction algorithm uses a greedy strategy to generate data extraction rules. It first tries to learn a rule that covers as many training examples as possible. After this, it deletes all the covered examples. If there still exist uncovered examples, it repeats the learning process again and generates a new disjunctive rule. This process continues until there are no more uncovered examples. The result of this learning process is a set of disjunctive rules that can be used in the data extraction process.

The wrappers generated by Stalker are more general than the ones generated by SoftMealy. For example each disjunctive rule in SoftMealy is either a single SkipTo(), or a SkipTo()SkipUntil() combination in which two contextual rules must match immediately after each other, while in Stalker, a disjunctive rule could be multiple SkipTo() and SkipUntil() combinations, thus more expressive than SoftMealy. Compare to WIEN and SoftMealy, the Stalker wrappers can extract objects inside nested structure in a web document.

Though Stalker generate wrappers that more expressive than that of WIEN and SoftMealy, it suffers its own limitations as compared to ServiceBuilder. First like WIEN and SoftMealy, Stalker requires that the user prepare the training examples. It provides a GUI to help the user to prepare each individual training example. The GUI does somehow relieves some human error, but the human involvement in making the training examples implies that using a large amount of training examples is not practical. The

second problem of Stalker is that it needs the explicit knowledge of HTML structure of the target documents expressed in the form of ECT tree as input. This is the price of having the ability to extract objects in nested structures. ServiceBuilder does not need the explicit knowledge of the structure of the source documents and can still deal with objects in nested structure. However, both SoftMealy and Stalker are better than ServiceBuilder in that they have the concept of "token class" which provides a higher level of abstraction than "token". The token class corresponds to the wildcard in regular expressions. The use of "token class" in some cases can dramatically reduce the number of extraction rules needed to cover all the training examples.

2.2 HTML Structure based approaches

2.2.1 Xwrap

Xwrap [LPH00] is a wrapper construction tool that shares the similar architecture with ServiceBuilder. It consists of the following four components:

- Syntactical Structure Normalization component that is responsible for fetching, cleaning a specific web document and generating a internal parse tree of the document;
- Information-Extraction component that is responsible for displaying the document in a tree format and interacting with the user step by step to explore and generate the extraction rule;
- Code-generation component that generates java code which implements the wrapper; and
- Testing component that run the generated wrapper on other user specified documents to test whether the wrapper could extract the correct data from other similar documents or not.

Among those four components, the Information-Extraction component is the most important one. It displays the source document to the user in an XML-tree graph. Then the user selects semantic tokens that are of interest to them from the tree and also identifies the structure within the tree that contains all the data of interest and this

component will generate a set of data extraction rules. After that the user will use the code generation component to generate a Java implementation of the wrapper and use the Test components to test the generality of the new generated wrapper.

Xwrap provides an interactive environment for the wrapper constructor to interactively build a wrapper without actually coding. But except for calculating the corresponding path to the semantic token, the system does not really try to generalize these paths into extraction rule. In other words, these rules are specific for the document where they are generated. If a similar document has a slightly different tree structure, then the learned rule may fail and Xwrap does not provide a mechanism to get a more general rule that could work on both documents.

2.2.2 W4F

W4F [SA01] (The World Wide Web Wrapper Factory) was developed in the University of Pennsylvania. This tool translates an example document into a DOM tree in memory. Each inner node in the tree is a HTML tag and each leaf node is a piece of data contained in the document. Based on this kind of representation, each piece of extraction rule is a path either absolute or relative in the tree that leads to a leaf node. Extraction rules are explicitly specified by the user in a high-level scripting language called HEL (HTML Extraction Language). Even though the tool displays the corresponding path of the piece of data when a user points the mouse to it in an editor, it still relies largely on the user to figure out and specify in HEL a general extraction rule. If all the target documents share the exact same HTML structure, then this tool works fine. But when there are structure variations among web documents, especially when the variation is large, it is hard or even impossible for the user to generalize a general extraction rule. So, in the sense of extraction rule "learning", W4F demonstrates no "intelligence" at all. But compared to Xwrap, W4F provides the scripting language that could be used by the user to write general data extraction rules.

2.2.3 Lixto

Lixto [BFG01] is an interactive wrapper-generator tool that also relies on the tree structure of HTML. But different from W4F, Lixto does not only rely on path information to get the extraction rule but can combine other constraints. It displays an example web document in an interactive GUI and lets the user indicate the interesting data items. After the user indicates a piece of data, the tool will try to generalize the underlying pattern and highlight all the instances in the example that satisfy the same kind of pattern. Depending on the result, the user can iteratively refine the pattern until the highlighted instances are only those that the user interested in. Now the user can save the last pattern as a data-extraction rule. Behind the scenes, an internal Elog script program is generated and saved, which can be executed by the Lixto executor component later to extract data from similar documents.

Compared to W4F, Lixto demonstrates "intelligence" in helping the user generalize the rule within a document. But this kind of help is limited, as it only tries to generalize one rule at a time. To generate all the rules that could cover the whole complex object of interest multiple rounds of trial and error are necessary. For documents that contain multiple complex objects, this process can be very time consuming.

2.2.4 RoadRunner

Before talking about RoadRunner [CMM01], let's briefly summarize some limitations the previous introduced techniques have. First, HTML structure based techniques like Xwrap, W4F and Lixto are based on a single example document. As a result, the generated wrappers are specific to that document. Even though some systems like Xwrap support the user to test the generated wrappers on other documents, they do not provide any assistance in generalizing the wrappers among multiple example documents. This characteristic limits their applicability to the general problem case of generating wrappers across multiple documents. Second, all the wrapper induction techniques we reviewed are essentially supervised learning techniques, i.e. labeled training examples are required. As preparation of these training examples is tedious and can be error-prone, unsupervised learning is preferred to supervised learning in this case.

RoadRunner is a wrapper-learning algorithm that does not have the two limitations mentioned above. It can learn wrappers from two or more example documents that belong to the same "document class". The basic idea behind RoadRunner is that a template can be found by comparing two documents and finding similarities and differences between them. The similarities or the invariants are the common HTML structures in the example documents; the differences are the actual data populated to the structure from the backend database.

But unlike other HTML structure based techniques, which represent the example document as a tree structure in memory, RoadRunner represents each example document as a string consisting of a sequence of tokens. It deems the dynamic document generation process as an encoding of the database content into string of HTML code and hence, the data extraction process as a reverse decoding process. The wrapper to be learned is a union-free regular expression that represents the HTML structure that contains database contents.

Tokens in the example documents are divided into two classes: HTML tags and ordinary strings. To learn the wrapper, RoadRunner use a match technique called ACME (Align, Collapse under Mismatch, and Extract). This algorithm works on two objects at a time:

- A sequence of token called sample;
- A wrapper, in other words a union-free regular expression.

Before the execution of the algorithm, the sample and the wrapper are initialized with the two example documents. Then the algorithm is trying to find a common regular expression for the two documents by sequentially comparing tokens from each object and trying to solve token mismatches between the wrapper and the sample. When a mismatch happens, the algorithm first determines the type of mismatch. If the mismatch is a string mismatch, i.e. a mismatch between two string tokens then this token spot is marked as #PCDATA, i.e. it is a contents comes from the database. If the mismatch is a tag mismatch, the algorithm will first try to search whether it is a start of a repeated pattern. If the search is successful, a generalization pattern is inserted in the wrapper otherwise

the algorithm will mark one of the mismatched tags (determined by a bit more further search in both wrapper and sample) as an optional tag. The result of dealing with mismatch is the generalization of the wrapper, i.e. either find a variable slot #PCDATA or find an optional tag or find a repeated pattern.

The major limitation of RoadRunner is that when string mismatch happened in place other than data of interest, RoadRunner would wrongly marked that spot as a piece of data of interest. This is based on RoadRunner's basic assumption: all string mismatches indicate a place corresponding to a variable in the backend script. RoadRunner relies on this assumption to discover data of interest and thus does not need labeled training examples. ServiceBuilder, on the other hand, rely on landmarks to help locate data of interest and hence avoid this problem. Of course, this comes with a price - the loss of fully automation, as in ServiceBuilder the user needs to specify a set of landmarks before the learning process starts. However, we believe this tradeoff is justified as with a little bit loss of fully automation, the accuracy of the generated wrapper is much more improved. Instead of returning data mixed with a lot of irrelevant information, the ServiceBuilder only returns those pieces of data that the user really wants at runtime.

2.3 Summarizations and Comparisons with ServiceBuilder

In this section, we summarize the works of HTML structure based wrapper construction techniques and compare them with ServiceBuilder. In general, the works we introduced in section 2.2 can be roughly classified into two categories:

- Tree path based approach: Xwrap, W4F and Lixto;
- Token sequence based approach: RoadRunner.

Though fundamentally both of them leverage the HTML structure invariants present in the examples to get the extraction rule(s), they attack the problem in different ways. The first approach (Xwrap, W4F and Lixto) uses the tree-structure information directly. These tools model a web document either as a DOM tree or a similar tree structure in memory. Data of interest are leaves in the tree and the internal nodes of the tree are the HTML elements. The objective is to find a generalized path expression to the data of interest.

This path expression is either an absolute path or a relative path to a certain HTML element. At run time, the wrapper translates a web document into a tree and then uses the path to find the relevant leaf node from the tree and extract the data out.

On the other hand, the second approach (RoadRunner) deems the web document as a sequence of tokens (HTML tokens and data tokens). Its goal is to generalize a sub-sequence of tokens from the input sequence that contains the data of interest. In this sense, RoadRunner is similar to WIEN, SoftMealy and Stalker. But RoadRunner makes explicit use of knowledge of "structure" information of HTML by treating "HTML tag" mismatch and "data string" mismatch differently.

In general, the token-sequence approach is better than the path-based approach, as a sequence of tokens could be translated to a context-free structure in the whole tree. It can be seen as a path from anywhere in the tree to a structure that contains the data of interest.

The major limitation of the path-based methods is that all of them are based on a single example. The extraction path is generalized to cover multiple objects within the same document, and does not necessarily cover different documents. This greatly affects the practicality of using this kind of techniques to build wrappers for web applications whose output documents do exhibit structure variations. On the other hand, RoadRunner is based on two or more training examples, thus the generated wrapper is more general than those tree path-based approaches.

Compares to the wrapper construction techniques we reviewed in this chapter, ServiceBuilder has the following features that make it suitable to generate wrapper for web applications:

- The extraction-rule learning process is automatic. This is a characteristic shared with most wrapper induction techniques. All the HTML structure-based techniques except RoadRunner are interactive: they either rely totally on the user or they require the user's help to learn the data extraction rule.
- It does not need annotated training examples. All the wrapper induction techniques we reviewed need labeled training examples to guide the learning

algorithm to generate the data extraction rule. ServiceBuilder does not need labeled training examples. Instead, it relies on a set of landmarks to direct the system to generate a small set of candidate extraction rules. Compare to labeling a large amount of training examples, preparing a set of landmarks is a fairly simple task.

- The extraction rules generated by ServiceBuilder are based on a large number of training examples. All other tools – except WIEN - generate extraction rules that are only based on one or two documents. The fewer the examples on which the extraction rules are based, the more fragile usually the wrapper is. From our observations, any tool that generates extraction rules based on a HTML tree structure suffers from this problem.
- The extracted data structure is not explicitly expressed but is rather implicit in the way the user defines the complex type. This enables it to extract complex structured data from a web document.
- The generated extraction rules are not based on a rigid tree structure of the source document but rather rely on the combination of a flexible (as certain HTML tags can be intentionally excluded from the interesting delimiters set) sequence of HTML tags and the landmarks to define extract rules, which makes the resulting wrapper more robust to website changes.

A table summarizing comparisons of the wrapper construction tools reviewed in this chapter is attached as Table 12 in Appendix.

Chapter 3 The ServiceBuilder System: Architecture and Process

The ServiceBuilder system consists of three major components: the *data retriever*, the *pattern learner* and the *code generator*. Figure 8 illustrates the architecture of ServiceBuilder and how it works in the context of wrapping a web application into web services. We will examine each component in detail in the rest of this chapter and we will illustrate their individual functionalities and the overall ServiceBuilder process with an example of wrapping a stock-quoting web application.

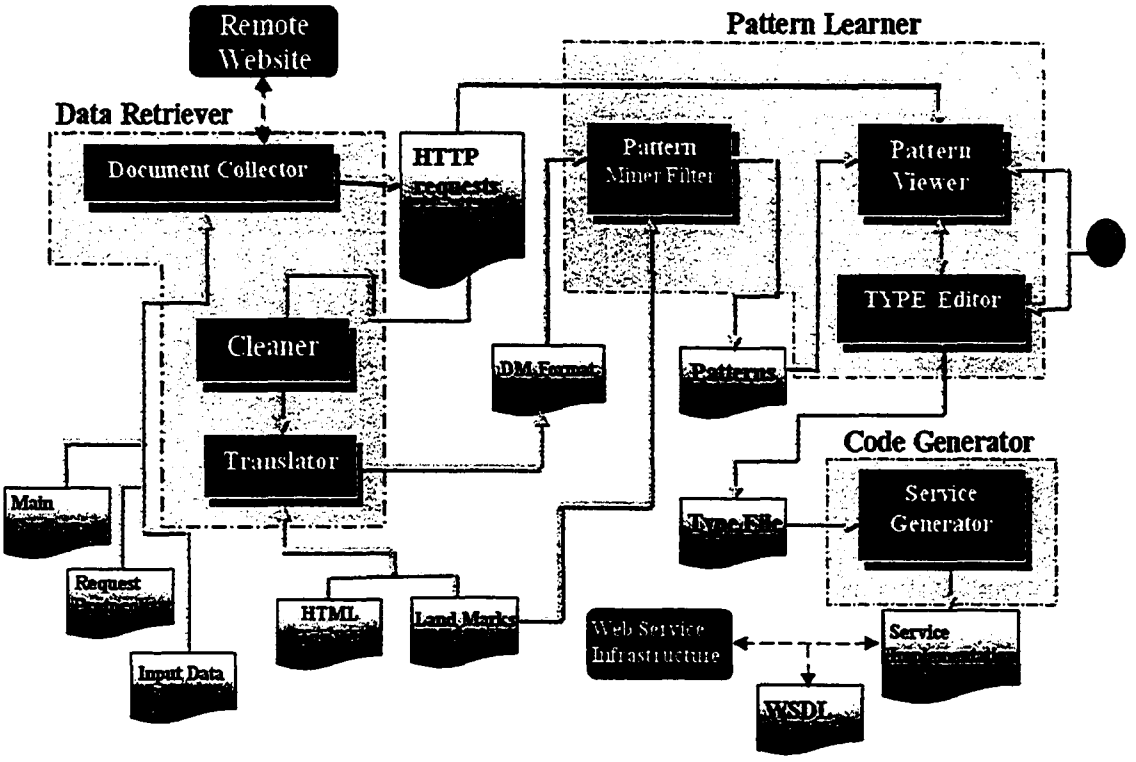

Figure 8: The overall architecture and process of the ServiceBuilder tool.

3.1 The Data Retriever

The *data retriever* component is responsible for interacting with the target web application and transforming the output of the web application into a suitable format ready for the *pattern learner* component to consume. It consists of three sub-components: the *document collector*, the *cleaner* and the *translator*.

3.1.1 The Document Collector

This component simulates the behaviors of a web-application user interacting with the web application with proper input data and saves the responses of the web application as HTML documents. For the *document collector* to work correctly, it needs to be properly configured. There are three types of configuration files that define the behavior of *document collector*, i.e. *mainConfig.xml*, *requestProtocol.xml* and *inputData.xml*.

The *mainConfig.xml* is the master configuration file of the *document collector* component. Only one *mainConfig.xml* exists in the system. The schema of mainConfig.xml is shown in Figure 9. From the schema, we can see that *mainConfig.xml* consists of one or more *site* elements. Each site element represents one web application to be reengineered. There are four sub-elements of a *site* element.

- The *siteName* element gives a unique descriptive name to the target web application;
- The *requestProtocolLoc* element specifies the file handle to the second kind of configuration file i.e. requestProtocol.xml that specific to this web application;
- The *outputLoc* element specifies the directory in the local file system where the collected HTML response documents from this web application should be stored; and
- The *inputSet* element specifies the third kind of configuration file, i.e. *inputData.xml* that describes the "test data" based on which the requests to the web application will be formulated.

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd='http://www.w3.org/2001/XMLSchema'>
<xsd:element name="mainPCConfig">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="site" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="site">
  <xsd:complexType>
    <xsd:sequence>
      <!-- the name of the site -->
      <xsd:element ref="siteName" minOccurs="1" maxOccurs="1"/>
      <!-- the location of the request protocol -->
      <xsd:element ref="requestProtocolLoc" minOccurs="1" maxOccurs="1"/>
      <!-- the location where the output will be put -->
      <xsd:element ref="outputLoc" minOccurs="1" maxOccurs="1"/>
      <!-- input set file -->
      <xsd:element ref="inputSet" minOccurs="1" maxOccurs="1"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="siteName" type="nonEmptyString"/>
<xsd:element name="requestProtocolLoc" type="nonEmptyString"/>
<xsd:element name="outputLoc" type="nonEmptyString"/>
<xsd:element name="inputSet" type="nonEmptyString"/>

<xsd:simpleType name="nonEmptyString">
  <xsd:restriction base="xsd:string">
    <xsd:minLength value="1"/>
  </xsd:restriction>
</xsd:simpleType>

</xsd:schema>
```

Figure 9: mainConfig.xsd

A *requestProtocol.xml* file describes the HTTP request protocol used by the document collector to access the target web application. Each target web application has its own *requestProtocol.xml* file. An example *requestProtocol.xml* file is shown in Figure 10. The document's root element is the *request* and it consists of a *method* and a *form* sub-element. The *method* element has two mandatory attributes: "type", which specifies the type of the HTTP request, in this example GET, and "url", which specifies the location of

the web application. The *form* element consists of one or more *parameter* sub-elements. Each *parameter* element corresponds to an input variable of the request: the "name" attribute indicates the variable name of this input parameter in the HTTP request and the "value" attribute indicates either an element name in the corresponding *inputData.xml* or a literal value depending on the third attribute – "input". If *input* equals "yes", the actual input value will come from elements of the corresponding *inputData.xml* file, and the corresponding element name is given by the value of the second attribute "value". If the input attribute is absent, then the value of the "value" attribute is deemed literal. Figure 10 specifies the syntax of the HTTP request to the yahoo web site to obtain the current value of a stock symbol. It is a GET request with two input parameters, named "s" and "d" respectively. Variable "s" will take a different value from the "symbol" element of the corresponding *inputData.xml* file with each HTTP request while "d" will use the literal value "v1" as its value every time.

