Structured Data Extraction

Erik Schlyter

December 12, 2007
Master's Thesis in Computing Science, 30 credits
Supervisor at CS-UmU: Michael Minock
Examiner: Perl Lindström

Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

Abstract

The internet is swarming with home-pages and sites that sells/rents products of various kinds. Often these products are published on the companies web-sites, but there's seldom a standardized way to collaborate the marketing of these products. Companies often decline to publish competitors products on their site, so it would be nice to have an independent free service gathering all information and publishing it in a centralized fashion. Since all information already is available on the net, the service only has to gather, normalize and store the information to make it searchable for a wider clientele.

This master's thesis focus on the problem of collecting this product information and the various methods that are available to do this. The main objective of this project is to implement a development environment to handle this task in an easy, perpicious and monitorable manner. The in-depth study of this thesis is focused on structured data extraction and the efficency, usability and level of automatization of the algorithms available to accomplish this.

The result of this thesis is the implementation and evaluation of PIEME (Product Information Extraction and Monitor Environment), which will apply an algorithm designed for the purpose of extracting product information in an automatized fashion.

Contents

List of Figures

Chapter 1

Introduction

The story starts with two university students with a business idea to collect information on a specified product from the internet, make it searchable and available for a wider clientele and hopefully make a dime on banner advertisement. The problem was that neither of them had any in-depth computer experience, so they sought assistance from M.Sc. students within the Department of Computing Science. The followed meetings and discussions about ideas, visions, feasibility and possible solutions gave birth to the problem statement of this master thesis: to design, implement and evaluate a complete system that could handle this task in a feasible fashion.

This report will first give a detailed description of the problem and what goals are feasible and desired. A discussion about modern research and other related work will be described with an analysis on why some methods are more appropriate than other in this specific scenario. The report will give a detailed explanation of the system design and the purposes and motivation of each designed detail. The architecture will also be discussed since this project has quite specific demands on performance and usability.

- Chapter 2 describes the problem, the actors and the nature of the vast environment we call the world wide web. Various obstacles are discussed from several perspectives and the goal of this project is formally stated.
- Chapter 3 explains the systems, algorithms and the methodology that has previously been conducted in the problem of data extraction from the web. The main focus of the theoretic part of this thesis is based on the problem of information extraction, which as been noted is only one part of the complete system.
- Chapter 4 gives an overview of the PIEME system in context. This chapter gives an explanation of the system architecture, which methods are employed and the reasons and motivations behind it. This chapter also gives a brief explanation of the design and idea of the GUI.
- Chapter 5 describes a new algorithm developed for this project's specific needs, with assumptions, feedback and evaluations from the methods mentioned in previous chapters.
- Chapter 6 explains the result and evaluation of PIEME and the developed algorithm used to automatize the product information extraction.
- Chapter 7 states the conclusions based on the results.

– Chapter 8 explains the future work and the natural next step for this project.

– Chapter 9 shows acknowledgments and gratitudes.

Chapter 2

Problem Description

The basic idea is to find several different web pages with some common advertised product, collect this information and after some compilation and normalization present it as a free web search service.

The problem could naturally be splitted into three individual components that each require individual consideration. The first part is to find and extract the desired information. The process of extracting the information has to be as automated as possible to make the system scalable, and it's this component that has the main focus of this master thesis.

When you've located the information and know how to extract it, you need some way to monitor the process to insure that the information is correct and complete. Because of the level of automation the system has to give some form of indication or warning when extracted information doesn't seem correct.

The last step in the process is to find a way to store all this information in a way that makes it available and searchable without any harrowing performance issues. A relational database system has to be managed along with an easy and user friendly web interface.

2.1 The Nature of Web Pages

When you examine different company web pages with the intention to collect their product information you stumble on a few basic revelations.

- Each company has different representations of their data
- Each company often publish different subsets of each product's properties.
- Often you have to navigate through their web page (various non-static links and search-forms) to find the information.
- The information is often stored in a structured fashion, i.e. tables etc.

2.1.1 Data representation

The sad fact about the web pages in the real world is that they're often designed and written by humans, and therefore they contain common errors often portrayed by humans. This means that the tables and pages often contain sloppy design, non-uniform

HTML code and inconsistent ways to represent their data. You will find that HTML code generated by underlying databases, such as search and query functions, are often very structured since the data is rendered from an already fixed template. The information in these templates is easy to extract since they form an easy to recognize pattern, but pages that are manually filled in by humans often contain structural errors that aren't visually noticeable on the page until you examine the underlying HTML code.

The HTML code itself has no guarantee to be written in valid XML, since the aim of these companies is to publish the information in the client's web browsers and not to make it interpretable by an autonomous extraction system. Common traits of this phenomenon is for instance that some pages contain HTML tags that miss their corresponding end tag, for example a `<B>`-tag that miss the corresponding `</B>`-tag. This is not a problem for the commonly used web browsers, since the graphical representation could be derived anyway, but a common XML parser would generate an error and would probably quit with an error code.

2.1.2 The nature of the product information

Another aspect of the desired information is that companies tends to publish different subsets of the products features and properties, which probably is caused by marketing and esthetic reasons. Say a specific product has the following properties $\{A, B, C, D\}$. Company C_1 might want to publish $\{A, B, C\}$ when company C_2 focus on $\{B, C, D\}$. This would normally not cause any problem for the customers, but for someone who collects and compile this information it would lead to null-values in the database structure. There are no obvious solutions for this problem and it should be considered in each specific case what data should be extracted. You could either choose to extract the intersection of this information, i.e. $\{B, C\}$, or you could choose a specific subset of properties and accept the derived incompleteness of the values as a limitation to the service.

In addition to the fact that they publish different information, a new problem arises when they publish the same information in different formats. If you're collecting information on books for example, some publishers might show you the authors name in full, while some other publisher only shows the authors last name. This would normally not cause any serious problem, but if some publisher gives you information on a book and another publisher has another book with the same author, how do you derive in your system that these books have the same author if one publisher says *A. Andersson* and another *Anders Andersson*? And what if a third book says *Andersson, Anders*? The resulting issue might not be a problem depending on the nature of the final service, but it is indeed a problem to be considered.

2.1.3 Web navigation - to derive the information

Depending on the vastness or nature of the product information the companies choose different ways to make it accessible for the customers. In small cases the information is legibly published on the front page in a neat table, but in most cases you have to navigate through several menus, fill in search forms and thus query their own database system for the information that is to be extracted. The information received could also be so vast that you'll need to navigate through several result pages before you've seen any chance to collect all data.