```
<?xml version="1.0"?>
<request>

  <method type="GET" url="http://finance.yahoo.com/q"/>

  <form>
    <parameter name="s" value="symbol" input="yes"/>
    <parameter name="d" value="v1"/>
  </form>

</request>
```

Figure 10: Example requestProtocol.xml

An *inputData.xml* file is the third kind configuration file that the *document collector* needs. It contains the actual data to be used in "testing" the web application, i.e., in formulating requests to it according to the request-protocol specification. Following the example above the corresponding *inputData.xml* file is shown in Figure 11. It consists of a sequence of input elements- "*symbol*" as specified in the requestProtocol.xml file (see Figure 10).

Base on the information provided by the *requestProtocol.xml* and *the inputData.xml*, the *document collector* component can generate a series of HTTP requests to the target web application and store the response HTML documents in the directory specified in the *mainConfig.xml* file.

```
<?xml version="1.0">
<inputSet>
  <symbol> MSFT </symbol>
  <symbol> AMD  </symbol>
  <symbol> IBM  </symbol>
  <symbol> KRK  </symbol>
  <symbol> KVM  </symbol>
  <symbol> ORCL </symbol>
  <symbol> HP   </symbol>
  <symbol> INTC </symbol>
  <symbol> AAPL </symbol>
  <symbol> SGI  </symbol>
</inputSet>
```

Figure 11:An example of inputData.xml file.

3.1.2 The Cleaner

This component consumes the documents collected by the *document collector* and deletes possible scripts embedded in them and cleans up potential mismatched HTML tags and other syntactical errors they may contain. The output is a set of well-formed script-free HTML documents.

The HTML document cleaning process is based on [JTidy], a robust and easy to use open source HTML syntax checker. By setting up the configuration file properly, JTidy can find and fix HTML syntactical errors in the document.

3.1.3 The Translator

The purpose of the translation process is to convert the collected clean HTML responses into a format suitable for the subsequent pattern-mining process, the so-called DM format, by eliminating all unnecessary information contained in them. The translation process is governed by two configuration parameters. The first one is a list of "*interesting delimiters*", identifying the HTML tags that will be retained. Intuitively, this list makes explicit some tacit knowledge about the domain of HTML-document design: usually, designers highlight interesting output information within HTML tags, such as tables and lists, or with a distinct font attribute, such as color or italic or boldface. Other HTML elements, such as images and unstructured paragraphs usually contain peripheral information, not directly related to the output expected by the user issuing the request.

By default, the interesting delimiters of ServiceBuilder are set to the set of valid HTML tags. As long as the ServiceBuilder can find useful patterns with the presence of some "noise" tags that are irrelevant to the output data, we do not remove them from the interesting delimiters even though their existence may cause some performance penalties for the pattern-learning algorithm. We only remove those "noise" tags from interesting delimiters when their presence actually prevents ServiceBuilder from finding useful patterns. The reason we adopt this "lazy" philosophy is that we do not want to subjectively decide beforehand which tags are "interesting" and which are not. Actually in the experiments described in Chapter 4, we only remove the tag <img>. Of course, if we can remove most noise tags from the interesting-delimiters set, the mining algorithm will be more efficient as the average input document length will be greatly reduced.

The second configuration parameter is the "*landmark*" list. A landmark is defined as a word or phrase frequently used in a specific application domain. For example, in the stock-quote domain, the following phrases are usually found: "last trade", "market", "bid", "open" ...etc. Intuitively, these landmark phrases are expected to be used as labels in close proximity to the output information of the web-application HTML responses. The landmarks for our stock-quote service-building example are shown in Table 11 of Appendix.

Once the interesting delimiters and landmarks are determined, the system maps each element of those two lists into a unique integer based on which list they belong to and their relative position in the list. The *translator* then converts all the collected clean documents into one token list. Here a token refers to either a HTML tag or a string between HTML tags. Then it iterates through this list: if the token is a HTML tag that belongs to the interesting-delimiter list or a string that contains an item in the landmarks list, then a unique integer corresponding to the interesting-delimiter or the landmark is inserted into the output; otherwise, the token is simply discarded. Thus, the delimiter HTML tags and landmark phrases are the only parts of the original response content retained in the DM format.

3.2 The Extraction-Rule Learner

After the web-application's responses have been collected and translated to the DM format, the next step of the service-building process is the mining of the data-extraction rules[1]. The extraction-rule learner consists of three sub-components: the *pattern miner*, the *pattern viewer* and the *type editor*.

3.2.1 The Pattern Miner

The *pattern miner* is responsible for generating the candidate patterns for data-extraction rules to be inspected by the user. Its first function is to use two mining algorithms to extract all repetitive patterns in the collected documents. As a result, a series of patterns that are "frequent" in the input are generated. We call these patterns "raw patterns", as most of them do not contain data that are of interest to the user.

To eliminate those unsuitable patterns, the whole set of raw patterns is forwarded to the subsequent filtering step. The filter is equipped with a set of heuristics that aim to eliminate those patterns that do not contain data of interest to the user. Those heuristics can be configured by the user to run individually or pipelined to run sequentially.

---

[1] We use the word "rule" and "pattern" interchangeably in this thesis.

Let us now discuss in more detail the mining algorithms and the heuristics of the pattern-miner component.

3.2.1.1 Sequitur

The first data-mining algorithm of the miner component is Sequitur [NMW97]. This algorithm compresses a string into a context-free grammar (without recursion) by inferring the grammar from the string. If there is structure and repetition in the input string then the grammar may be very small compared to the original string, and the composition rules of the grammar capture essentially the frequently repeated subsequence in the original string. Sequitur relies on two intuitive rules: First, that no pair of adjacent symbols (diagram) should appear more than once in the grammar (instead it should be substituted with a composition rule) and second, every production rule should be used more than once (there should be no non-repeatable rules).

As the desired patterns need to meet a minimum occurrence threshold, in the actual implementation of the pattern miner, we wrap the Sequitur algorithm with a frequency calculator component. After running the Sequitur algorithm on the input data in DM format, we get a group of composition rules. Each rule corresponds to a repeated subsequence in the input data. Then the frequency calculator calculates the actual number of occurrences of each rule in the input data and removes those not satisfy the minimum occurrence threshold.

The original reason we choose Sequitur to implement the pattern mining is that it is very fast compare to the IPM algorithm introduced in the next section. However, pilot experiments [JS04] showed that the actual number of useful patterns found by Sequitur is very low, and all the patterns found by Sequitur could also be found by IPM, as the later is an exhaustive pattern-mining algorithm. After the pilot experiment, we re-implemented the IPM algorithm and inserted some constraints into the inner loop of the algorithm and the speed of IPM was greatly improved, so its efficiency shortcomings are not as grave and its effectiveness advantages far more outweigh them. As a result, the Sequitur

algorithm is deprecated now in the ServiceBuilder. We introduce it here only for historical reasons.

3.2.1.2 IPM

The second pattern-mining algorithm of the pattern miner is IPM [ERSS02]. IPM is a sequential pattern-mining algorithm, designed to discover patterns with insertion errors, i.e., patterns whose instances may not be exact replicates of the pattern itself but may contain a certain number – below a configurable threshold – of extraneous alphabet characters. This feature makes it especially suitable in situations where the input sequences may be noisy and a certain degree of flexibility is desired when inferring a pattern.

IPM is based on Apriori [AS94] in that it starts with short patterns and it proceeds to expand them in order to identify longer ones. However, unlike Apriori that identifies item sets, IPM identifies sequential patterns with a pre-defined number of insertion errors. IPM is an exhaustive pattern-mining algorithm, i.e., it can find all the patterns that exceed the minimum occurrence threshold.

In our implementation, in order to further improve the run time efficiency of IPM algorithm, we put some domain specific constraints into the algorithm. For example, all candidate patterns that contain <html> or </html> tags in its middle are immediately pruned and are not used to generate longer patterns, since in our problem domain, we are not interested in patterns across the HTML document borders.

3.2.1.3 Filtering Heuristics

In practice, the pattern-mining algorithm usually outputs a large number of patterns, often in the order of several hundreds, and most of them do not contain data of interest to the user. If all these patterns were forwarded to the pattern-viewer component, it would result in a nightmare for the service developers, as they would have to sift through these hundreds of patterns to locate the ones that actually contain data of interest.

Thus, ServiceBuilder employs two heuristics to help filter out those irrelevant patterns. We use landmarks as the indicators of the degree of relevance of the pattern, as landmarks by definition are in close proximity to data of interest. Each of these two heuristics is appropriate in different situations. We will discuss when they are applicable in detail in Chapter 4. Following, we will describe each of the heuristics in detail.

3.2.1.3.1 Minimum Rule Set (MRS) Heuristic

The goal of this heuristic is to find the minimum subset of the input pattern set that can cover all the landmarks covered by the input pattern set. More formally:

- Given a set of patterns S that covers N landmarks, find S' a subset of S, such that
  - S' covers all the N landmarks, and
  - There is no other subset S'' with smaller cardinality that also covers the same landmarks.

This is a classic NP-complete problem known as the "minimum cover set" problem. We have implemented a greedy heuristic, as introduced in [Joh73] to obtain an approximate solution to this problem. The general idea of the heuristic is to iteratively look for a pattern in the input pattern set that contains the largest number of uncovered landmarks and move it from the input pattern set to the solution set until all the landmarks that appear in the input set of patterns are covered. Even though there is no analytical upper bound to the number of cycles of the process iterations, in practice our experiments (see section 4.1.3) have shown that it works quite well in this application.

3.2.1.3.2 Maximum Common Sub-pattern (MCS) Heuristics

To explain this heuristic, we need to introduce the "effective sub-pattern" concept.

- An effective sub-pattern of a pattern *P*, denoted as *esp(P)*, is the sub-pattern of *P* that starts with the first landmark in pattern *P* and ends with the last landmark in pattern *P*.

The goal of *maximum common sub-pattern* heuristic (from now on referred to as MCS heuristic) is to cluster the input patterns into groups so that all patterns in a group have the same effective sub-pattern. Then a single pattern, called the maximum common sub-pattern, can be used to represent the whole pattern group.

The heuristic works as follows:

- First, it calculates the effective sub-pattern of each individual pattern.
- Second, it groups patterns with the same effective sub-pattern into different groups.
- Third, within each group, the algorithm tries to generate a representative maximum common sub-pattern through the following steps:

    1. Pattern alignment: The algorithm aligns all the patterns according to the common effective sub-pattern and the resulting representative maximum common sub-pattern is initialized to the common effective sub-pattern.
    2. Head extension: The algorithm examines the token just before the maximum common sub-pattern of each individual pattern. If all the tokens examined in the pattern group are the same, the token is inserted into the head of the maximum common sub-pattern. This step is repeated until either there is a mismatch or the head of one of the pattern in the group is reached.
    3. Tail extension: A similar technique as head extension is used to extend the maximum common sub-pattern at the tail.

This heuristic tries to use a single representative maximum common sub-pattern of a pattern group to represent all the patterns in that group. MCS is a fairly simple heuristic; however, we will demonstrate with experimental results in Chapter 4 that it quite effective in practice.

3.2.2 The Pattern Viewer

After the pruning of the heuristics, the ultimate output of the *pattern miner* component is a set of "good" patterns, which together cover the parts of the web-application response documents that contain the information of interest to the user of the web application. Each pattern corresponds to a frequently occurring sequence of HTML tags and domain-specific landmarks, which is hypothesized to be a consistently structured part of the HTML response containing some of the desired information output of the request.

However, the user is not necessarily interested in all the output data of the target web application. It is possible that the user actually is only interested in a sub-set of the output data. This is why the "good" patterns generated by the pattern miner are forwarded to the next component, the pattern viewer, which highlights these patterns in the context of the collected examples and lets the user make the final decision regarding which candidate patterns actually contain the data that they are interested in.

The graphical user interface (GUI) of the pattern-viewer is shown in Figure 12. We can see that it consists of three frames. The frame to the left displays a list of candidate patterns (rules) generated by the pattern miner component. In the bracket following each pattern, there is a pair of numbers separated by a "/". The first number indicates the number of example documents that contain this pattern; the second number indicates the total number of example documents used in the pattern learning process. For example the pair "18/20" follows "rule 1" suggests that rule (pattern) 1 appears in 18 of all the 20 example documents used in the pattern learning process. Essentially, this ratio is an indicator of the support of this pattern in the collected document set.

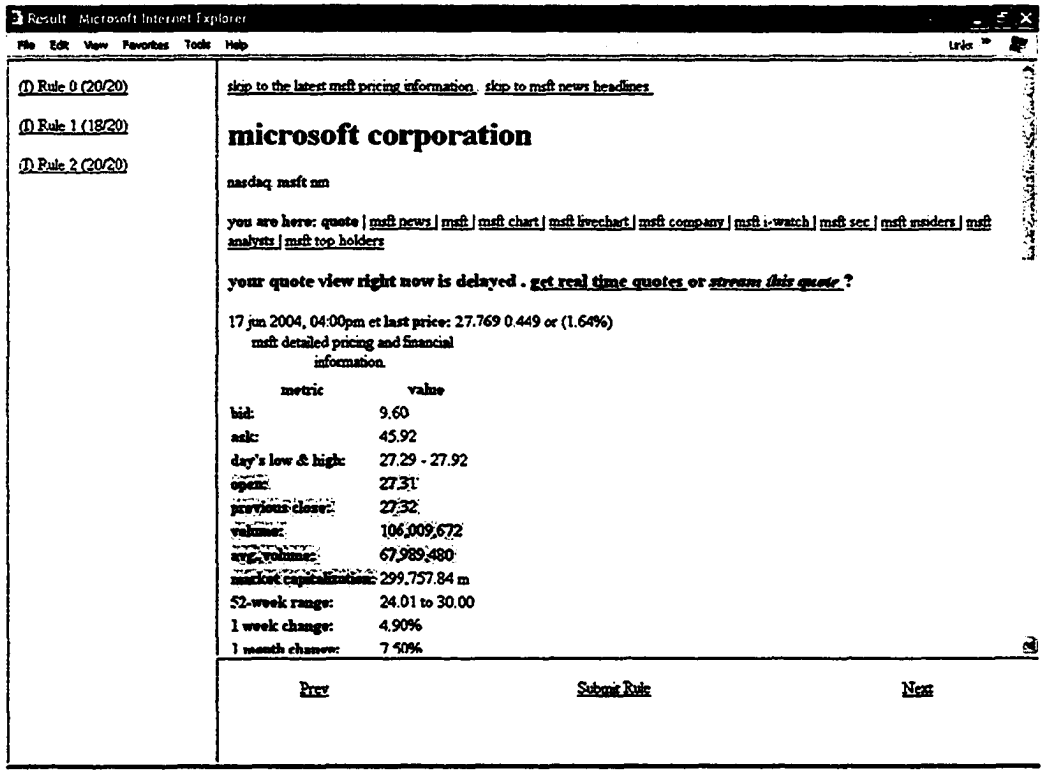

Figure 12: The pattern viewer.

The frame at the right side is used to display a response document returned by the web application, with the areas covered by the selected pattern highlighted. For example, in Figure 12 we see that the selected Rule 2 represents a pattern, whose occurrence in the displayed response document covers part of the tabular structure containing information about "open", "previous close", "volume", "avg. volume" etc.

The frame to the bottom contains a set of controls to select different example web documents used in the learning process. Using the "Prev" and "Next" links, the user can see the occurrence of the same rule in other documents of the collection. In this manner, the user can perceive whether the rule covers consistent parts of the HTML response with information of interest to the user. If this is the case, the pattern is useful for extracting (some of) the data expected as part of the return message of the potential web service.

Finally, the submit button, contained in the middle of this frame, is used to confirm to the system that the current displayed pattern as a valid data-extraction rule.

3.2.3 The Type Editor

Once the user submits a pattern by clicking on the "submit" button in the pattern viewer, the following information is sent to the backend *type server*:

- The current selected rule number; and
- The content of the current displayed document.

Based on the above information, the server calculates the relative position of each piece of the highlighted string in the pages within the selected rule. Then the result is propagated through an event mechanism to a GUI component called *type editor*. Upon receiving the result from the *type server*, the *type editor* activates its *simple type editor* interface as shown in Figure 13.

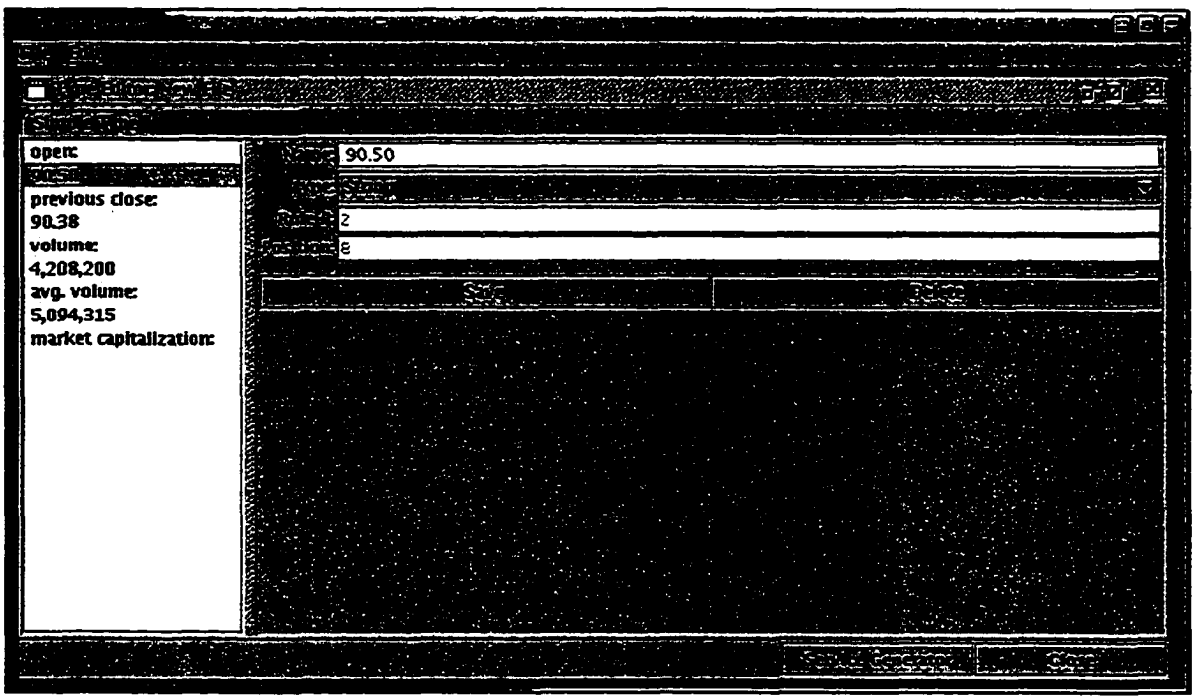

Figure 13: Simple type editor.

Every piece of data highlighted in the pattern viewer is by default listed as a simple type in the left-hand side frame of the type-editor interface. In the right-side frame four types of information are displayed:

- The user can input a meaningful name in the "Name" field for the selected type. For example, in this picture, a number "90.50" is selected in the left-hand side pane, which is the "open price" of the stock displayed in the example document. In this case, the name in the "Name" field may be set to "openPrice".
- The "Type" field indicates what the data type this simple data type is. It can be selected from a predefined group of types such as "string", "int", "boolean" and "float" etc. In this example, the type may be set to "string" or "float".
- The "Rule" field displays a number to indicate which rule the user just submitted through pattern viewer component; this is also the rule that contains those pieces of information listed in the left hand side. This field is provided for maintaining the context of the overall mining process in this type-editor phase: it reminds the user which rule is used to extract the instance of the data type that is currently being edited.
- The "Position" field indicates the relative position of the instance of this data type with respect to the rule instance. Similar to "Rule", this is also an informative read only field.

There also exist a "Save" button and a "Delete" button at the right side pane. After specifying the type name and selecting the appropriate data type, the user can click the "Save" button to save the selected simple data type. The name of the newly created simple data type will appear at the list at the left-hand side pane. The user can use the "Delete" button to delete any undesired simple data types.

Instead of extracting one piece of data from a web document and returning it to the client every time the user may want to extract a set of information out of a single page all at once and pack them together and return to the client as a whole. To accommodate this need, the type editor provides a *complex data type editor* window. Using this window, a user can create *complex data types* by composing those *simple data types* already defined. A complex data type editor window is shown in Figure 14. The use of the complex data type editor is straightforward.

- By default all the simple data types a user has created using simple type editor are displayed in the "Simple Types" field. In the following picture, three simple types are listed, i.e. "openPrice", "PreviousClosePrice" and "volume";
- A new complex type may be defined by an identifier specified in the "Name" field. In this example, the identifier of the new type is "StockInfo";
- All the simple types that will constitute the sub-elements of the complex type may be selected from the "Simple Types" field. Clicking on the "Save" button then results in the newly created complex data type to be displayed in the left hand side pane. In this example, the "openPrice" and "PreviousClosePrice" are the chosen sub-elements of the StockInfo.

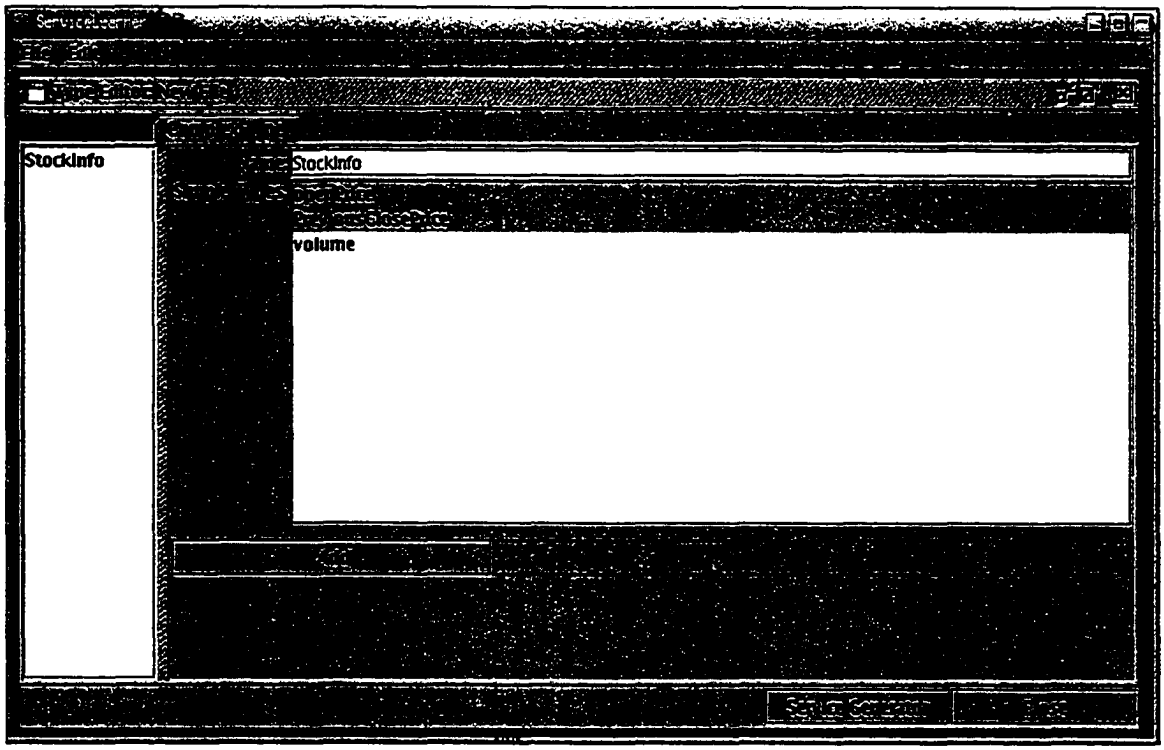

Figure 14: Complex data type editor

A user can create any number of complex data types. After creating all the complex data types, the user can choose to save all the type related information during type editing into a type file. An example type file is shown in Figure 15.

```
<?xml version="1.0"?>
<types xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance
" xsi:noNamespaceSchemaLocation='./types.xsd'>
<service>
   <serviceName>LycosQuotes</serviceName>
   <rulefileName>rules.txt</rulefileName>
   <simpleTypes>
      <simpleType>
         <name>openPrice</name>
         <type>String</type>
         <rule>2</rule>
         <path>8</path>
      </simpleType>
      <simpleType>
         <name>PreviousClosePrice</name>
         <type>String</type>
         <rule>2</rule>
         <path>19</path>
      </simpleType>
      <simpleType>
         <name>volume</name>
         <type>String</type>
         <rule>2</rule>
         <path>30</path>
      </simpleType>
   </simpleTypes>
   <complexTypes>
      <complexType>
         <name>StockInfo</name>
         <property>openPrice</property>
         <property>PreviousClosePrice</property>
      </complexType>
   </complexTypes>
</service>
</types>
```

Figure 15: An example type file.

A type file is an XML file consisting of a number of <service> elements. Each service element contains all the type information about a target web application. The <service> element contains the following sub-elements:

serviceName element: a unique name of the target web application. It is used to distinguish one target web application from another.

ruleFileName element: the name of the corresponding rule file of the target web application. The rule file contains all the data extraction rules learned during learning process.

simpleTypes element: it contains a series of *simpleType* sub-element. Each simple type element contains all the type information we get in the type editing process. This element essentially establishes all the information need to extract different kind of data out during wrapper execution.
complexTypes element: it contains a series of *complexType* sub-element. Each *complexType* element specifies how to using simple type instances to construct a complex type instances.

In short, the type editor is a wizard that guides the user to define the data extraction and packaging rules. More specifically: first, through defining *sample types* it establishes the data-extraction rule for each elementary piece of data on the web documents and associates it with a meaningfully named type; second, through defining *complex types* it establishes the output data encapsulation schema. In other words, it defines how the extracted data will be packed together.

3.3 The Code Generator

The *code generator* component takes the type file as input and produces a set of java classes as the wrapper implementation. The code-generation process consists of the following steps:

- Generation of a bean class: In this step, the code generator extracts the service element from the type file that corresponds to the current target web application. Then it extracts all the complex data types and translates each complex data type into a JavaBean class.
- Generation of the service interface: The service interface is essentially a set of get*ComplexTypeName* (*InputType* formalParameter) methods. At run time, *ComplexTypeName* will be replaced by the actual complex data type name from the type file. The number of "get*" method are determined by the number of complex data types defined in the target web application. In our current implementation, as we do not support complex input data (a limitation of our

current implementation of the document collector component), the *InputType* is always instantiated into "String".

- Generation of a service implementation class. At run time, a wrapper behaves as follows:
    - Receives the input from its client.
    - Instantiates a document collector, forwards the user input to it, and uses it to collect the response of the target web application and cleans this response.
    - Uses one or more patterns to extract data from the cleaned document and packs the extracted data into a predefined complex object.
    - Returns the complex object to the client.

From the above steps we can see that the logic of the various wrappers is the same, with different data-extraction rules and complex data types used in each wrapper. Thus, ServiceBuilder provides a GenericServiceImpl class that encapsulates the generic data extraction logic as presented above. Then at the service (wrapper) implementation generation stage, the code generator generates a concrete class by subclassing the GenericServiceImpl class and customizes the data extraction rules and complex data types according to the information of the type file.

Figure 16 shows the class diagram of part of the *code generator* component and the relationship between the code-generator classes and the classes that it generates. In this diagram, the saved type information is encapsulated in the SimpleTypes and ComplexTypes classes. Both these two classes are collections of more fine-grained type classes. SimpleTypes contains a collection of SimpleType objects; each corresponding to a simple type defined using simple type editor. ComplexTypes contains a collection of ComplexType objects, each corresponding to a complex type defined using type editor. The RuleSet class is used to model a set of learned rules and the HTMLPage class is used to model a cleaned HTML document. In addition to other operations, the HTMLPage class provides a series of very important getValue operations. For example, the method getValue (Rule aRule, int pos) takes as input a Rule object and an integer, searches the

document to find a subsection that matches the Rule object, extracts the data value between positions pos and (pos+1) and returns that value.

At run time, first the SimpleTypes and ComplexTypes and RuleSet objects are initialized according to the information in the type and the learned pattern files. Then the code generator object translates each ComplexType object into a *BeanClass where the * represents the actual name of the complex type. In the same manner, the code-generator object generates a single *ServiceIF interface for the target web application. A series of get*BeanClass method are defined in the interface.

The last class generated by the CodeGenerator object is the *ServiceImpl class. This *ServiceImpl class extends the GenericServiceImpl class and all its methods follow the same template, customized where appropriate. For example, in the code example in the diagram, all the string "Instance" in the code body will be replaced by the actual complex type name.

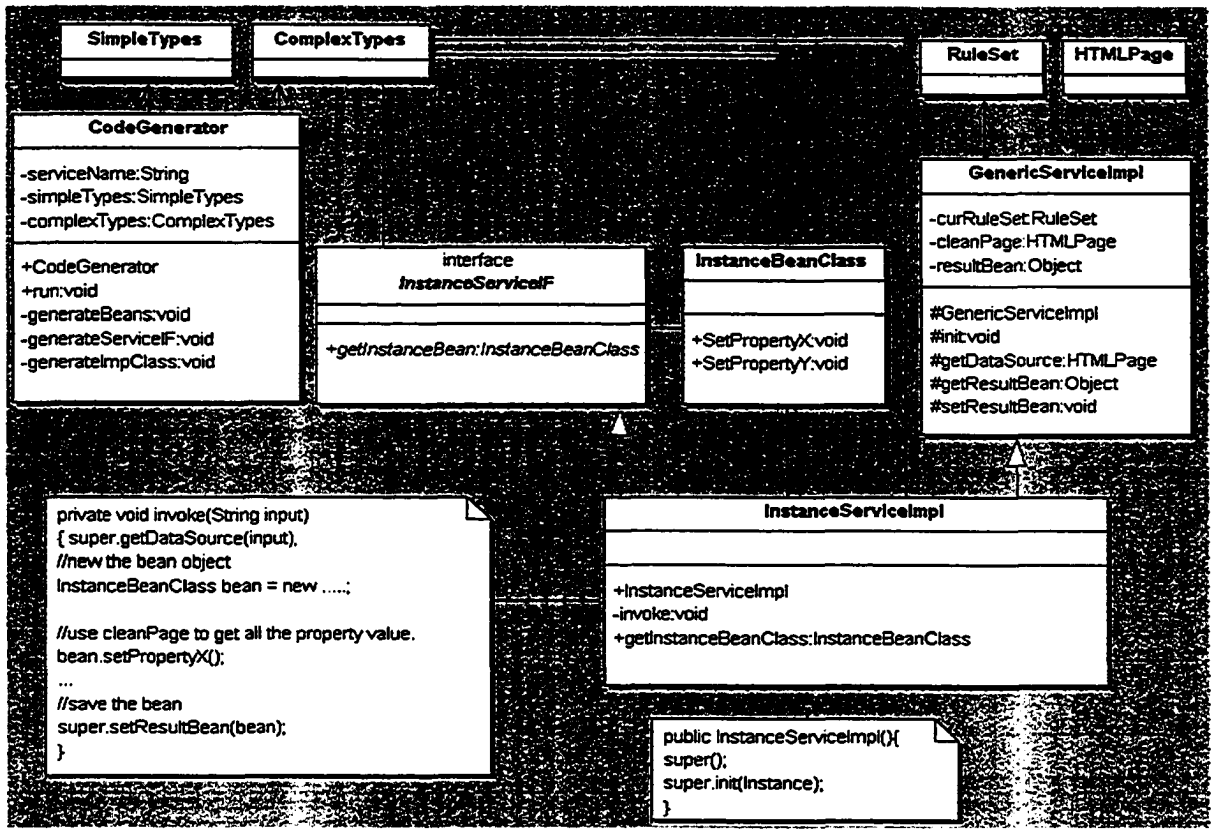

Figure 16: Relationship between Code Generator and the code it generates.

After the wrapper has been generated, we can use the tool provided by most web-service platform to generate a WSDL specification for the service and deploy the wrapper code as a web service to the web service platform. A WSDL specification for the stock-quote service wrapper we produced is shown in Figure 17.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:tns="urn:Foo" xmlns:xsd="http://www.w3.org/2001/XMLScheme"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" name="yahooStockService" targetNamespace="urn:Foo">
  <types>
    <schema xmlns="http://www.w3.org/2001/XMLSchema" xmlns:soap11-enc="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:xsi="http://www.w3.org/2001/XMLScheme-instance" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        targetNamespace="urn:Foo">
      <import namespace="http://schemas.xmlsoap.org/soap/encoding/"/>
      <complexType name="StockInfoBean">
        <sequence>
          <element name="lastTrade" type="string"/>
          <element name="tradeTime" type="string"/>
        </sequence>
      </complexType>
    </schema>
  </types>
  <message name="YahooStockQuoteServiceIF_getStockInfo">
    <part name="String_1" type="xsd:string"/></message>
  <message name="YahooStockQuoteServiceIF_getStockInfoResponse">
    <part name="result" type="tns:StockInfoBean"/></message>
  <portType name="YahooStockQuoteServiceIF">
    <operation name="getStockInfo" parameterOrder="String_1">
      <input message="tns:YahooStockQuoteServiceIF_getStockInfo"/>
      <output message="tns:YahooStockQuoteServiceIF_getStockInfoResponse"/></operation></portType>
  <binding name="YahooStockQuoteServiceIFBinding" type="tns:YahooStockQuoteServiceIF">
    <operation name="getStockInfo">
      <input>
        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" use="encoded" namespace="urn:Foo"/></input>
      <output>
        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" use="encoded" namespace="urn:Foo"/></output>
      <soap:operation soapAction=""/></operation>
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="rpc"/></binding>
  <service name="YahooStockService">
    <port name="YahooStockQuoteServiceIFPort" binding="tns:YahooStockQuoteServiceIFBinding">
      <soap:address xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        location="http://localhost:8080/yahoostockquote-jaxrpc/yahooStock"/>
    </port>
  </service>
</definitions>
```

Figure 17: WSDL file generated from the wrapper.

Chapter 4 Experimental Evaluation

In the previous chapters we described how ServiceBuilder could be used to generate a service wrapper for a web application. In this chapter, we focus on evaluating the tool in terms of the following aspects:

- Efficiency: How quickly does the tool get the job done?
- Effectiveness: Does it produce the correct result?
- Scope: What types of web applications does it deal with?

In this chapter, we are trying to answer those questions by a series of experiments and evaluations. Our experiments are mainly focused on the pattern miner component, because, human-factors aside, it is the most complex and most time-consuming component of the whole service-generation process and its quality attributes determine the corresponding quality attributes of the whole system. All the experiments presented in this chapter were performed in an IBM NetVista with 2.5MHZ Intel Pentium 4 processor and 1G memories.

There are two steps involved in the process of extraction rule learning: first, pattern mining with IPM, aim at identifying all the patterns appearing in the example documents that conform to the input criteria, and second, heuristic filtering of the generated patterns, aimed at eliminating the spurious patterns among them. Thus, the efficiency and effectiveness of the pattern-mining process depend on the efficiency and effectiveness of those two algorithms.