The worst case scenario is when you need a membership or a login username to gain access to the information. Some companies have no reasons to publish the information to the vast majority, and the specific product might just be a small part of a service that normally requires membership and account information.

2.1.4 Structured data, a pleasant sight

A good aspect of extracting information from web pages is that even if the page is badly formatted and contains sloppy code, it almost always have *some* sort of underlying structure. The HTML code itself promotes it with the use of `<table>` or `<div>`-tags. The rows and columns might be packed with non-consistent information and aesthetic layout tags, but they're at least confined in a table. This is at least some comforting fact when tackling this problem.

2.2 Correctness, Duplication and Validity of Data

Several problem arises apart from the details stated above depending on where you choose to extract the data. The extracted data could itself be extracted data and this complicates matters if the individual products to be displayed aren't physically unique. If each product could be distinguished this will not present any problem since you could easily apply restrictions and constraints to the database, but what if two different products with the same name and same specifications are found? As illustrated in figure 2.1 the same product could be extracted from different sources in the belief that it's two distinctly different products, and worse is if the data is manipulated in the process as a normalization for the author-example stated in section 2.1.2.

When considering this scenario it's easy to see that if the sources of data isn't chosen wisely (in some cases, there's probably not even any good solution), the data could easily be corrupted, distorted, duplicated or even missed out. Another aspect to consider is the correctness of the actual information that each company publishes. Some companies could publish the same products with misguided or poorly updated information.

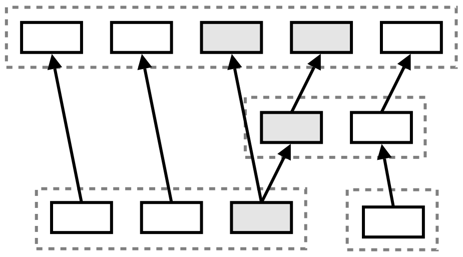


Figure 2.1: Illustrating the process of collecting already compiled data. Some data records are duplicated as they are collecting from an actor that already collected it from other source.

2.3 Monitoring the Process

Apart from collecting the data it is also an issue to monitor the process. Product companies have the tendency to update their page layout and internal structures on an irregular basis. If a system is based on collecting this information and have a set of rules on how this is to be done, these rules needs to be updated as well. If the system finds the information in a complete autonomous fashion (more on that in section 3.6) it might be sufficient, but if the system is rule-dependent for each web page the system might collect the information by a pure haphazard basis. If a web page for instance adds another column in a specific table the resulting data might say that some author's named "ISBN xxx", which is obviously incorrect.

The system clearly needs some function of monitoring this to insure that the collected data is somewhat correct. By practical performance issues it's easy to say that a human supervision would be tedious if practically possible at all. A fairly simple way to give some faint feeling of insurance would be to check each field with a regular expression, but the result could still be inconclusive since the non-matching expressions would instead generate no data at all. To exemplify this think of a expression that is set to match on string representation of $\{A, B, C\}$, when the new table have an layout like $\{A, B, d, C\}$.

2.4 Storing Data and Database Structure

The requirements of the database is quite particular in this case since the system will use it in a quite peculiar way. If you analyze the scenario you can make the following statements on the properties of this database:

– The database relations are fairly simple.

– The insertion and look-up operations must obviously be very quick.

– The entire data set have to be replaced (at least) each 24 hours.

– The system must be able to insert (or update) vast amounts of data in a short time simultaneously as customers use the look-up service.

– The need for performance could very well outweigh the need for reliability in this case, since the data is considered old and probably worthless at the end of each day anyway.

These properties give both good prospects for performance enhancements as well as mind blowing performance demands. A good thing is that the design of the relations are often very simple. A fully operational system would basically only need to populate a single table of data, or perhaps a couple if the system has some extra features or functions.

An interesting fact about the simplicity of the relations is that even if you pack this database with product information it will not require immense amount of space. Let's look at an example with a system with somewhere about 500,000 different products. Since the listed information on each product probably isn't very vast (some strings, some integers, maybe some URL and some complementary data), each row in the database would probably not take more space than a single kilobyte. If this is the case, then the entire data collection would take no more than 500 MB of space, which is far within the range of most modern systems primary memory storage. If the entire database could be kept in-memory, which is the primary assumptions of database systems today, it would mean great advancements on the performance issue.

The big bottleneck in performance would clearly be the fact that the entire database has to be replaced each day, and preferably at an instantaneous time. This sounds like a hazard but it will probably not be such a major problem in practice because the extraction process itself will most likely suffer more in performance than the database transaction.

2.5 Goals and Assumptions

The main goal is to implement a robust and easy-to-use system that handles the task mentioned above. The data extraction environment should be implemented as a graphical user interface to make it easy for administrators to guide and monitor the process. The generation of the extraction rules for each page has to be as automated as much as possible, but it's sufficient if the resulting program is semi-automated, i.e. it can generate the extraction rules with some human guidance. This presumption is made in the belief that the semantic meaning of the structured data is to difficult for a machine to understand. This limitation is acceptable with the motivation that once the extraction rules have been defined, there will be no more need for a human hand to guide the extraction process.

The desired information is assumed to be simple enough to be fitted in a single database table. More complex relational dependencies are assumed to complicated for the need of this particular application. Relationship constraints could very well be used in SQL statements provided by the administrators, but the semantic complexity of the information should not affect the extraction process. This is motivated by the fact that the purpose of the system is to extract information on a *specific* product, not a variety of objects with inherent structural relationship dependencies.

As mentioned earlier, the main focus of this thesis is the extraction process. If empirical experiences show that complex web navigation systems are required or that the web service requires extensive functional adaption these topics will only be covered briefly.

Chapter 3

Related Work in the Research Community

This chapter will cover the methods and systems that have been conducted in the area of data extraction. The first section shows a brief survey of the systems currently employed on the market. The next section will describe how the extraction process could be divided into sub problems, followed by a brief explanation of common approaches and terminology. The problem of locating the desired information will be discussed, and the systems will be divided into *semi-automated* and *automated* systems depending on their need of human influence. The goal of this chapter is to examine the general methods of data extraction in order to conclude which method is most suitable for product retrieval.

3.1 Currently Employed Applications

A brief survey has been conducted in order to find out how the actors on the market are currently dealing with this problem. Three fairly popular web-sites: AllaAnnonser[1], PriceRunner[2] and PrisJakt[3], with the similar objective have been contacted with questions regarding their methods of data extraction.