To explain the design of our experimental-evaluation procedure, let us introduce the concepts of fixed- and variant-attribute objects. If we refer to the set of data we want to extract from the web-application's response document as the "target object", then each individual piece of data that constitutes the object is an "attribute" of this object. For example, if we want to extract stock-quote information from the response document as shown in Figure 18, then the target object is the complex object inside the red box, which consists of attributes such as "last trade price", "previous close price", "open price",

"bid", "ask" etc. In general, two kinds of objects could appear on a response document of a web application i.e. fixed-attribute objects and variant-attribute objects.

The term "fixed-attributes object" refers to the kind of object that whenever it appears in a web document it contains a fixed set of attributes. The stock quote object in Figure 18 is a fixed-attributes object because no matter what the input tick is, the output object always consists of the same set of attributes as shown in Figure 18.

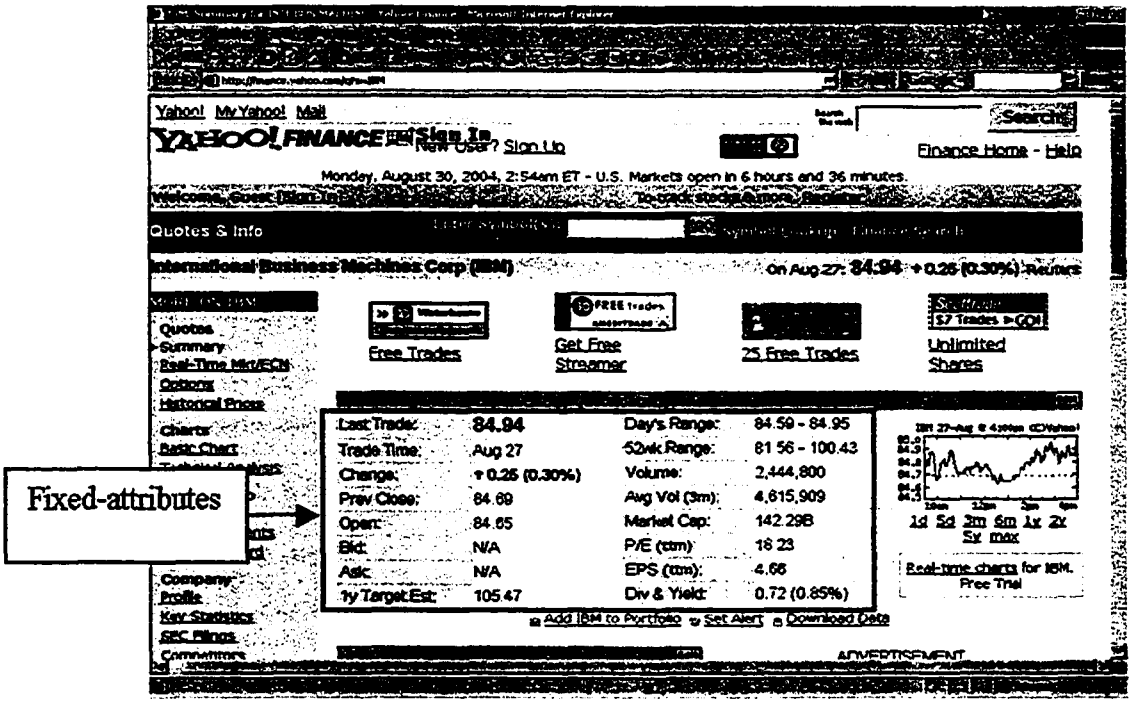

Figure 18: An example of fixed-attributes object

On the other hand, the term "variant-attributes object" refers to the kind of object that with each of its appearance, it may have a slightly different set of attributes. Figure 19 displays a document returned by the Amazon search engine with the input keyword "extreme programming". The returned document contains a list of books-related information. Each item of this list is a complex object we could call "book object". If we examine each "book object" carefully, we see that even though most "book object" contains the same set of attribute such as "list price", "our price", "you save" etc., variations do exist. For example, item 6 only contains "our price" attribute. Item 4

contains attributes such as "US list price", "CDN equivalent" that are not common in most other "book object".

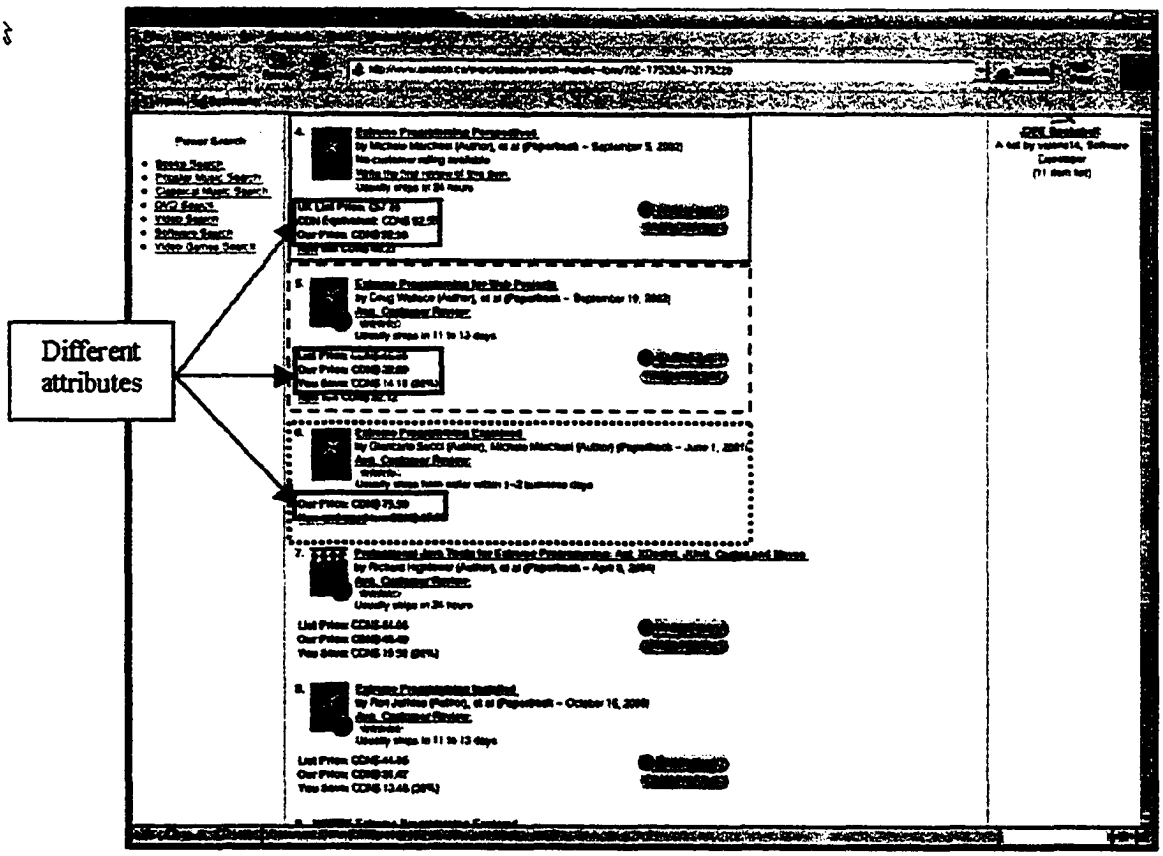

Figure 19: An example of variant-attributes object

In the extraction rule learning process, depending on which kind of output object is contained in the returned document, different pattern-filtering heuristics must be employed. Therefore, we used the fixed- vs. variant-attribute object distinction as the criterion differentiating the two experiments designed to test the ServiceBuilder system.

4.1 Extracting Services with Fixed-attributes Objects

In this experiment, we test the efficiency and effectiveness of IPM algorithm and the MRS heuristics that is only appropriate to web documents where the object of interest is of the fixed-attributes type and there is no more than one object of the desired type on each document.

This experiment was performed as follows:

- We selected the following five web applications that conform to the above mentioned criteria:
  - CBC weather (http://www.cbc.ca/weather/map.jsp);
  - CNN weather (http://weather.cnn.com/weather/forecast.jsp);
  - Yahoo quote (http://finance.yahoo.com/);
  - PC quote (http://www.pcquote.com/); and
  - Lycos quote (http://finance.lycos.com/qc/default.aspx).
- For each web application we collected 6 different sets of example documents (namely 5, 10, 20, 30, 40, and 50 documents) by feeding them with appropriate input data;
- With each collected example document set, we performed the mining process 6 times, each time with a different minimum and maximum pattern length range (30 to 39, 40 to 49, 50 to 59, 60 to 69, 70 to 79, and 80 to 89). The support rate of the pattern was always set to 100%. In other words, we only looked for those patterns that appeared on all the example documents.
- For each run of the mining process we recorded the following data:
  - Time consumed in running the IPM algorithm.
  - Number of patterns generated by the IPM algorithm.
  - Time consumed in applying Minimum Rule-Set (MRS) heuristics.
  - Number of patterns remaining after the application of the MRS heuristics.
  - Percentage of landmarks covered by the remaining patterns.

This experiment was designed with the following considerations in mind. The run time of IPM reflects its efficiency, and the study of its run time with respect to factors such as pattern and training sample number reflects how these factors affect its efficiency. The effectiveness of IPM can be evaluated based on the percentage of landmarks that are covered by the generate patterns[2]. The efficiency of the MRS heuristic is reflected by its

[2] Note: this is just an approximation, as it is possible that one document only contains a subset of all the landmarks.

run time, while its effectiveness can be determined by the ratio of patterns eliminated by the heuristics.

4.1.1 Efficiency of IPM

The original experiment data of IPM run time collected is shown in Table 1 of the Appendix. All the data in Table 1 are visualized as a chart in Figure 20. From this figure we can see that in our experiment configurations, in the worst case (PC Quote, 5 samples, length 80-89), it took IPM 57.497 seconds to generate all the patterns, which is realistically practical for a learning algorithm. In most cases, the run time of IPM is under 30 seconds and the average run time is 14.342 seconds.

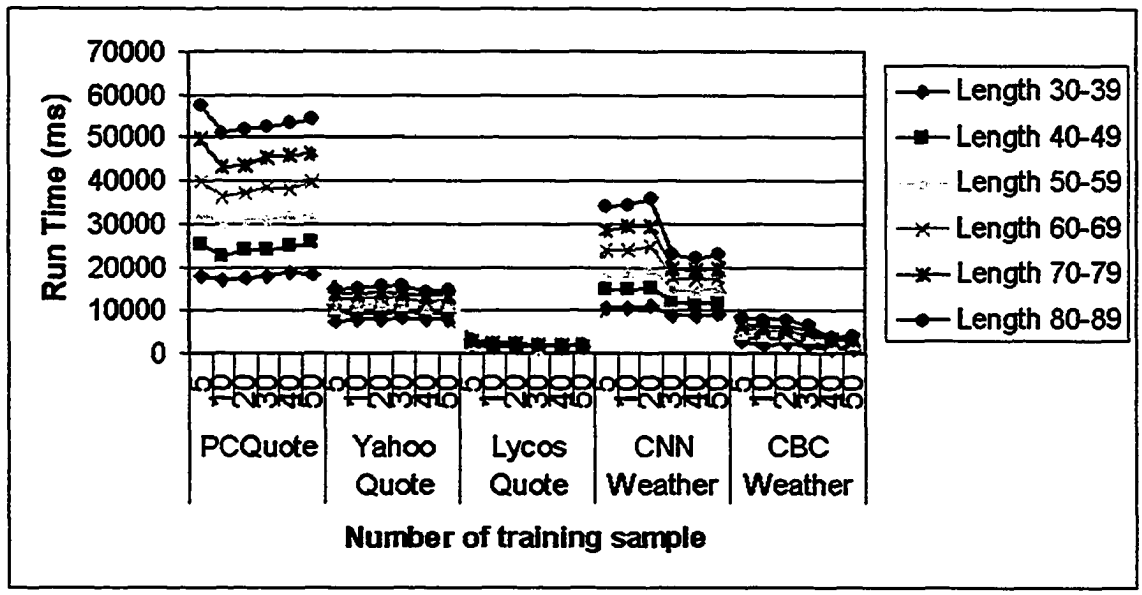

Figure 20: Experiment result of IPM run time

The figure also clearly demonstrates that the run time of IPM algorithm is increased with the increase of the desired pattern length, which is an inherent characteristic of IPM, as the generation of the longer patterns is based on the shorter patterns. There is a substantial variation in the run time of IPM among different web applications, with the highest average of 35.248 seconds (PC Quote) and the lowest average of 1.887 seconds (Lycos Quote). Overall, the average run time is 14.342 seconds. There are two factors that may affect the run time of IPM. The first one is the degree of similarity among the

training samples. In general, the higher the similarity, the longer IPM will run. The similarity among training examples can be partially reflected by the number of patterns actually mined by the algorithm. Another factor that may affect the run time is the average document length of the training example, the longer the average document length, the longer the IPM run time will be. The actual IPM run time is determined by the combined effect of those two factors. Figure 21 shows the number of patterns mined by IPM for the tested web applications (for detailed data see Table 2 in Appendix) and Figure 22 shows the average document length of the tested web applications. If we compare Figure 20 with Figure 21 and Figure 22, we can see that they roughly display the same figure pattern. That is consistent with our above analysis of the relationship between IPM run time, similarity among documents and document length.

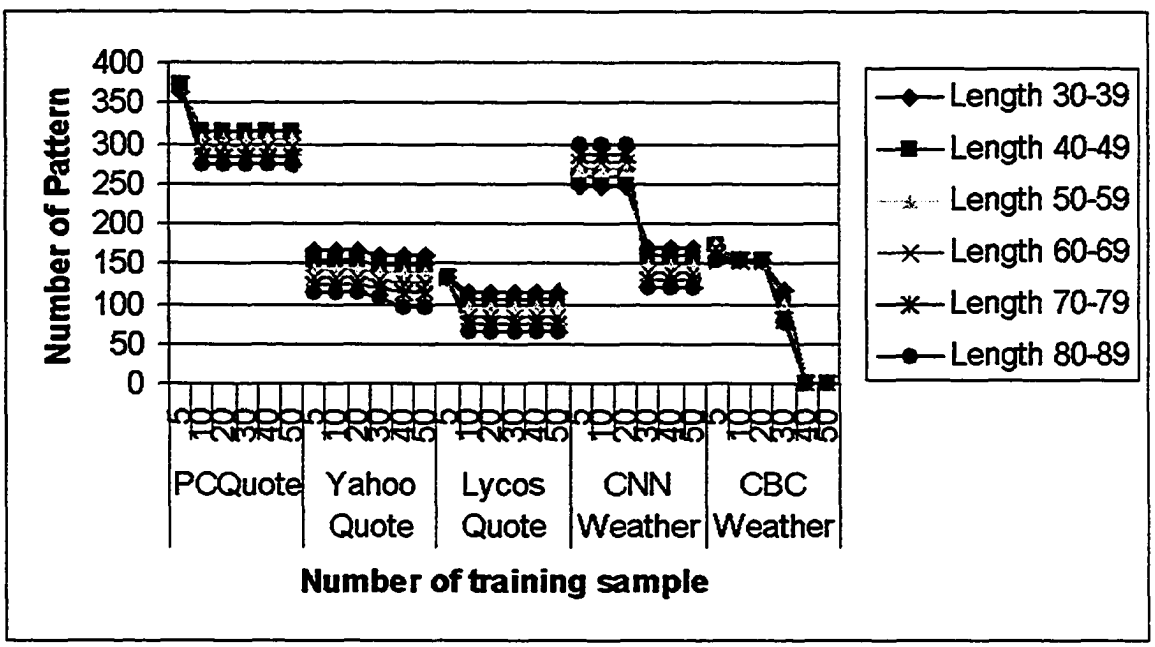

Figure 21: Number of patterns mined by IPM

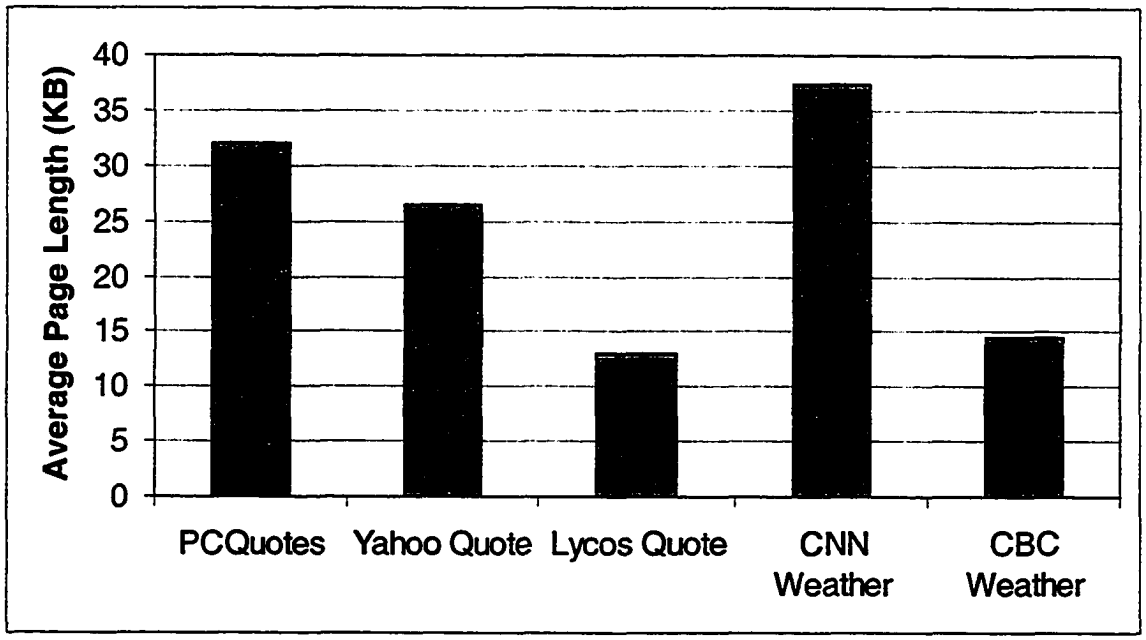

Figure 22: Average document length

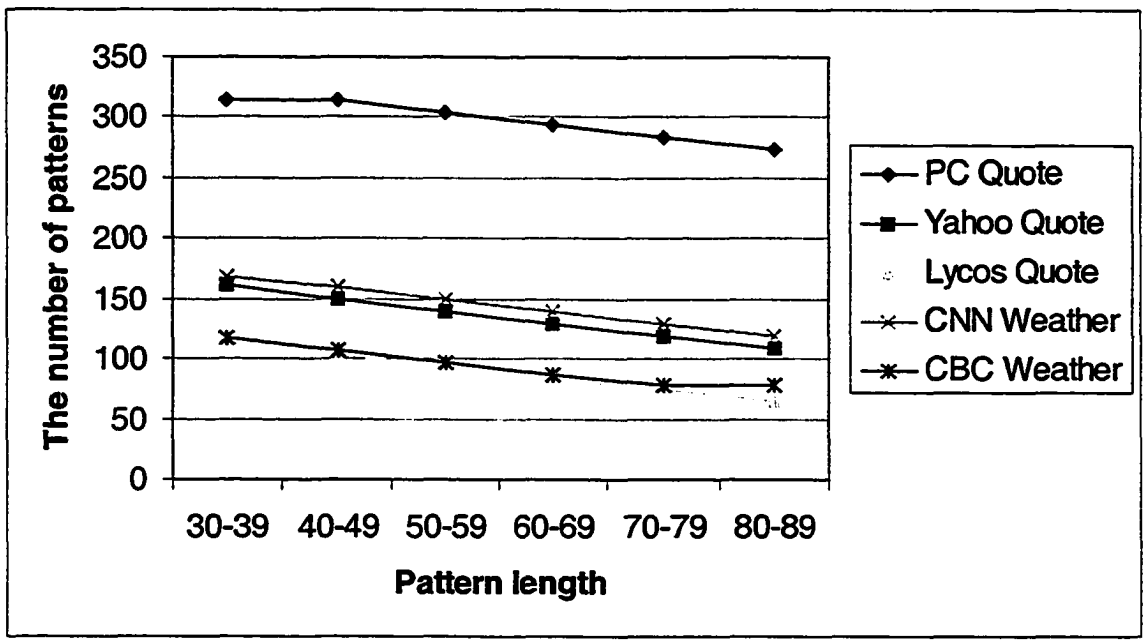

Figure 23: Number of pattern vs. pattern length

The relationship between the number of patterns mined and the desired pattern length is illustrated in Figure 23. For the simplicity of the figure, we only select one row of data (the row with 30 training examples) from each web application. All the other rows of data

follow the same trend. The figure illustrates that with the increase of the pattern length, the number of patterns found decreases. This kind of decrease can happen in two different situations:

- All the shorter patterns are contained within one or more longer patterns;
- Some shorter patterns are not included by any longer patterns.

If the first situation happens, then we did not lose anything. Because the less number of longer patterns covers at least the same amount of areas as those shorter patterns can cover. But if the second situation happens, then some of those areas covered by all the shorter patterns are not covered by larger patterns. This may affect our ability to get the pattern that can cover those data of interest to us. We will discuss this in more detail in the section when talking about the effectiveness of the algorithm.

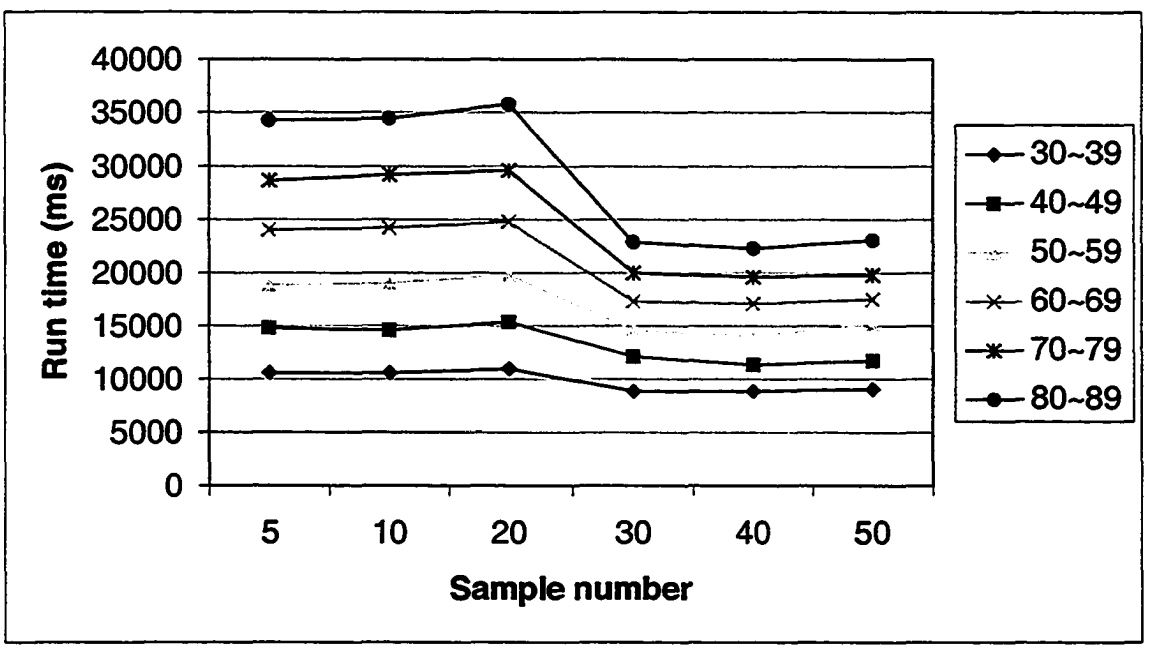

Figure 24: IPM run time vs. number of training examples

How does the number of sample documents affect the run time of IPM then? Intuitively, we might tend to believe that the larger the number of sample documents, the longer the time needed to mine the pattern. However, the experiment results show that if we keep

the same support rate, the increase of the number of training documents have little effect on the IPM run time.

Figure 24 shows the relationship between the IPM run time and the number of training examples in the experiment of CNN weather. The experiment results of the other web application hold the similar trend. We only show result of CNN weather for a clear view of the experiment result. From the graph we can see that with the increase of the training sample between 5 and 20, the corresponding run-time changes are barely noticeable. The same is true to the change between 30 and 50. The only noticeable change is a decrease between 20 and 30. By examining the corresponding training documents, we discovered that the decrease of the run time is due to the fact that the newly introduced training documents contained a noticeable structural change that did not appear in the previous training samples, which decreased the overall pattern numbers sharply.

A summary of the major factors that affect the run time of ServiceBuilder (dominated by IPM run time) is attached in Appendix as Table 13.

4.1.2 Effectiveness of IPM

The previous section discussed the question of how fast the IPM algorithm can generate the result patterns and what factors can affect the run-time efficiency of IPM. In this section, we examine what percentage of the data of interest is covered by the generated patterns. As we assume that the data of interest is in the proximity of landmarks, we can use the percentage of landmarks covered by the generated patterns as a measure of the percentage of data of interest being covered by the patterns. In this experiment, the landmark coverage rate is defined as the number of landmarks covered by the mined pattern set divided by the number of landmarks appearing on the training documents. In general, the larger the landmark coverage rate, the greater the chance any piece of data appearing on a document could be extracted by at least one of the rule.

Figure 25 shows the visualized experiments results of the landmark coverage rate. For more detailed experiment data, see Table 3 in Appendix. From the figure we can see three kinds of trends in the coverage rate:

- With the increase of the pattern length, the coverage rate decreases. (Yahoo Quote and CBC Weather).
- With the increase of the number of training examples the coverage rate drops. (PC Quote, CNN Weather and CBC Weather)
- No change at all. (Lycos Quotes)

The first trend reflects the fact that small HTML structures that are consistent among the training examples collectively contain more data of interest than larger invariant structures. In other words, there are structural changes within the big block that contains the data of interest. The second trend reflects the fact that some previous patterns are eliminated due to the new structural changes appearing in the newly introduced example documents. The last situation reflects the fact that all the shorter patterns are contained by larger patterns and hence they cover the same amount of landmarks.

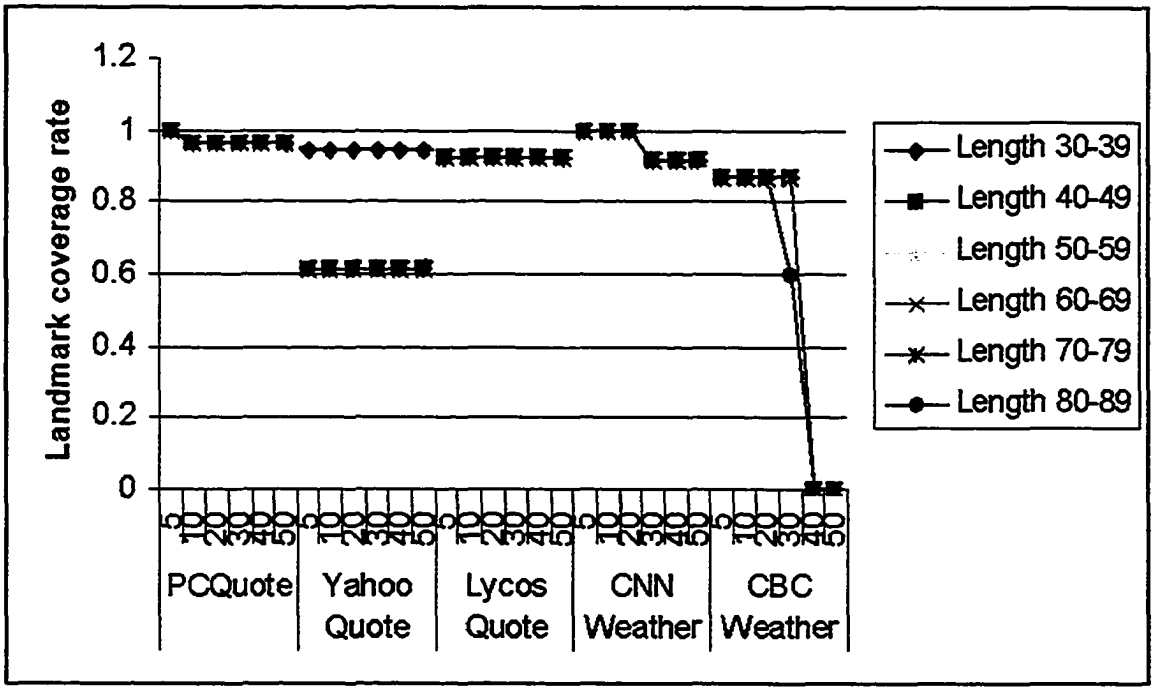

Figure 25: Landmark coverage rate

From this experiment, we can see that in most cases ServiceBuilder can find a pattern set with 100% support (those patterns appeared on each single example document) that contains most of the data of interest. But in order to construct an effective wrapper, we do

not need to be confined by those patterns. We could use other, less supported patterns as well. For example, Figure 26 shows four output documents of the CBC weather application. Each document displays the weather condition of a specific city (Edmonton, Beijing, St. Petersburg and Tokyo). Actually, the four documents are the representatives of four major types of output documents of the application (this is the types we discovered so far experimenting with around 100 different input city names). Edmonton represents the type of documents that contains the most detailed weather information. It contains three blocks of information, i.e. "current condition", "forecast" and "text forecast". Beijing represents the kind of document with slightly less detailed weather information, which only has the "current condition" and "forecast" information. St. Petersburg of Russia represents the kind of document with even less weather information, which only has "forecast" information. The last one Tokyo contains no information at all! This is actually the only city that the CBC weather forecast service that returns nothing. Though no actual weather information appears on the last document, it could be viewed as representing a kind of "exception" condition. It is exactly because we happened to include "Tokyo " as an example input and there is no data object on this document, that the IPM algorithm could not find any pattern that appears on all the example documents (see Figure 21).

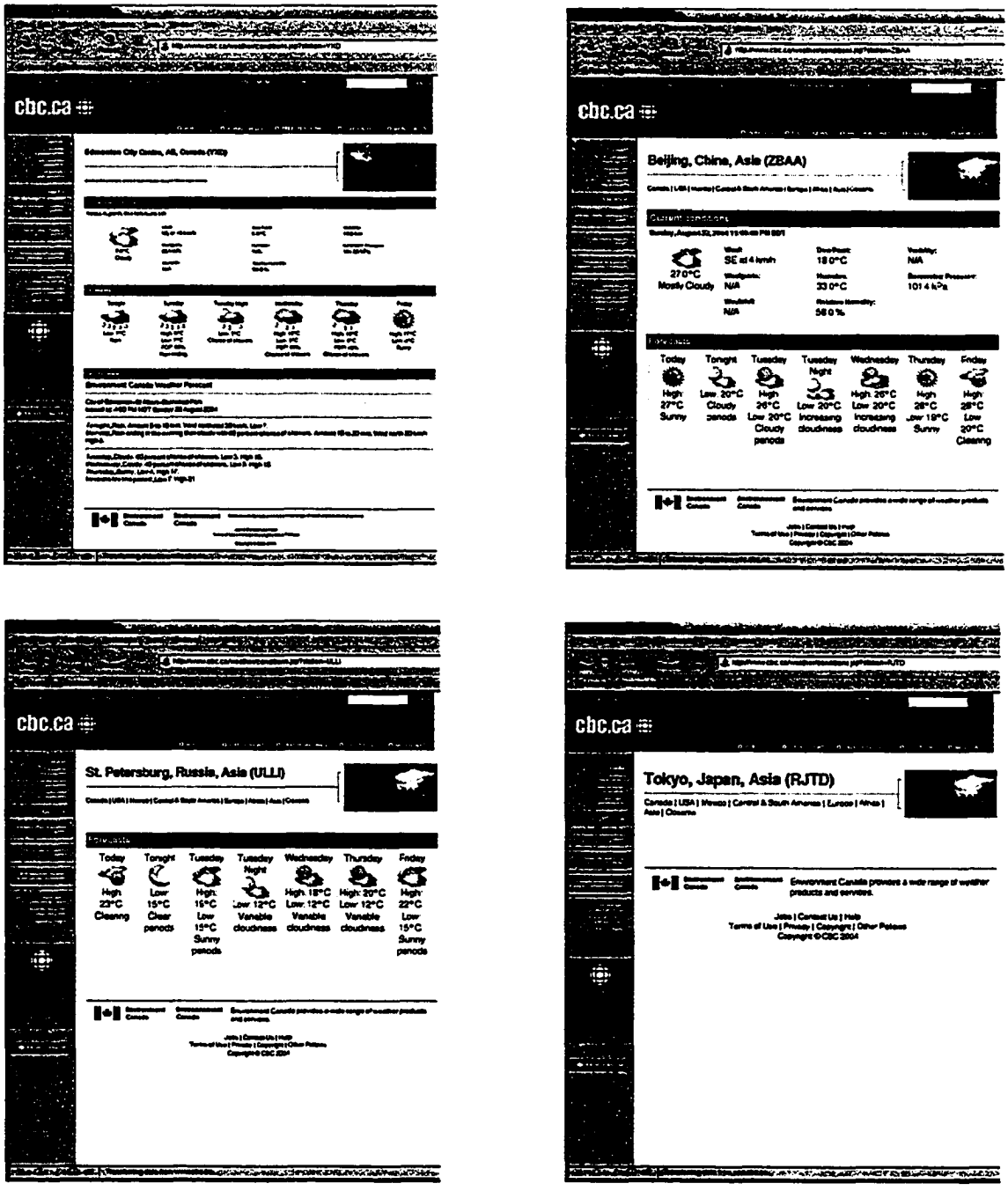

Figure 26: Four types of CBC weather output document

Therefore, in practice, we need not set the support rate of a pattern to be 100%. A 60% or 70% support rule makes sense, as the above example shows. One thing we have not dealt with yet is how to deal with "exceptions" like the case of Tokyo. Tokyo is different from the other three outputs. All the other documents are "valid" output documents that contain the expected data, but the Tokyo case roughly corresponds a "null" exception in

programming language. Even though it does not contain weather data, it conveys an exception probably useful to its user. The wrapper should be able to either throw an exception or return a "null" object to its client accordingly. The current implementation of ServiceBuilder does not include such a mechanism.

4.1.3 Efficiency and effectiveness of MRS heuristic

The run time of the MRS heuristic is shown in the Figure 27 (see Table 4 in the Appendix for the data). In the worst case (CBC Weather, 10 examples, pattern length 30-39), it only took 0.139 seconds to apply the MRS heuristic. In most cases, it took less than 0.02 seconds. Compared to the IPM algorithm (14.342 seconds on average), the MRS heuristics is very fast.