It seems that the major part of their extraction is based on manually written programs with extensive use of regular expressions. The extensive labor required is justified by the fact that many web sites contain similar structure, and therefor a lot of software can be reused. PriceRunner uses a software that basically is a XML parser, and then techniques like XSLT[4] and XPATH[5] are applied in order to extract the information. Much of the information is also retrieved by collecting pure formatted data that is exported and provided in collaboration by the administrators of those sites that are interresting.

This brief but interresting survey shows that there is at least a market for automatization in the extraction process, and the rest of this chapter will be devoted to the systems developed by the research community.

[1]http://www.allaannonser.se
[2]http://www.pricerunner.se
[3]http://www.prisjakt.se
[4]http://www.w3.org/TR/xslt
[5]http://www.w3.org/TR/xpath

3.2 Components of the Extraction Process

A thorough research has been done in the subject, and the methodology of extracting structured data from the web can be summarized in several parts. As a generalization of the extraction problem three individual sub-problems dependent on each other will be defined, namely *identifying data rich regions*, *extract the information* and *semantically define the extracted data*.

The first task is to locate the data-rich regions of the web page. Product web pages often contain lots of information that isn't interesting for the extraction, like advertisement, navigation links, company information etc. Even though this information is useful for the browsing user, it has a tendency to complicate the extraction process [20] and there are various methods on how to solve this.

The second task is to find out how to extract the actual data. Basically there are two algorithmic methods on how to do this [24], either by wrapper induction or automated extraction. Since the desired information often is originated from an underlying database, the web pages are often created dynamically by scripts and therefore they often follow a fixed template. This *hidden* template must be found in order to create the wrapper or to automatically extract the information.

Since many systems just do a syntactical extraction of the data, they tend to miss the information that is structurally implied by the page layout. Some systems solve this by extracting *nested* structures and derive this by aligning data tables when the extracting process is completed [20]. Although some systems exclude this part, this process is an important task when you're extracting information into an already known database schema.

The last part is to semantically define the data that has been extracted. As mentioned, some systems are not interested in the meaning of the data as long as the data is extracted. As we shall see, there are several methods to identify data records, with several pros and cons depending on the situation and objective of the system.

3.3 Common Approaches

3.3.1 Wrappers

Much of the early research was based on using *wrappers* to extract information, which means to write a script or a set of rule based instructions for each page layout. The idea was that a programmer would examine each page and decide what instructions was necessary, but since human intervention tends to be error-prone and costly, this approach is not very scalable.

This method is however the most fundamental way and according to [3] there are several systems like XWRAP [1], WIEN [12], Stalker [15] and Softmealy [11] that with some human guidance can simplify the generation of these wrappers.

Most modern research however is based on how to generate these wrappers automatically. Even though methods of extracting data differs profoundly, almost any method can be abstracted to some form of wrapper generation or template manifestation. Two well-known early systems in this area are ROADRUNNER [7] and IEPAD [6], which will be examined and explained shortly.

3.3.2 Tree structures

Since a HTML document is based on nested tags it can be interpreted as a tag tree or a DOM (Document Object Model) tree. Many early systems work solely with string representations, but their methodology could nevertheless be abstracted into tree structure manipulation. Even though tree representation requires the code to be strictly structured, the web pages don't always require the source to be in valid XML to be displayed correctly. There are however tools that correct the code from common errors. An example of this is JTidy[6] that is currently employed in ROADRUNNER[7] and the EXALG system [3]. Several systems require the source to be valid and therefore has an extensive use of similar tools.

3.4 Locating Data Rich Sections

In order to extract the structure from the document the non-data regions of the document has to be excluded. It is possible to omit this step in the process and derive the information anyway, but as we shall see this could lead to immense complexity increase and aggravate the heuristic functions that are used in most methods.

3.4.1 Finding tables

Yalin Wang and Jianying Hu [22] did a research on how a machine based learning approach could be applicated to detect data rich tables on a web page. They define tables as *genuine* or *non-genuine* depending on their data content and structure, and they examine the `<TABLE>`-tags to find features including layout, content type and word groups in order to classify each table. With statistical and heuristic functions they compare among other things the table layout, relation between rows and columns, the length of data in each cell and contents (images, links, texts) within each cell in the target page with a ground truthed data collection.

The problem with this approach is that you need an entire data collection of ground truthed data in order for it to work. Even though the system seems to find data-rich tables, there is no guarantee that they find the *correct* data-rich tables. If you are looking for product information you just as might find company information, addresses or etc., and most product pages (like those containing books, computers, hardware) contain lots of images and links that would aggravate their heuristic functions used for classification. Product pages *will* have fundamental differences in layout, and even if you use very varying input the result is totally dependent on the ground truthed data.

3.4.2 Identifying similarities

Another approach used by [20] is to examine similarities between different pages within the same product web site. They employ an algorithm called DSE (Data-rich Section Extraction) [21] which compares the tree structure from two different pages and sorts out the identical sub-trees. The concept is that if you do some search queries within the site the resulting pages will be identical (they are generated with the same script) except for the tables that contain information from the underlying relational database. The idea is solid and the method is effective, but it still requires the database to *have* enough information that it would require several result pages. This is true in most common sites

[6]http://sourceforge.net/projects/jtidy

(big bookstores etc), but some product pages don't have that many products. Many product pages don't even have a search form in order to do the query, as the main page is just a single page containing the company specific products.

3.4.3 Heuristics on product properties

Some systems that have a clear view of what their products properties are can use heuristics to find the tables that contain correct data. Some properties like dates, ISBN or specific product serial numbers could easily be recognized by regular expressions, and could therefore be used to identify which tables contains the desired information. Features among text strings like length, paragraph size and word occurrences can be used as heuristics to identify news information [18]. This approach is also applicable to labeling columns and identifying data semantics.

3.4.4 Human intervention

The most basic and easy way is of course to use humans to identify the sections. As discussed earlier this is error-prone, costly and not very scalable, but in some cases it is actually the last resort. Even though a system could identify the right data 95% of the time, it is sometimes required that a human monitors the process in order to guarantee the correctness of the lasting 5%. Even if humans tends to make mistakes.

3.5 Semi-Automated Wrapper Generation

3.5.1 XWRAP

The first generation of wrapper generation was semi-automated, that is, the system could find structure and data with the help of user guidance. A famous work in this area was the *XWRAP*-system[1], which was presented in 2000. This system was not only developed to extract data, but to generate general wrappers in executable code (Java). The XWRAP architecture is composed of four components:

- *Syntactical Structure Normalization.* Fetches the web page from the desired URL, cleans up bad HTML tags and generates the parse tree.
- *Information Extraction.* Identifies interesting regions and semantical tokens in the parse tree and finds hierarchical structures in order to generate the extraction rules.
- *Code Generation.* Uses the extraction rules derived in earlier step and generates the executable code.
- *Testing and Packing.* The user may now test the code with similar URLs from the same web source, and this step is used for wrapper program debugging and to make sure the wrapper is somewhat generalized towards the web source.