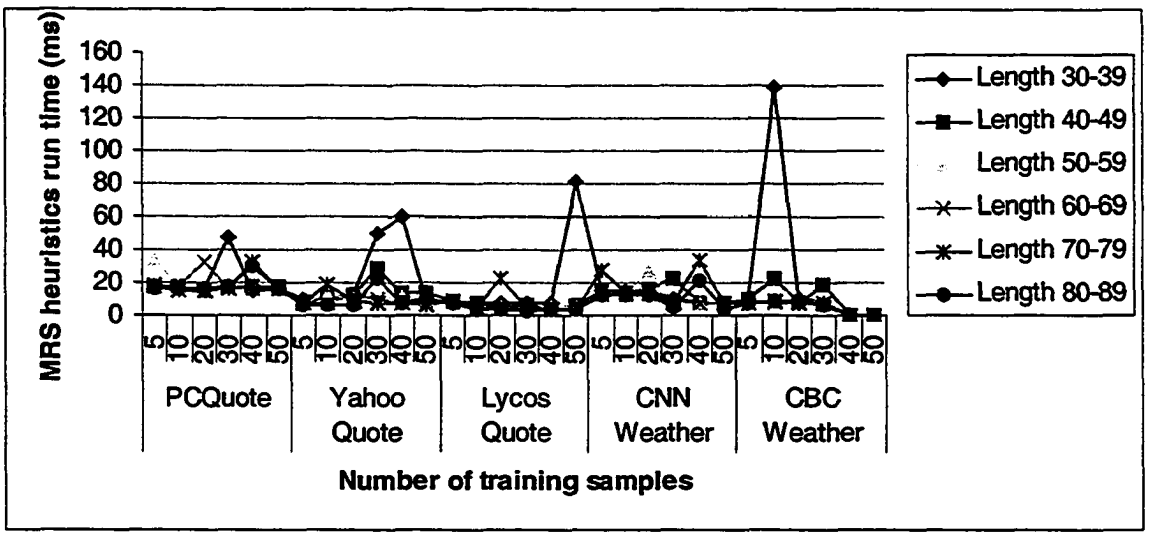

Figure 27: Experiment results of MRS heuristics efficiency

Recall that the main goal of applying the heuristic is to eliminate those redundant patterns and return a relatively small yet sufficient set of patterns that cover the same set of data of interest as all the patterns generated by IPM algorithm. Therefore, the effectiveness of the heuristic can be measured by the number of patterns remaining after running the heuristic or what percentage of the patterns has been eliminated by the heuristic. The number of patterns left after heuristics is show in Figure 28 (see Table 5 in Appendix for the complete data).

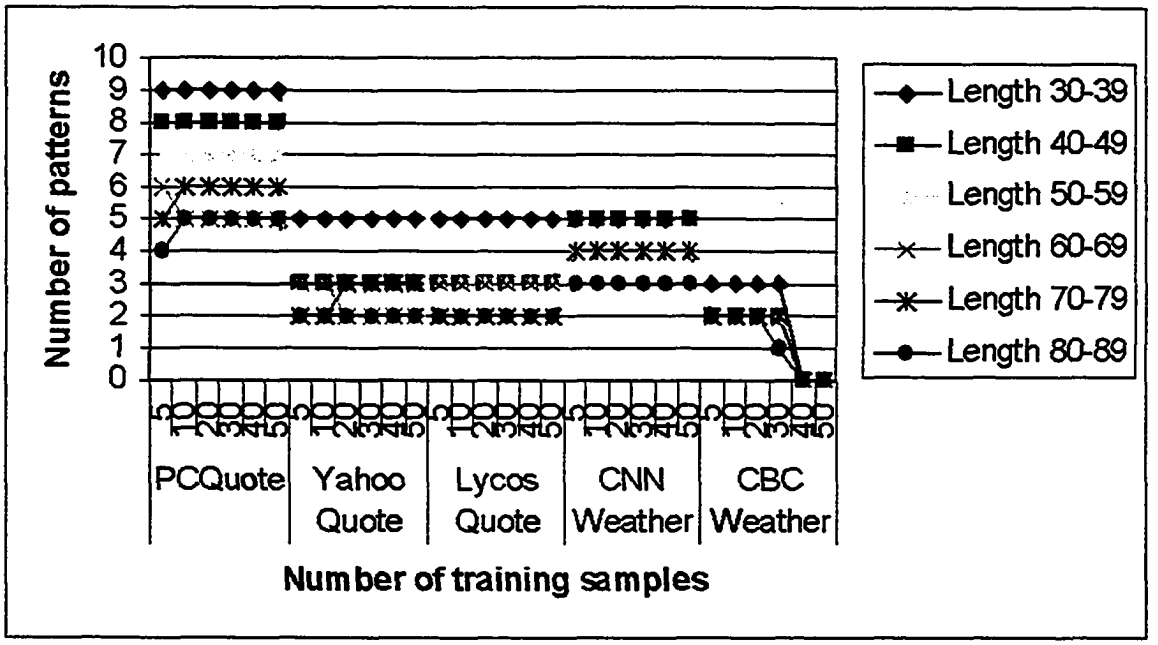

Figure 28: Experiment results of MRS effectiveness

We can see that in all cases less than 10 patterns are left after the MRS heuristic has been applied and in most cases the pattern number is no more than 5, which is a very small number and would not bring too much burden to the user to select from. If we compare this figure with Figure 21, we can see that in most cases 97% of the patterns generated by IPM are eliminated. With this high elimination ratio, we can safely say that MRS is very effective in eliminating redundant patterns.

4.2 Extracting Services with Variant-attributes Objects

In this experiment, we examine the efficiency and effectiveness of our system when dealing with web applications whose output documents contain objects with a variant attribute set and there may be multiple objects of the same type appearing on the same document. An example of this kind of document is shown in Figure 19.

The experiment design is similar to the previous one.

- We selected two web applications:

- Amazon book buying search engine: (http://www.amazon.ca/exec/obidos/tg/browse/-/915398/701-3668674-9981157)
- Chapter online book buying search engine: (http://www.chapters.indigo.ca/default.asp?gog=1)

- For each web application, we used the same set of input data with 10 keywords and collected the returned documents.
- We run the mining algorithm with different minimum support number and desired pattern length and recorded the following data:
  - Time consumed by IPM
  - Number of patterns generated by IPM
  - Percentage of objects covered by those patterns
  - Time consumed by the MCS (Maximum Common Sub-pattern) heuristic
  - Number of patterns' after groups the application of the heuristic

In this experiment, the goal was similar to the previous experiment, i.e. to test the efficiency and effectiveness of our system when dealing with these more complex web documents, and thus implicitly also assess the scope of the overall applicability of the ServiceBuilder process.

4.2.1 Effectiveness and Efficiency of IPM

Figure 29 illustrates the run time of IPM in this experiment (see Table 6 in the Appendix for the complete data). We can see that in the worse case, it only takes less than 0.9 seconds for IPM to generate all the patterns. The average run time is 0.351 seconds. Compared to the IPM run time in the first experiment (Figure 20), IPM runs much faster with these applications. This can be explained by the fact that the degree of similarity among the response documents of these applications is much less than that of the previous experiment.

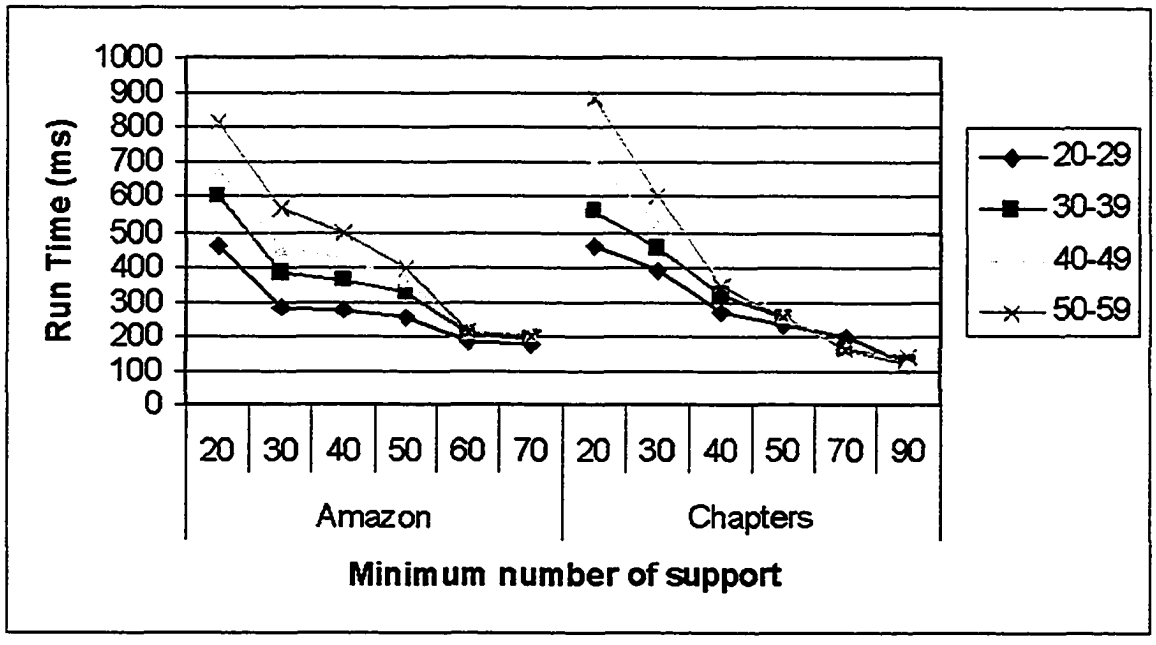

Figure 29: Efficiency of IPM

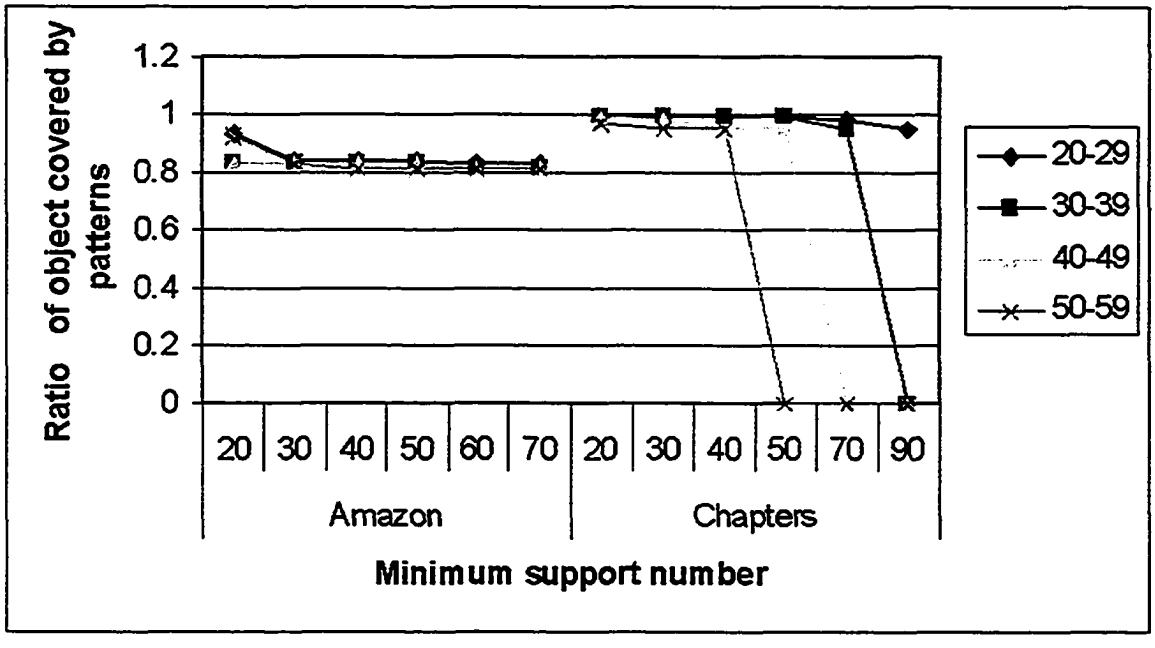

Figure 30: Effectiveness of IPM

To evaluate the effectiveness of IPM in this experiment, we recorded the number of objects in the returned documents that are covered by the patterns and calculated the ratio between the number of covered objects and all the objects on the returned documents.

Figure 30 shows the visualized results (for original data, see Table 7 in Appendix). From the figure we can see that in most case, the patterns can cover more than 80% of all the objects returned by the web application. Only in three cases the coverage rate is less than 80% (0% actually). When we look into those three cases, we found that, that is due to the fact that there is simply no patterns meet the length requirement and the minimum occurrence requirement at the same time. Those three cases actually reflect a shortcoming of ServiceBuilder in this stage: the result is affected by the input parameters and right now the selection of input parameters are totally up to the end user, we do not provide any support to help the user select those input parameters.

4.2.2 Efficiency and effectiveness of the MCS heuristic

The execution time of the MCS heuristic is illustrated in Figure 31 (for original data see Table 8 in Appendix). The average run time is 0.049 seconds. The worst case is 0.184 seconds. Compare to the time consumed by IPM in this experiment, the execution time of the MCS heuristic is almost negligible.

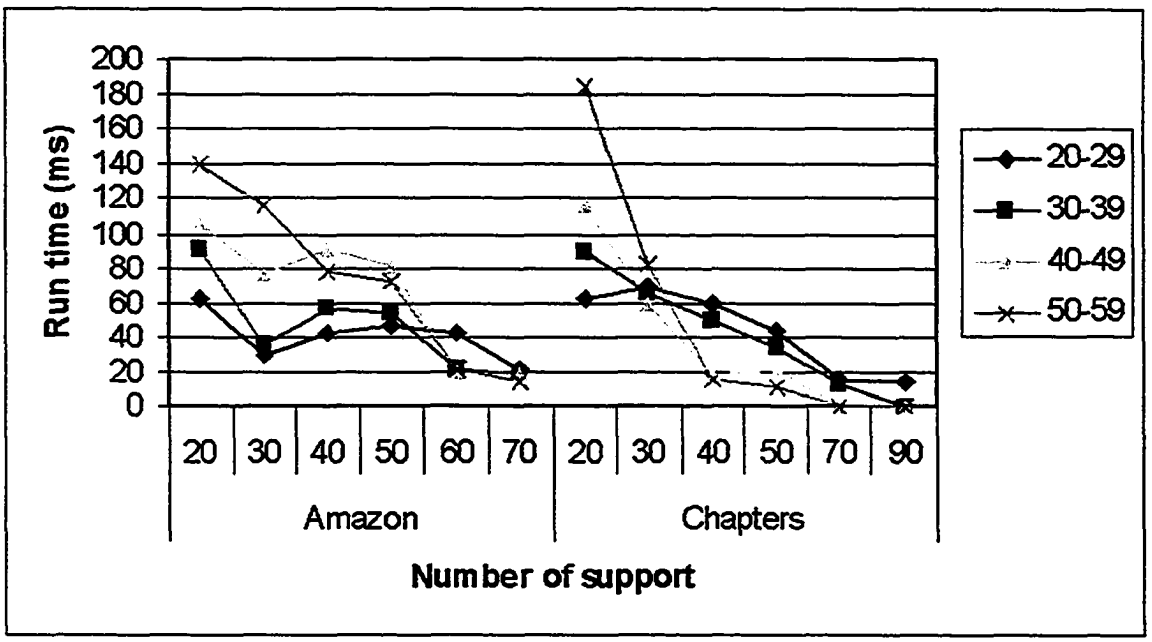

Figure 31: Run time of MCS

In order to evaluate the effectiveness of the MCS heuristic, we could compare the total number of patterns mined by IPM and the number of pattern remained after the MCS

heuristic. Figure 32 and Figure 33 illustrate pattern mined by IPM before MCS heuristics and pattern number after MCS heuristics under different experiment configurations (for original experiment data, see Table 9 and Table 10 in Appendix respectively).

From Figure 32 we can see that, in most cases the pattern number before heuristics is less than 80. In the worst case the pattern number is 106. The average pattern number before heuristics is 36.54.

Figure 33 shows that in most cases there are less than 8 patterns left after the MCS heuristic has been applied and in the worst case the number is 13. The average number of pattern left after the heuristic in this experiment is 4.06 that are only about 11% of the number of pattern before heuristics. To put it another way, without MCS heuristics, on average, a user needs to view around 36 different patterns as compared to view around 4 patterns if the MCS heuristics is applied. That is a nearly 90% reduction of the user's workload.

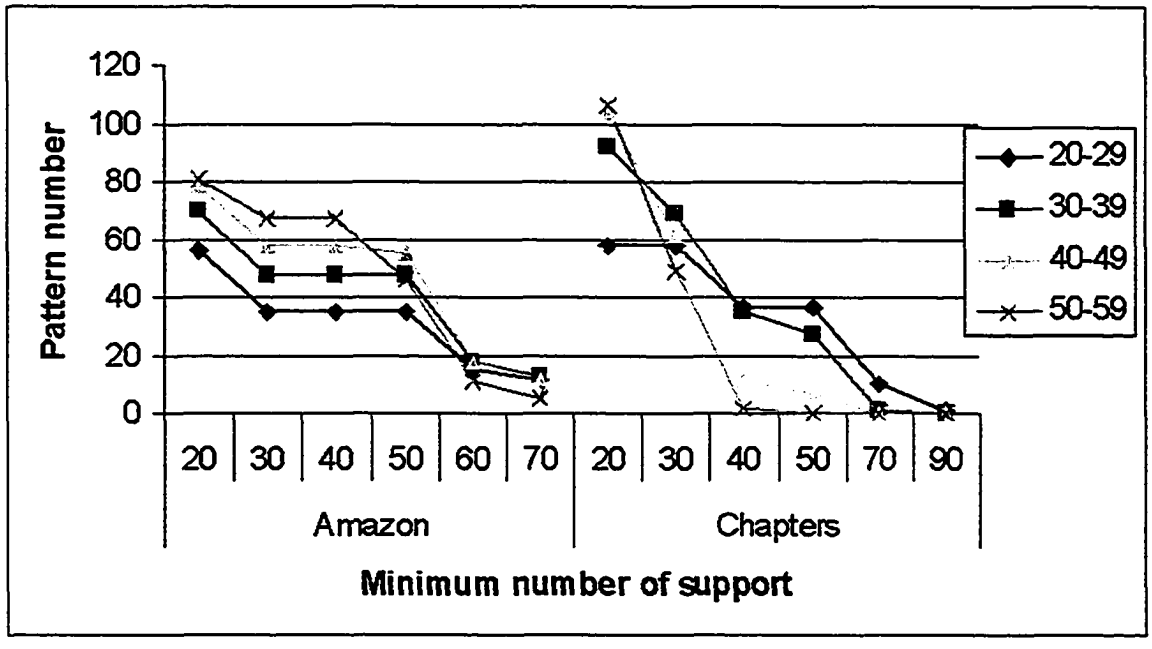

Figure 32: Number of pattern before The MCS heuristic

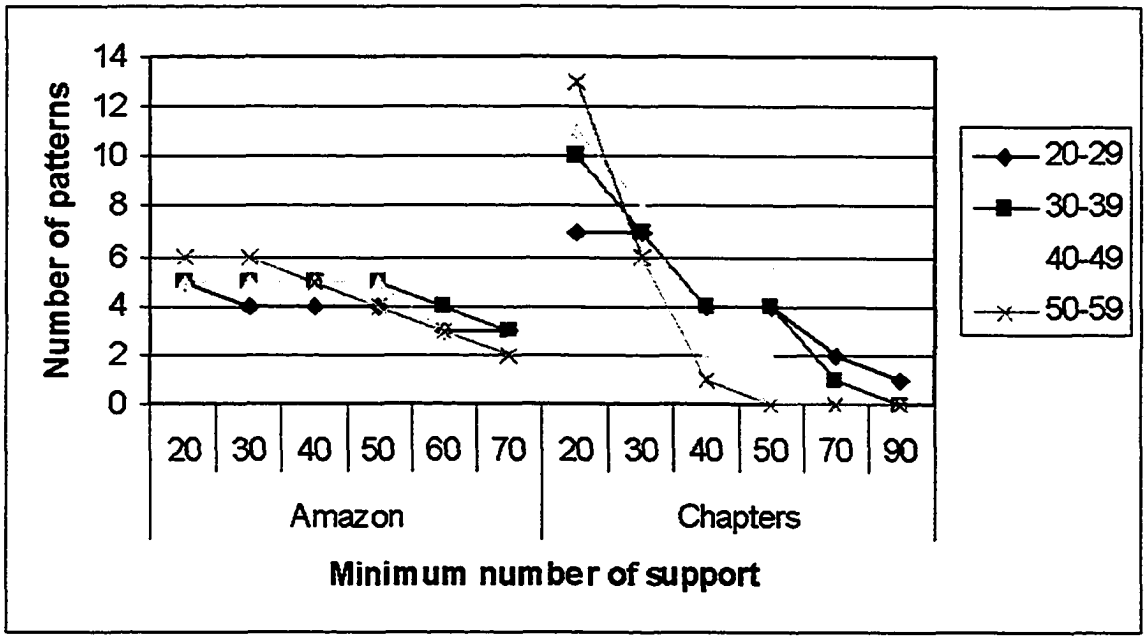

Figure 33: Number of patterns after MCS heuristic

4.3 Evaluation and discussion

In this section, we hope to provide an overall assessment of the ServiceBuilder system. We identify the distinct features of ServiceBuilder and its limitations, and we outline possible solutions to address these limitations in future research.