The XWRAP-system uses a graphical user interface that guides the user through the whole process. The user is presented with a parse tree and he/she can then click to define which regions are interesting. When the user has defined the semantical meaning of nodes in the tree, the system generates the extraction rules with a feedback-driven interaction with the user. The system prompts with heuristic based guesses in an iterative process with an attempt to describe the syntactic structure of the document in a context-free grammar. The extraction rules are specified in a well-formed XML template that is passed on to the code generator, which generates the wrapper code. The code and the extraction rules can then be tested with other pages to see which rules needs to be altered in order to make the wrapper code effective.

The systems easy-to-use interface makes the process fairly easy, but the weakness of the system is still the need of human supervision. Later research has been made in order to make the human interaction process obsolete.

3.5.2 IEPAD

This system was designed to use pattern discovery techniques in order to extract the data. IEPAD (Information Extraction Based on Pattern Discovery) [6] make use of PAT trees to find repetitive patterns in an effective way. According to [6] a PAT tree is a "Patricia tree (Practical Algorithm for Retrieve Information Coded in Alphanumeric) constructed over all the possible suffix strings". [9][14]

It is in essence a binary tree where the edges are labeled with a repetitive substring. All internal nodes are labeled with a number designating a *bit* position in the given string, and all external nodes (leafs) are labeled with a number that indicates the starting position of that particular substring in the encoded string. These properties gives the ability to retrieve possible substrings by traversing from the root node to a leaf node. This structure also gives the ability to retrieve repetitive patterns in a quick fashion by examining the internal and external node labels. The system includes three components that are necessary to the extraction process.

Extraction rule generator

The first component is called the *extraction rule generator* which takes a web page as input and generates the extraction rules for the wrapper. This component can be subdivided into four other components, namely the *Token Translator*, *PAT Tree Constructor*, *Validator* and the *Rule Composer*. The token translator converts the HTML code into tokens of two kinds, either a HTML tag token or a text token. These tokens are encoded as binary strings for use in the PAT tree, which encodes the entire document into a suffix tree structure. The validator is used to sort out undesired information since a lot of patterns derived aren't interesting for the user. The validator uses three criteria that can be configured by the user, namely *regularity*, *compactness* and *coverage*, which consists of threshold values that can be individually set or ignored. Regularities specifies the standard deviation of the spacing between adjacent pattern occurrences, compactness is a measure of the density of the occurrences (i.e. if the patterns are scattered or not) and coverage is specified by the amount of the entire document that the patterns cover.

a	b	c	d	f	e
a	b	c	e	f	-
a	b	c	e	f	e

Figure 3.1: A generalization of strings using string alignment. These strings can be generalized into `abc[d|e]f[e|-]`.

The rule composer uses a technique of multiple string alignment to generalize strings that are approximate matches, which is illustrated in figure 3.1. The idea is that several extracted patterns are quite similar and this component tries to merge those strings into general strings. When the patterns are extracted and the rules have been generated the user is prompted with the *Pattern viewer* which is the second component of the IEPAD system.

Pattern viewer and extractor module

The pattern viewer is a graphical user interface that allows the user to select from several target patterns that contain the desired information. When the user has selected which patterns are to be used, they are forwarded to the final component which is called the *Extractor Module*. This component uses the patterns on other input web pages to execute the pattern matching and information extraction.

The system takes no regard to the semantics of the extracted data and the patterns to be (in contrast to ROADRUNNER) are dependent on user input.

3.6 Automated Wrapper Generation

Several projects have been conducted to find ways to generate these wrappers automatically without the need of human supervision, and some of them will be examined and explained in this report. *ROADRUNNER*[7], and the Omini-project[2] are early works from 2001, followed by *DeLa*[20], the *EXALG*-algorithm[3] and *MDR*[13] in 2003. *RTDM*[18] was introduced in 2004 to extract web news using Tree Edit Distance in order to compare sub-trees, and *MDR-2* came as an improvement of this concept in 2005 as part of the *DEPTA* system [24].

3.6.1 ROADRUNNER

In an attempt to implement an extraction system that fully automated the wrapper generation the ROADRUNNER[7] was developed. The system performs a number of queries on the underlying database provided by the target site, and examines two different documents from the same source at the same time. The similarities and differences are compared to generate a general union-free regular expression that matches both documents.

The idea is to take a document as an initial wrapper page, and tag by tag compare it with a second page (called the *sample*-page) and progressively refine it by solving mismatches until a general wrapper page is generated. The mismatches are divided in *string mismatches* and *tag mismatches*, where the tag mismatches are used to identify either *optionals* or *iterators*. The basic assumption of this method is that data originated from the database should be different from different search queries, whereas the data (text, tag-structure) generated by the HTML template script should be identical.

String mismatches

When two strings differs from the documents it is assumed that these strings represent some form of element in a data record. When this happens the wrapper page (initially the first page) is modified and the string is replaced with `#PCDATA`, which is a token that the system identifies as a data field element.

Tag mismatches

Different tag structures often indicates some optional field or some repeated pattern. This happens for example when two different data tables are compared which contains different number of rows. To solve this mismatch the system first tries to see if the tag mismatch could be rounded up in a repeated pattern. The system finds the *initial* and *terminal* tag for the mismatched block and defines it as a *square*. This square is then matched against an upward portion of the sample to see whether it indicates a repeated pattern or not, for instance if it represents a row in a table. If a repeated pattern is found the wrapper page is modified with an iterator token, which in manners of regular expression would be to enclose the body within a statement like `(...)+`.

If the system fails to identify an iterator it does a cross-search in the wrapper and the sample page to see if this mismatched tag could be skipped. If the system is available to resume parsing (continue identifying similarities) after the mismatched block it is marked as a *optional* block, which would be specified by a statement like `(...)?`. A sample execution of the algorithm is illustrated in figure 3.2, where both a iterator and an optional is detected and written into the final wrapper expression.

This is a somewhat simplified description of the algorithm as the real world often present much more complex examples. The method of finding patterns is recursively defined since the process of identifying iterators could lead to more iterators or optionals. New mismatches could be found during the upward search which generates *internal* mismatches, which is dealt with in the same manner as other mismatches with the exception that the two *squares* within the document are compared instead of two different documents.

Generalizing the wrapper

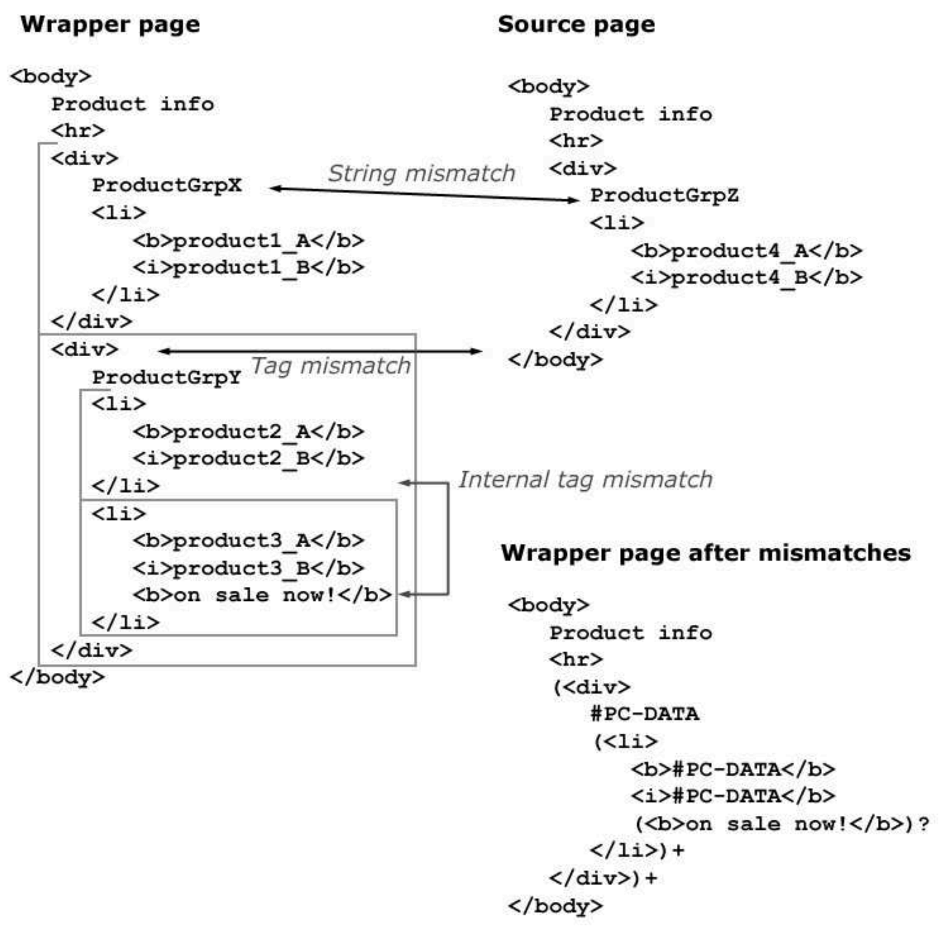
The ROADRUNNER system examines several web documents from the target site in order to produce a wrapper expression that is as general as possible. The result is generated as a HTML document with the same internal tag structure, but the fields are specified with labels $\{A, B, C, ...\}$. The result is promising but there are a few remarks worthy of mentioning.

The system succeeds in identifying data fields within the document, but this method requires the target site to provide several pages of query results. As stated in 3.4.2 this is not always the case, and a problem arises when the set of tuples generated by the queries contains a field with a homogenous content. As an example of this consider a search query on a bookstore that only (for this or that particular time being) have books from one designated publisher. The field that represents the publisher would be interpreted as a common text label and not a field originated from the underlying database.

The system also lacks the ability to automatically define the semantic meaning of the extracted data. The goal of the project was only to find ways to extract data from sites without any prior knowledge, not to extract *specific* data or to define the meaning of the content. Since the labels are specified $\{A, B, C, ...\}$ it requires a human hand to define the semantics.

3.6.2 Omini

Another attempt to fully automate the extraction process was conducted in 2001 and called *Omini* [2]. The aim was similar to that of ROADRUNNER, to make the extraction fully independent of human intervention, but the approach was more concentrated



```
<body>
  Product info
  <hr>
  <div>
    ProductGrpX
    <li>
      <b>product1_A</b>
      <i>product1_B</b>
    </li>
  </div>
  <div>
    ProductGrpY
    <li>
      <b>product2_A</b>
      <i>product2_B</b>
    </li>
    <li>
      <b>product3_A</b>
      <i>product3_B</b>
      <b>on sale now!</b>
    </li>
  </div>
</body>
```

```
<body>
  Product info
  <hr>
  <div>
    ProductGrpZ
    <li>
      <b>product4_A</b>
      <i>product4_B</b>
    </li>
  </div>
</body>
```

```
<body>
  Product info
  <hr>
  (<div>
    #PC-DATA
    (<li>
      <b>#PC-DATA</b>
      <i>#PC-DATA</b>
      (<b>on sale now!</b>)?
    </li>)+
  </div>)+
</body>
```

Figure 3.2: A sample execution of the ROADRUNNER algorithm, similar to that illustrated in [7].

on analyzing tree structure. The system consists of three phases that performs the extraction process.

Phase 1: Prepare the web document

The web page is fetched from a user specified URL and is transformed into a well-formed document using syntactic normalization algorithms. After the document has been cleaned a tag tree representation is constructed based on the nested structure of start and end HTML tags.

Phase 2: Locating objects of interests

This phase tries to locate the data rich regions of the page by using two consecutive steps. The first step is called *Object-rich subtree Discovery* and the goal of this task is to locate the *minimal* subtree of the document which contains all the objects of interests. This is done by combining three heuristic evaluations of the subtrees, which includes the number of children in each node (the *fanout* of the branch), the size of data in bytes in each node and the count of subtrees.

The next step is called *object separator extraction* which is responsible for deriving which tags (or nodes) are semantically defined as separators of the data records. Five different heuristic functions are used that independently rank each candidate tag, and a statistical probability function is used to combine these measures. The functions include the *standard deviation heuristic* which calculates the distance (in characters) between two consecutive occurrences of a candidate tag. The *repeating pattern heuristic* counts the number of occurrences of all pairs of candidate tags that have no text between them. The intuition behind this is that tags that have a closely repeated pattern often are used to designate some form of data field layout. *Identifiable path separator tag heuristic* ranks the tags by examining the specific tags that are normally used as separators. An example of this is that `<tr>` or `<td>` tags are often used as separators within the `<table>`-tag. *Sibling tag heuristic* counts pairs of tags that are immediate siblings in the minimal subtree, and the candidate tag is ranked by counting the occurrences of these pairs. The final function *partial path heuristic* counts the number of paths that exists from each candidate tag to any other node reachable within that subtree (see [2] for more details).