4.3.1 Distinct ServiceBuilder Features

ServiceBuilder does not require manually labeled training examples as most other wrapper-induction tools do. Training example labeling is a labor intensive and error prone process especially for complex and lengthy web documents. Even with tool support, labeling a large amount of training example is just too expensive to be practical. ServiceBuilder can collect any number of training examples specified by the user. To collect these training examples, it only requires that the user provide input data for each

training example, much like in a web-application testing scenario[3]. Compared to labeling a web document, preparing input data for a web application is a much simpler task.

ServiceBuilder can learn from a large number of training examples. Learning from a large amount of examples is extremely important for wrapping web applications, as the web documents structure is controlled by the internal logic of the web application. Depending on certain properties of the input of the web application, the output document may reveal different layout. This is clearly demonstrated in the CBC weather application we discussed previously in Figure 8. If only a small number of examples (say 10 documents) are examined, it is likely that not all those four types of output documents will be included. The more example documents are available the more likely it is that got all the different types of output documents will be included and thus the discovered patterns will represent all types of desired data. The experimental results in section 4.3.1 already show that run time of IPM is not affected by the number of training examples. This means that ServiceBuilder can learn from a larger number of examples without sacrificing its efficiency.

The wrapper produced by ServiceBuilder tends to be more robust to source document changes than wrappers generated by other approaches. That can be explained by the fact that ServiceBuilder is not trying to find a pattern that "literally" appeared on the source document. But instead, it mines the pattern on an already "filtered" sequence of tokens. The translator component works as a filter, and the "interesting delimiters" and "landmarks" collectively determine what tokens can sift through. Just like a filter in an electronic circuit can filter away certain degree of electronic noise and guarantee the smooth performance of the output side, the translator at run time can filter out certain new introduced source changes and protect the wrapper from corrupting. Of course this kind of protection does not always work, but it is still more robust than those wrappers that based on literal patterns that lack of this filtering mechanism.

---

[3] In fact the input data may already be available in a test-case suite employed to exercise the web application.

4.3.2 Limitations

One important limitation of ServiceBuilder is that the generated wrapper code does not completely simulate the behaviors of an actual browser that could carry and receive cookies that used to maintain a state in the server side. As a result, the current ServiceBuilder can only generate wrappers for web applications that implement only one-step interactions with the user. It cannot generate wrappers for web applications that implement workflow interactions with the user. For example, an online bookstore application usually involves the following steps: "Search the desired book", "Add the book to the shopping cart", "Proceed to checkout", "sign to secure server"...etc. During the process from one step to another, a token called cookie is transfer from the client to the server as an identification of the client enabling the server to maintain an internal state.

With an addition of a component that can simulate the browser's capability of receiving and managing cookies and a platform that supports web-services composition, the ServiceBuilder could support the generation of wrappers for this kind of workflow-based web applications. The idea is that ServiceBuilder would first be used to wrap each step of the interaction as a web-service wrapper, and then these elementary services could be composed into a larger service wrapper corresponding to the whole web application. Of course, that would require the client to supply all the data necessary during the web-application interaction all at once.

The second limitation of the current ServiceBuilder is that it lacks a "crawler" capability. Some web applications, like book-buying search engines actually return a large amount of data objects that span multiple web documents. When a real person interacts with the web application, the user will click a certain button to request the next result document that contains data until all the result documents are viewed. To simulate this behavior, a crawler capability would be required in the generated code that can continually issue requests to the server until all the result documents are received. This capability would require support for state maintenance ability as above.

A third limitation of ServiceBuilder is the determination of its input parameters. To learn the patterns, three major input parameters need to be specified by the user, i.e., the minimum support, minimum pattern length, and maximum pattern length.

The minimum support specifies the minimum number of occurrence a pattern needs to appear in the training examples. Any pattern with fewer occurrences will be eliminated. There is no rule of thumb for determining how large this support should be. It all depends on each individual web application. For web applications "expected" to output the same "type" of web documents, this support could be set to a high value compared to the estimated number of objects appearing in the training examples. However, if a trial run reveals that there are different "types" of returned documents, like in the CBC weather example, this parameter should be lowered to try to mine the least supported pattern.

The minimum pattern length and maximum pattern length - as their names suggest - determine the target pattern length range. Ideally, this range should be long enough to cover the biggest object appearing in the returned response document but not too long to sacrifice the efficiency of the mining algorithm. As we demonstrated in the previous section, the longer the target length of the pattern the slower the mining algorithm will be. For web documents where there is no duplicate attributes in the same document, the pattern length is not a very important parameter: if the pattern is too short to cover the whole object, one can always use several smaller patterns to extract data and compose them into the larger actual object. For example, in Figure 34, suppose the whole stock information object is the desired object; then four patterns -- p1, p2, p3 and p4 – can be used to extract the data from the document and compose those data in the final target object. However, for web documents that contain multiple objects and the objects have a variant-attributes set, the selection of those two parameters becomes crucial. If the mined pattern is not long enough to cover the whole target object, you cannot use smaller patterns to compose larger object, as there is no fixed blueprint (long enough pattern) for you to follow in the process of composing. For instance, in Figure 35, without longer patterns like p1 and p2 that can distinguish each book object from one another, shorter patterns like p3 does not really help. As we do not know where the "object boundary" is and which object an instance of the smaller pattern belongs to.

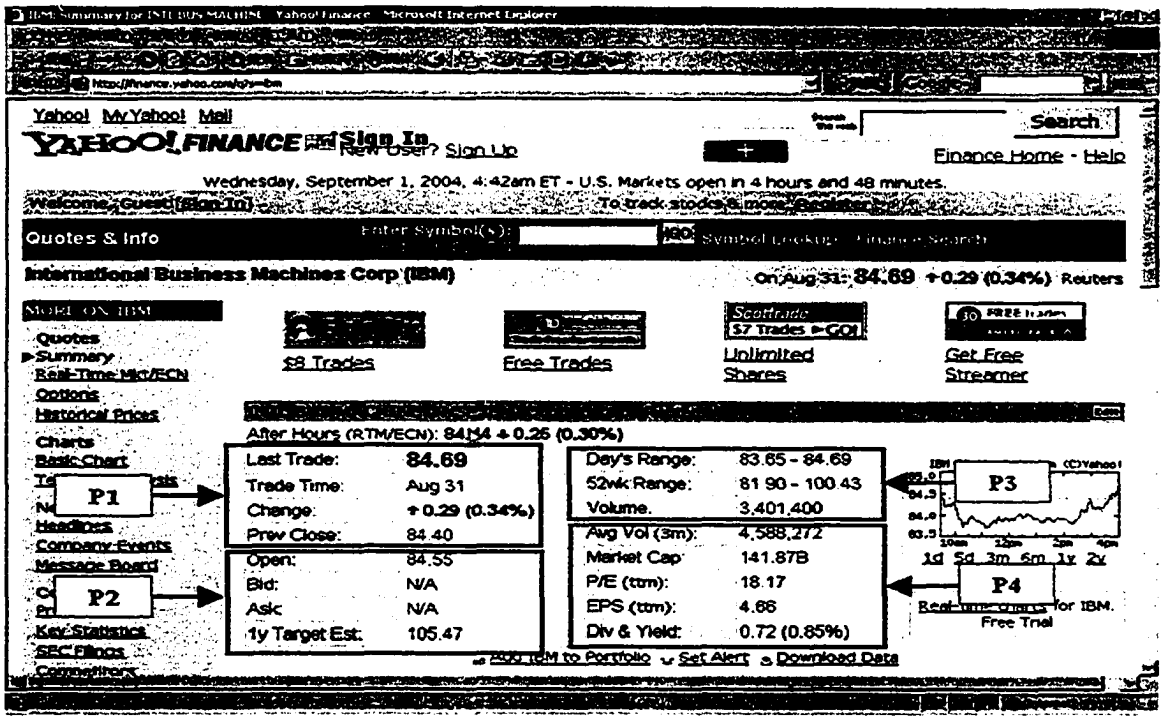

Figure 34: Situation where both small and large pattern can be used

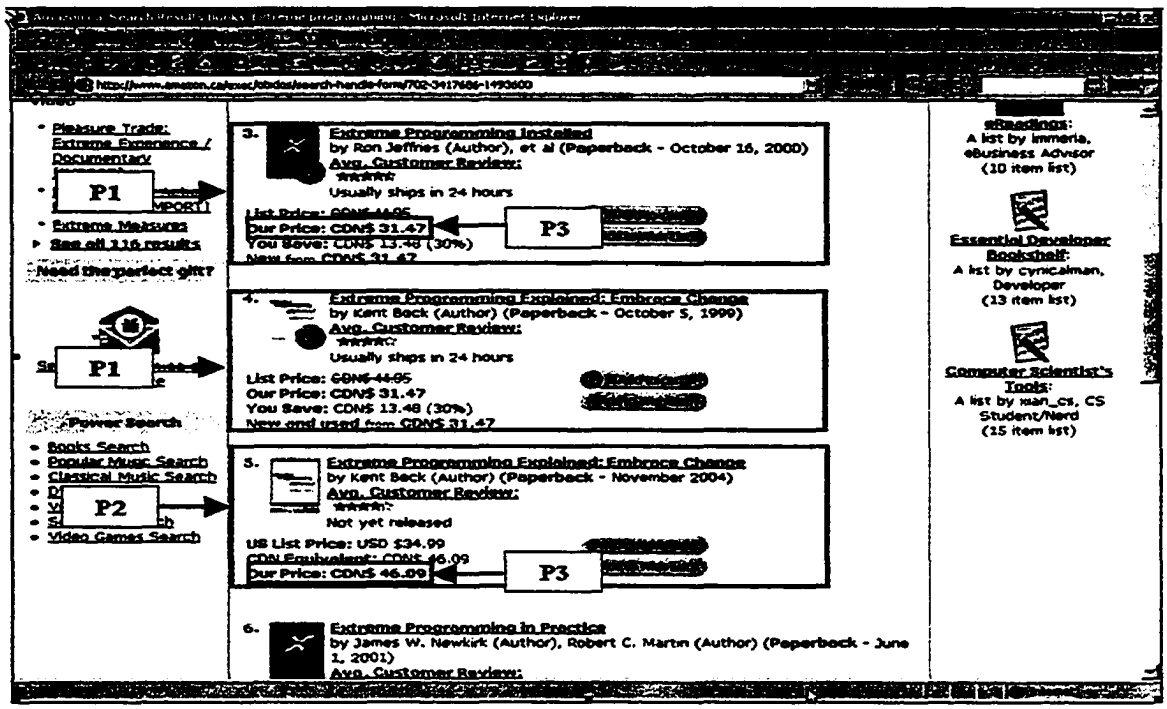

Figure 35: Situation where only large pattern is useful

Chapter 5 Contributions and Future Work

5.1 Research contributions

In this thesis, we presented ServiceBuilder, a new wrapper-construction tool that can semi-automatically reengineer a web application into a web service provider. The main contributions of this thesis are summarized in the following.

- ServiceBuilder represents and implements a new approach to migrating traditional web applications into web services providers. Traditional reengineering approaches usually assume access to code base of the target web application while our approach does not have this requirement (discussed in section 1.2).
- We introduced sequential-pattern mining techniques to solve the problem of extraction-rule learning in wrapper construction in general. This technique has several advantages over traditional wrapper induction and tree-structure based wrapper construction techniques (discussed in section 4.3.1).
    - First, it does not require the manual labeling of training examples; the data of interest is assumed to be in the vicinity of domain-specific landmark words. In general, it is simpler and less error prone to identify the domain-specific words that are used as labels for the data of interest on the web-application responses than to label the actual data itself.
    - Second, the produced wrapper is more robust to source changes than wrappers generated by other approaches; the robustness comes from the fact that the mining algorithm does not working on the original source document. Instead, it works on a sifted token sequence. So any source changes that cannot pass through the sift process (translation) will not affect the wrapper.
    - Third, this approach can efficiently learn from a large amount of training examples and consequently provides more confidence for the resulting data-extraction rules. The larger the number of training examples the more likely it becomes that several variations of the source documents will be

examined during learning. As a result, the data-extraction rules learned from a larger set of example tend to be more general and robust than those from a smaller set of examples.

- ServiceBuilder is a highly automated, easy to use, wrapper construction toolkit. With proper setup information, similar in nature to the information required for test the web application, the tool automatically collects training examples from the target web application and learns a small set of candidate extraction patterns. The tool then visually presents these patterns to the user in the context of example web documents and lets the user select the pattern or patterns containing the data of interest and defines the output data format with the help of a wizard. Finally, the tool generates Java implementations that essentially wrap the target web application and presents it as a web service provider (discussed in Chapter 3).
- ServiceBuilder offers a set of heuristics that can efficiently and effectively eliminate most spurious patterns and greatly alleviate the user's burden of wading through a large amount of candidate patterns to select the one that containing data of interest (discussed in section 3.2.1.3, section 4.1.3 and section 4.2.2).
- The generated wrapper is a Java implementation readily deployable as a web service and therefore easily accessible on a variety of platforms. As a web service, the wrapper is accessible to a broader range of clients.

5.2 Future work

There are several possible extensions on ServiceBuilder that could be explored in the future:

- This work provides a foundation for further service composition extensions. Now ServiceBuilder can only migrate single-step, search-engine-like services into web services. With a state management component and a platform that support the future web service composition standards, it could be extended to migrate multi-step, complex web applications into web services.
- One possible way to enable the generated web-service wrapper with the ability of state maintenance would be to add a cookie and state management framework into

the code generator component. At run time, the code generator will generate a customized cookie and state management component for each individual wrapper.

- Crawler functionality should also be added to the generated web service wrapper to enable multi-step interaction migration. This extension would require the support of state maintenance extension we discussed above. Furthermore, the system would require a way to know the widgets on the return page that leads to the subsequent documents containing the rest of the result. This information could either provided by the user or through testing the web application with a learning algorithm.
- Even though, wrappers generated by ServiceBuilder are more robust to source changes than wrappers generated by other techniques, they are not immune to corruption. To be practical in a realistic world, the ability of automatic detection of wrapper failure and automatic wrapper repair is desired. Some research works [LMK03][Kus00b] have already been done on those topics. Incorporating and extending these techniques into ServiceBuilder, would result in a more practical and more useful wrapper-construction tool.
- Finally, intelligent techniques could be developed to automatically learn the object boundary. If this could be done in an efficient manner, then the input parameter selection is not a critical problem, as discussed in section 4.3.2. In this case, small patterns could be employed to extract the data, which could then be composed into a complex object without mixing the data members of different objects together.

Bibliography

[AM98] Gustavo O. Arocena, Alberto O. Mendelzon: WebOQL: Restructuring Documents, Databases and Webs: in Proceedings of the 14th IEEE International Conference on Data Engineering, pp. 24-33, Orlando, Florida, 1998.

[AS94] R. Agrawal and R. Srikant: Fast algorithms for mining association rules: In Proc. 1994 Int. Conf. Very Large Data Bases (VLDB'94), pp. 487-499, Santiago, Chile, Sept. 1994.

[BFG01] R. Baumgartner, S. Flesca and Georg Gottlob: Visual Web Information Extraction with Lixto: Proceedings of the 27th International Conference on Very Large Data Bases, pp. 119—128, 2001.

[CM98] Valter Crescenzi, Giansalvatore Mecca: Grammars Have Exceptions: Information Systems. Vol. 23, No. 8, pp. 539-565, 1998.