Phase 3: extracting objects of interest

The final phase is responsible for extracting the data by using the separators defined in previous step. After the object separators are chosen the desired objects must be extracted with the regards that sometimes the separators are between the data object, and sometimes they reside in a subtree under the separator.

3.6.3 DeLa

The *Data Extraction and Label Assignment* [20] system consists of four components whose structure is quite similar to earlier approachers with some new features and add-ons. A *form crawler* performs queries to a web page to generate the documents, a *wrapper generator* finds a common regular expression, a *data aligner* derives nested data and the *label assigner* tries to define the semantic meaning of the extracted data.

As a form crawler they use an existing web crawler called *HiWe* [17] to perform all necessary search queries. The resulting web pages are feed into the wrapper generator which is greatly inspired by the IEPAD [6] system. First they employ the *DSE (Data-rich Section Extraction)*-algorithm [21] to sort out non-interesting data from the documents, which works by comparing two different documents from the same source to identify similarities (mentioned in section 3.4.2).

When the data-rich sections have been identified they build a token suffix-tree of the tokens in order to locate continuous repeated (C-repeated) patterns. The suffix-tree is iterated and a *pattern tree* is generated to expose which patterns are dependent on each other to be found, as some patterns can't be extracted until some others has been defined. The pattern tree starts as a empty root node, and each discovered pattern is inserted as child to that root node.

After a pattern from the suffix tree is identified, the patterns discovered right after that are inserted as children to that node of the pattern tree. The pattern with the highest nested-level in the tree is chosen as the general pattern for that web page. As the C-repeated patterns are interpreted as kleene-star in regular grammar, this method however isn't sufficient to identify optionals or disjunctions in the structure. To identify these the system downloads several web pages and merges the derived expressions using a string alignment algorithm similar to that of IEPAD, as illustrated in figure 3.1.

Aligning nested data and assign labels

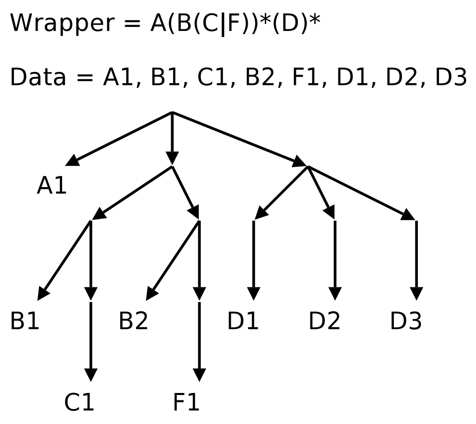


Figure 3.3: A tree representing nested levels of a regular expression, as illustrated in [20].

The innovation of this system lies in the data alignment and label assignment components. To extract nested data into a table similar to that of a relational DBMS they employ a *data-tree* for the regular expression generated by the wrapper (see figure 3.3). The tree is generated by examining the nested structure of the expression, and each nested expression is defined as a child to the enclosing expression as the tree is generated recursively. Each node represents a partial table, and these tables are merged by constructing unions and Cartesian products of the subtables, as illustrated in figure 3.4. Before the data is inserted an attribute separation is performed. The HTML tags are stripped and the system tries to find what the actual data is in each cell. The assumption here is that some cells contain comma-separated strings or other *noisy* information that is not interesting for the extraction.

The label assignment component use heuristics to define the semantic meaning of each column in the table. The form element labels in the web query page are examined to retrieve some clues, as most search engines often provide some sort of label to identify the text box for the user. Labels are also searched within the parsed document and some cells can be identified by their formats. For instance, a date usually follows a significant format ("dd/mm/yy") and email addresses contain the symbol "@".

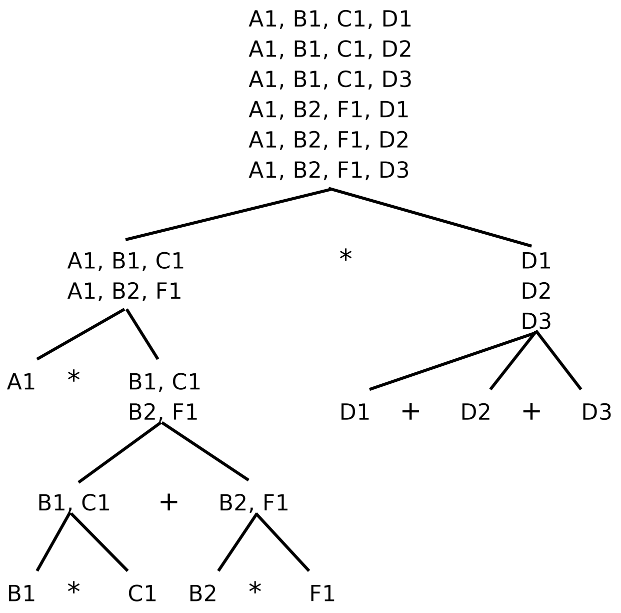


Figure 3.4: A datatree generating the final table through a recursive merging using unions and Cartesian products, as illustrated in [20].

3.6.4 EXALG

The EXALG[3]-algorithm uses a rather different approach from earlier works. The system works in two stages, the first is called *ECGM (Equivalence Class Generation Module)* and the second is called *Analysis Module*. The main objective of the first stage is to find and derive *equivalence classes*, which is defined as the maximal set of tokens having the same number of occurrences in each input page. A token in this system is defined as a word or a HTML tag, and the module counts the occurrences of each token to find what words or tags are commonly used in the set of input web pages. The assumption is that the repeated occurrences of these tokens often is generated by the page template. EXALG also differentiate the meaning of these tokens by examining the relation between their occurrences, which can be observed by comparing different paths from the root in the HTML parse tree. The sets of these equivalence classes are refined in a iterative process within this module (the reader is referred to [3] for more detail about this process) and the output is these sets and the web pages represented as a set of strings of these tokens. This is used by the analysis module to generate the templates.

3.6.5 MDR

The algorithm proposed by *MDR (Mining Data Records in Web pages*) [13] compares nodes in the HTML tag structure to generate a wrapper. The three steps to complete this task is performed by first building a tag tree representation of the web page, mine it for data regions by comparing strings and sub tree and then identify data records within each region. The method to accomplish this is to generate *generalized nodes* which is defined as similar sibling nodes with the properties that they are adjacent and have the same parent. The function is a recursive depth-first comparison where all combination of each child nodes are compared in order to find which nodes that could be grouped together. During the first iteration of each node the comparison is done consecutively, the second iteration compares them two by two and so on. For example, given a set of nodes $\{A, B, C, D, E, F\}$, the first iteration compares $(A - B)$, $(B - C)$, $(C - D)$, $(D - E)$, $(E - F)$, followed by $(A, B - C, D)$, $(C, D - E, F)$ and $(A, B, C - D, E, F)$. Each node is furthermore examined recursively to discover nested tag structures.

The data records in the tree nodes are identified by comparing string edit distance between different nodes. An improvement of this algorithm was presented in the DEPTA system [24], which will be discussed in section 3.6.7.

3.6.6 RTDM

The *Restricted Top-Down Mapping* algorithm[18] uses the concept of *tree edit distance* to compare tree nodes in the web documents. The system was implemented to extract web news and has some different presumptions compared with other approaches, as most news sites follow a very typical pattern. Most news sites contain a single body of information, followed by a title, author, date, etc, in contrast to product pages which often contain several products on each page. The tree edit distance is defined as the total cost of transforming a tree T_1 into another tree T_2 by using operations like vertex insertion, deletion and replacement. The transformation could be defined as a *mapping* between two trees, and that mapping is used to generate a generalized tree that could match data from both T_1 and T_2.

The system works in three steps. Firstly it crawls news portals to fetch interesting web pages, and secondly it generates clusters of pages that share common features using a hierarchical clustering technique described in [23]. The extraction step is conducted by generalizing a cluster into what they call a *node extraction pattern* (ne-pattern), which is defined as a rooted ordered labeled tree that contains special wildcard vertices. The wildcards can be *single*, *plus*, *option* or *kleene star* which is similar to the notation used in regular expressions. The ne-pattern could thus be seen as a regular expression for trees and therefore used to extract the data. The generation of these ne-patterns is done by creating compositions of the trees (using RTDM to generate mappings) from the web pages of each cluster.

Note that since the aim of this system is to extract web news the layout and content of the web pages are quite different from usual product web pages. They use a data labeling technique to semantically define the extracted data, but the function is heuristically based on just web news and is thus not applicable on general product information.

3.6.7 DEPTA

To improve the efficiency of MDR a new algorithm was presented as a part of the *DEPTA (Data Extraction based Partial Tree Alignment)* system [24]. The system uses a two-step strategy where the first step is to employ MDR-2 (which is an improvement of the previous MDR) and the second is to use a partial tree alignment algorithm to align and extract corresponding data items from the discovered data records.

The new version of MDR includes a visual examination of the document. Each HTML element is interpreted as a rectangle that is derived by calling the embedded parsing and rendering engine of a web browser, and a tag tree is constructed by their containment relationship. This method make the process of identifying data regions more accurate and reliable.

Since a single data record can be contained within a nested sub tree, the system defines a *data region* as the sub tree that contains all elements within a single data record. Note that a sub tree doesn't necessarily contain a single data record, as a sub set of elements from data records could be within some tree and the rest of the elements could be within an other, as illustrated in figure 3.5.

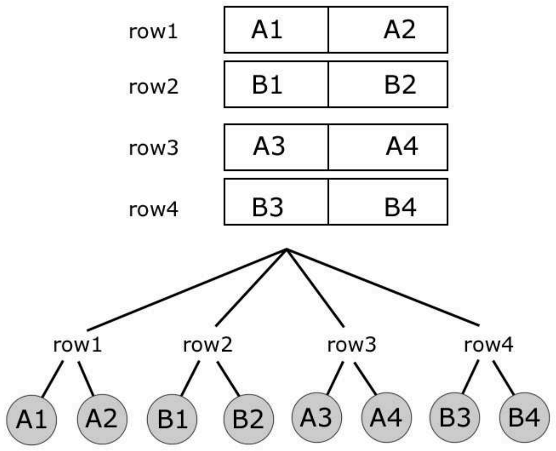


Figure 3.5: Multiple records within several sub nodes, similar to the illustration in [24].

The extraction process can be divided into two consecutive sub-steps. Firstly a rooted tag tree is constructed from each individual data record, then each tree is aligned using a tree matching technique based on partial tree alignment. The tree with the most data items is used as a *seed* tree, and each tree is then compared in an iterative process to insert each unaligned sub node into the seed tree. The resulting tree is then used as a template to extract data records into a database table.

The MDR and MDR-2 don't do any semantic considerations on the extracted data, but since the user usually is only interested in particular data or product information, the MDR provides an option to only extract data based on heuristic indicators, e.g., image, price, string matches and others.

3.6.8 WDE

A new way of using tree edit distance to extract data was presented in 2007 called the *Web Data Extractor* [16]. In similarity with MDR they compute clusters of similar tree nodes using tree-edit distance, but in contrast they use an algorithm that approximates the cost due to the performance cost of finding the actual edit distance. They also assign different weights to nodes depending on their height and internal depth.

All possible *candidate* records are enumerated and compared using tree edit distance, and the records are clustered into sets based on their similarities. These clusters are used to generate data extraction patterns which according to their own evaluation is superior to MDR in regards of loosely structured records (in contrast to strictly structured documents, like big tables etc). At the time of writing the exact details of this method is not yet presented, but the reader can verify the basic methodology of this system in [16].

3.7 Visual Content

Although most research has been conducted in examining the structure of web pages, some attempts have been made to extract structure based on visual layout of documents. According to [19] text-lines, paragraphs and columns can be identified by grouping objects (characters, words, images) according to their structural layout relationships. Their system was designed to examine PDF documents and is based on an machine learning approach to examine different documents within a training set. Although the principle is interesting to consider when extracting product information, the idea would probably not be applicable as a stand alone extraction system. Visual content examination is however a good supplement to other techniques as demonstrated by the DEPTA system described in 3.6.7.

Chapter 4

PIEME - Product Information Extraction and Monitor Environment

This chapter will explain the details about the implemented system, and the presumptions and motivations behind the design. The fundamental design is based on the following observations:

– The system requires a visual view of both a monitoring state as well as a view for editing and evaluating wrappers.

– Most of the modern methods described in this report are very good, but they're never 100% accurate. The system will therefore provide a very basic opportunity for generating wrappers manually, and later on patched with a more effective and semi-automated method. If a site with a very obscure content is found and the automatization fails, the system could still rely on a manually written wrapper that describes specific extraction rules with the aid from regular expressions.