[CMM01] V. Crescenzi, G. Mecca and P. Merialdo: RoadRunner: Towards Automatic Data Extraction from Large Web Sites: in Proceedings of 27th International Conference on Very Large Data Bases, pp. 109—118, Rome, Italy, 2001.

[EMR02] 2002 E-commerce Multi-sector Report, available at: http://www.census.gov/eos/www/papers/2002/2002finaltext.pdf; Last access to site: July 2004.

[ERSS02] M. El-Ramly, E. Stroulia, P. Sorenson: Interaction-Pattern Mining: Extracting Usage Scenarios from Run-time Behavior Traces, The Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, July 23 - 26, 2002, Edmonton, Alberta, Canada.

[GSB+01] S. Graham, S. Simeonov, T. Boubez, T. Boubez, D. Davis, G. Daniels, Y. Nakamura, and R. Neyama: Building Web Services with Java: Making Sense of XML, SOAP, WSDL and UDDI: Sams, 2001.

[HD98a] C.-N. Hsu, and M.-T. Dung: Generating Finite-State Transducers for Semi-Structured Data Extraction from the Web. Information Systems. Vol. 23, No. 8, pp. 521-538, 1998.

[IBM-FAQ] IBM FAQ, What is a component model, available at: http://www.developer.ibm.com/tech/faq/individual?oid=2:22733; Last access to site: July 2004.

[Joh73] D. S. Johnson: Approximation algorithms for combinatorial problems: Proceedings of the fifth annual ACM symposium on Theory of computing, pp. 38—49, Austin, United States, 1973.

[JS04] Yingtao Jiang, Eleni Stroulia: Towards Reengineering Web Sites to Web-services Providers: Proceedings of the 8th European Conference on Software Maintenance and Reengineering, pp. 296-308, Tampere, Finland, 2004.

[KMA+98] C. Knoblock, S. Minton, J. Ambite, N. Ashish, J. Margulis, J. Modi, I. Muslea, A. Philpot, and S. Tejada: Modeling web sources for information integration. : In Proc. 15th Natl. Conf. Artif. Intell. (AAAI-98), 1998, pp. 211-218.

[KT03] S. Kuhlins and R. Tredwell: Toolkits for generating wrappers: In NODe02, Vol. 2591 of LNCS, pp. 184-198, 2003.

[Kus97] N. Kushmerick: Wrapper induction for information extraction: Ph.D Thesis, University of Washington, Seattle, WA, 1997.

[Kus00a] N. Kushmerick: Wrapper induction: efficiency and expressiveness: Artif. Intell., Vol. 118, No. 1-2, pp. 15—68, 2000.

[Kus00b] N. Kushmerick: Wrapper verification: *World Wide Web J.* 3(2): 79-94, 2000.

[KWD97] Wrapper induction for information extraction: In Proc. IJCAI-97, Nagoya, Japan, 1997, pp. 729-735.

[LMK03] Kristina Lerman, Steven N. Minton, and Craig A. Knoblock: Wrapper Maintenance: A Machine Learning Approach: Journal of Artificial Intelligent Research, 18: 149-181, 2003.

[LPH00] L. Liu, C. Pu, and W. Han. XWRAP: An XML-enabled wrapper construction system for web information sources. International Conference on Data Engineering (ICDE), documents 611--621, 2000.

[MMK01] Ion Muslea, Steven Minton, and Craig A. Knoblock: Hierarchical wrapper induction for semistructured information sources: Autonomous Agents and Multi-Agent Systems: Vol. 4, Issue 1-2 (2001), pp. 93-114, 2001

[MWSD] Microsoft Windows 2000 Server Documentation, available at http://www.microsoft.com/windows2000/en/server/iis/htm/core/iigloss4.htm; Last access to site: September 2004.

[NMW97] C. G. Nevill-Manning, I. H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. Journal of Artificial Intelligence, 7:67-82, 1997.

[PS03] Sanjay Patil, Nick Simha: Integration approaches: Web services vs. distributed component models – WSJ feature: Web Service Journal, May, 2003.

[SA01] A. Sahuguet and F. Azavant: Building intelligent web applications using lightweight wrappers: Data and Knowledge Engineering, Vol. 36, No. 3, pp. 283--316, 2001.

[SOAP] Simple Object Access Protocol (SOAP) http://www.w3.org/TR/2003/REC-soap12-part0-20030624/; Last access to site: September 2004.

[SERIS03] E. Stroulia, M. El-Ramly, P. Iglinski, P. Sorenson: User Interface Reverse Engineering in Support of Interface Migration to the Web, Automated Software Engineering Journal 10(3) 271 – 301 2003, Kluwer Academic Publishers.

[UDDI] Universal Description, Discovery and Integration of Web Services (UDDI), available at http://www.uddi.org/ ; Last access to site: September 2004.

[Vin02] Steve Vinoski: Where is Middleware? : IEEE Internet Computing, Vol. 6, No. 2, pp. 83-85, 2002.

[Vin03] Steve Vinoski: Integration with Web Services: IEEE Internet Computing, Vol. 7, No. 6, pp. 75-77, 2003.

[Vog03] Werner Vogels: Web Services Are Not Distributed Objects: IEEE Internet Computing, Vol. 7, No. 6, pp. 59-66, 2003

[WSA] Web Services Architecture, available at: http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/; Last access to site: September 2004.

[WSDL] Web Services Description Language (WSDL), Version 2.0, W3C working draft, 26 March, 2004; available at http://www.w3.org/TR/wsdl20/; Last access to site: September 2004.

Internet Resources

[Artix] Artix Developer Kit. Available at:
http://www.iona.com/devcenter/artix/download.htm; Last access to site: September 2004.

[HD98b] SoftMealy 98 Talk
http://www.iis.sinica.edu.tw/~chunnan/SLIDES/softmealy98.ppt

[JTidy] JTidy, Available at:
http://jtidy.sourceforge.net/; Last access to site: September 2004.

Appendix

Table 1: Experiment data of IPM run time (ms)

		30-39	40-49	50-59	60-69	70-79	80-89
PC Quote	5	18064	25266	32644	40128	49660	57497
	10	16898	22734	29722	36368	43429	51266
	20	17378	24101	30792	37409	43925	52003
	30	17771	23797	30629	38741	45382	52472
	40	18852	24845	32117	38396	46201	53256
	50	18242	25816	32002	40256	46527	54332
Yahoo Quote	5	7342	9670	10786	12322	13631	14943
	10	7924	9254	11065	12712	13954	15306
	20	7915	9426	11416	12970	14074	15511
	30	8096	9526	11271	12788	13930	15488
	40	7614	9117	10771	12081	13266	14435
	50	8005	9369	10934	12471	13523	14816
Lycos Quote	5	1760	2205	2520	2901	2932	3166
	10	1585	1746	2115	2263	2324	2250
	20	1485	1880	2226	2252	2319	2270
	30	1320	1332	1482	1526	1693	1674
	40	1222	1379	1646	1696	1616	1735
	50	1250	1450	1691	1589	1689	1743
CNN Weather	5	10500	14808	18766	23957	28677	34306
	10	10563	14618	19070	24143	29249	34443
	20	10955	15331	19740	24888	29541	35768
	30	8864	12061	14535	17344	19921	22848
	40	8752	11421	14098	17191	19522	22279
	50	9094	11657	15125	17483	19895	23039
CBC Weather	5	2617	3504	4783	5967	7091	8477
	10	2002	3018	4009	5140	6186	7739
	20	2173	3151	4012	5175	6473	7877
	30	1901	2661	3552	4462	5345	6564
	40	2233	3292	4111	5218	6259	7588
	50	2406	3223	4251	5292	6350	7616

Table 2: Experiment data of IPM pattern number.

		Length 30-39	Length 40-49	Length 50-59	Length 60-69	Length 70-79	Length 80-89
PCQuote	5	364	373	373	373	373	373
	10	315	314	304	294	284	274
	20	315	314	304	294	284	274
	30	315	314	304	294	284	274
	40	315	314	304	294	284	274
	50	315	314	304	294	284	274
Yahoo Quote	5	165	153	143	133	123	113
	10	165	153	143	133	123	113
	20	165	153	143	133	123	113
	30	161	149	139	129	119	109
	40	159	147	137	127	114	94
	50	159	147	137	127	114	94
Lycos Quote	5	133	133	133	133	133	133
	10	114	104	94	84	74	64
	20	114	104	94	84	74	64
	30	114	104	94	84	74	64
	40	114	104	94	84	74	64
	50	114	104	94	84	74	64
CNN Weather	5	247	257	267	277	287	297
	10	247	257	267	277	287	297
	20	247	257	267	277	287	297
	30	169	159	149	139	129	119
	40	169	159	149	139	129	119
	50	169	159	149	139	129	119
CBC Weather	5	172	172	172	169	159	155
	10	153	153	153	153	153	153
	20	153	153	153	153	153	153
	30	117	107	97	87	79	78
	40	0	0	0	0	0	0
	50	0	0	0	0	0	0

Table 3: Landmark coverage rate (100% support)

	Training Sample number	30-39	40-49	50-59	60-69	70-79	80-89
	Sample 5	1	1	1	1	1	1
	Sample 10	0.963	0.963	0.963	0.963	0.963	0.963
	Sample 20	0.963	0.963	0.963	0.963	0.963	0.963
	Sample 30	0.963	0.963	0.963	0.963	0.963	0.963
	Sample 40	0.963	0.963	0.963	0.963	0.963	0.963
PC Quote	Sample 50	0.963	0.963	0.963	0.963	0.963	0.963
	Sample 5	0.9444	0.6111	0.6111	0.6111	0.6111	0.6111
	Sample 10	0.9444	0.6111	0.6111	0.6111	0.6111	0.6111
	Sample 20	0.9444	0.6111	0.6111	0.6111	0.6111	0.6111
	Sample 30	0.9444	0.6111	0.6111	0.6111	0.6111	0.6111
Yahoo	Sample 40	0.9444	0.6111	0.6111	0.6111	0.6111	0.6111
Quote	Sample 50	0.9444	0.6111	0.6111	0.6111	0.6111	0.6111
	Sample 5	0.9231	0.9231	0.9231	0.9231	0.9231	0.9231
	Sample 10	0.9231	0.9231	0.9231	0.9231	0.9231	0.9231
	Sample 20	0.9231	0.9231	0.9231	0.9231	0.9231	0.9231
	Sample 30	0.9231	0.9231	0.9231	0.9231	0.9231	0.9231
	Sample 40	0.9231	0.9231	0.9231	0.9231	0.9231	0.9231
Lycos Quote	Sample 50	0.9231	0.9231	0.9231	0.9231	0.9231	0.9231
	Sample 5	1	1	1	1	1	1
	Sample 10	1	1	1	1	1	1
	Sample 20	1	1	1	1	1	1
	Sample 30	0.9167	0.9167	0.9167	0.9167	0.9167	0.9167
CNN	Sample 40	0.9167	0.9167	0.9167	0.9167	0.9167	0.9167
Weather	Sample 50	0.9167	0.9167	0.9167	0.9167	0.9167	0.9167
	Sample 5	0.8667	0.8667	0.8667	0.8667	0.8667	0.8667
	Sample 10	0.8667	0.8667	0.8667	0.8667	0.8667	0.8667
	Sample 20	0.8667	0.8667	0.8667	0.8667	0.8667	0.8667
	Sample 30	0.8667	0.8667	0.8667	0.8667	0.8667	0.6
CBC	Sample 40	0	0	0	0	0	0
Weather	Sample 50	0	0	0	0	0	0

Table 4: MRS run time (ms).

		Length 30-39	Length 40-49	Length 50-59	Length 60-69	Length 70-79	Length 80-89
PCQuote	5	17	17	34	19	17	16
	10	16	17	17	17	15	16
	20	16	15	16	33	15	16
	30	48	16	16	16	16	17
	40	15	18	32	16	33	30
	50	16	18	16	16	16	16
Yahoo Quote	5	10	8	7	7	8	6
	10	10	11	13	8	19	6
	20	12	12	10	12	9	6
	30	50	29	10	10	7	22
	40	60	14	11	8	7	8
	50	10	14	10	9	6	10
Lycos Quote	5	9	9	9	9	9	8
	10	7	7	5	5	5	4
	20	7	5	5	5	22	4
	30	7	6	4	4	4	3
	40	7	4	5	4	4	4
	50	81	6	5	4	4	4
CNN Weather	5	15	15	13	13	27	13
	10	15	14	13	13	14	13
	20	15	16	28	13	14	13
	30	10	23	7	9	7	5
	40	8	8	8	8	34	21
	50	7	8	8	8	8	4
CBC Weather	5	10	10	8	8	8	8
	10	139	23	9	8	9	9
	20	10	8	8	8	8	9
	30	9	19	10	8	7	6
	40	0	0	0	0	0	0
	50	0	0	0	0	0	0

Table 5: Pattern numbers after MRS heuristics.

		Length 30-39	Length 40-49	Length 50-59	Length 60-69	Length 70-79	Length 80-89
PCQuote	5	9	8	7	6	5	4
	10	9	8	7	5	6	5
	20	9	8	7	5	6	5
	30	9	8	7	5	6	5
	40	9	8	7	5	6	5
	50	9	8	7	5	6	5
Yahoo Quote	5	5	3	2	3	2	2
	10	5	3	2	3	2	2
	20	5	3	2	2	3	2
	30	5	3	2	2	3	2
	40	5	3	2	2	3	2
	50	5	3	2	2	3	2
Lycos Quote	5	5	3	3	3	2	2
	10	5	3	3	3	2	2
	20	5	3	3	3	2	2
	30	5	3	3	3	2	2
	40	5	3	3	3	2	2
	50	5	3	3	3	2	2
CNN Weather	5	5	5	4	4	4	3
	10	5	5	4	4	4	3
	20	5	5	4	4	4	3
	30	5	5	4	4	4	3
	40	5	5	4	4	4	3
	50	5	5	4	4	4	3
CBC Weather	5	3	2	2	2	2	2
	10	3	2	2	2	2	2
	20	3	2	2	2	2	2
	30	3	2	2	2	2	1
	40	0	0	0	0	0	0
	50	0	0	0	0	0	0

Table 6: IPM Run time for Amazon and Chapters (ms)

		20-29	30-39	40-49	50-59
Amazon	20	461	602	680	817
	30	286	382	454	570
	40	275	364	429	493
	50	253	327	360	395
	60	181	204	213	216
	70	180	192	203	198
Chapters	20	464	557	705	886
	30	392	456	525	601
	40	272	316	376	345
	50	231	256	260	252
	70	201	161	165	160
	90	123	123	126	140

Table 7: Object coverage rate

		20-29	30-39	40-49	50-59
Amazon	20	0.9386	0.8298	0.8298	0.9255
	30	0.8404	0.8298	0.8298	0.8298
	40	0.8404	0.8298	0.8298	0.8191
	50	0.8404	0.8298	0.8298	0.8191
	60	0.8298	0.8191	0.8191	0.8191
	70	0.8298	0.8191	0.8191	0.8191
Chapters	20	1	1	1	0.9681
	30	1	1	0.9894	0.9574
	40	1	1	0.9574	0.9574
	50	1	1	0.9574	N/A
	70	0.9894	0.9574	N/A	N/A
	90	0.9574	N/A	N/A	N/A

Table 8: Time Spent in The MCS heuristic (ms)

		20-29	30-39	40-49	50-59
Amazon	20	62	91	107	140
	30	30	36	76	116
	40	42	57	91	78
	50	47	54	81	72
	60	42	21	20	21
	70	21	19	19	14
Chapters	20	62	90	117	184
	30	70	65	59	82
	40	59	50	18	15
	50	44	34	21	11
	70	16	13	0	0
	90	14	0	0	0

Table 9: Pattern number before MCS heuristics

		20-29	30-39	40-49	50-59
Amazon	20	56	70	78	81
	30	35	48	58	67
	40	35	48	58	67
	50	35	48	55	46
	60	15	18	18	11
	70	12	13	12	5
Chapters	20	58	92	104	106
	30	58	69	61	49
	40	37	35	11	2
	50	37	27	7	0
	70	10	1	0	0
	90	1	0	0	0

Table 10: Pattern number after MCS heuristics

		20-29	30-39	40-49	50-59
Amazon	20	5	5	5	6
	30	4	5	5	6
	40	4	5	5	5
	50	4	5	5	4
	60	3	4	3	3
	70	3	3	2	2
Chapters	20	7	10	11	13
	30	7	7	8	6
	40	4	4	2	1
	50	4	4	2	0
	70	2	1	0	0
	90	1	0	0	0

Table 11: Landmarks used in yahoo stock-quote service.

```
last trade:
trade time:
change:
prev close:
open:
bid:
ask:
1y target est:
day&#39;s range:
52wk range:
volume:
avg vol
(3m)
market cap:
p/e
(ttm)
eps
div &amp; yield:
```

Table 12: A summary of the comparisons among wrapper construction tools.

	WIEN	SoftMealy	Stalker	XWrap	W4F	Lixto	RoadRunner	ServiceBuilder
Requires labeled training examples	yes	yes	yes	no	no	no	no	no
Feasible to learn from a large amount of examples	no	no	no	no	no	no	yes	yes
Automatic learning process	yes	yes	yes	no	no	no	yes	yes
Can deal with nested structures	no	no	yes	yes	yes	yes	yes	yes
Can dealing with variant/missing attributes	no	yes	yes	yes	yes	yes	yes	yes
Robustness of the generated wrapper	low	moderate	moderate	low	low	low	low	high

Table 13: A summary of some factors that affect the ServiceBuilder run time[4].

	Target pattern length	Minimum support rate	Degree of similarity within/among page(s)	Average page length	Number of training examples
ServiceBuilder run time	positive	negative	positive	positive	positive/negative [5]

[4] "positive" in the table means the bigger (higher, larger...) the value of a factor, the longer the run time of ServiceBuilder and "negative" means the opposite.

[5] Depends on the actual minimum support and the degree of similarity within/among page(s), this factor may positively or negatively relates to ServiceBuilder run time. In general, when minimum support rate is high, the relation is negative and vice versa.