4.1 Software Architecture

For practical and system independence reasons the PIEME system was written in Java. For simplicity the existing database management system *PostgreSQL*[1] is used with *JDBC*[2] as a means of communication. A standard web server application, in this case Apache[3], in combination with a PHP scripting module is used to present the search service for the general public. As illustrated in Figure 4.1 the components of the system could be duplicated and distributed in order to eliminate performance bottlenecks if necessary. The whole system could also be located on a single machine if the usage is in a smale scale.

Each site is represented as a *task*, which consist of basic site/company information, status, schedule and a wrapper program henceforth called a *task program* or just *program*. Each program consists of a sequence of *operations* that is executed according to

[1]http://www.postgresql.org

[2]http://java.sun.com/javase/technologies/database/

[3]http://httpd.apache.org

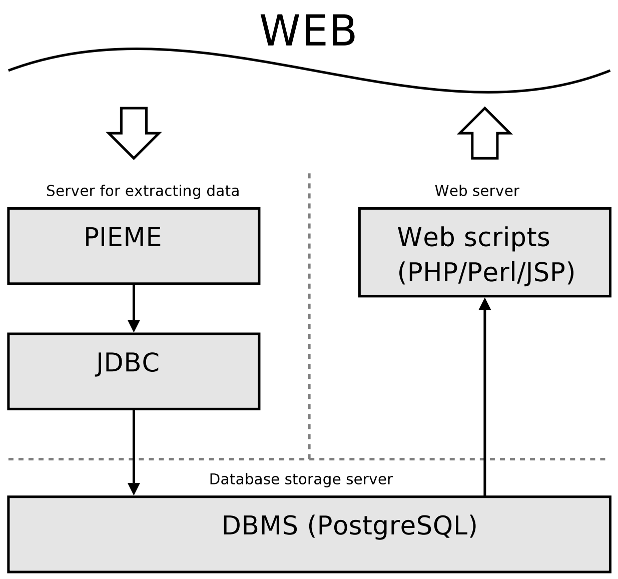


Figure 4.1: A software architectural overview of the complete system. The extraction component and the web service could be duplicated straightforwardly if necessary for performance reasons. In order to duplicate the DBMS some kind of inter-database communication would be needed, which is not provided by this report. A work-around would be to let the extraction system send duplicate transaction calls to each database machine.

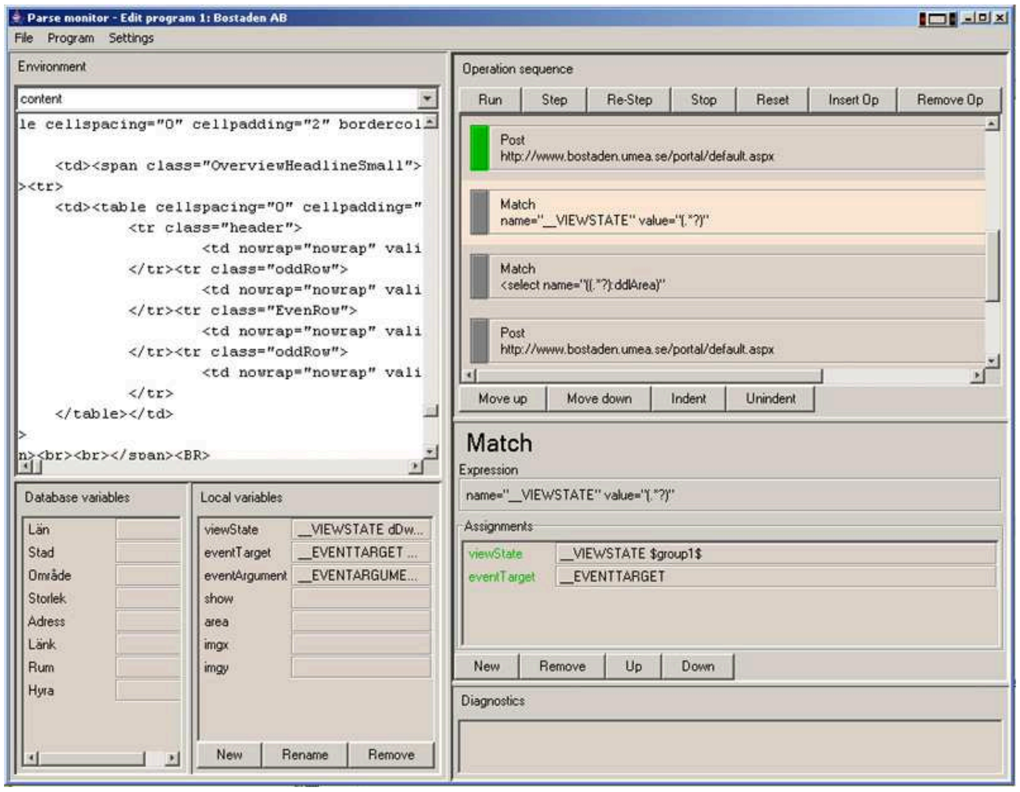
the standard imperative paradigm. The set of operations available ranges from HTTP request/post operations to several operations for manipulating the document with regular expressions (*match*, *substitute*, etc.). There are also operations for creating loops and conditional branching like *if*- and *for each*-statements. PIEME executes each task in an iterative manner according to each individual schedule. When the execution of a program terminates successfully a transaction towards the underlying database is made and the new data replaces the old.

It has been observed that the writing of the corresponding regular expressions for the wrapper program is much more complicated than originally estimated. The execution of regular expressions in Java contains recursive calls, which if the expression is badly written and the document contains an exhaustive ammount of matches, can make extensive requirements on the system's stack memory. If the administrator of the system doesn't write the expressions very carefully it is very possible that the execution will generate a runtime error (which is more serious than just throwing an exception) due to unsufficient memory assets, and therefore make the JRE (Java Runtime Environment) crash. This is obviously not acceptable in a monitoring system and it's therefore recommended that the actual retrieving executions (not counting the test executions made during the program development process) is made in separate JRE:s controlled by the underlying operating system. PIEME therefore provides means of performing task executions without the graphical user interface, which could easily be controlled with the help of simple shell scripts.

In the light of this observation it was decided that the underlying filesystem had to be a sufficient mean of synchronization between different JRE:s. Each task is stored as a directory on the designated computer's filesystem, and each directory contains the files describing the program and the results of the executions. The JRE controlling the GUI could thereby retrieve and display the result and status of executions performed by a parallel JRE by polling timestamps on task directories.

4.1.1 Generating the wrapper programs

In order to define the wrapper programs a clear view of the operations, HTML content and current variables is necessary. Figure 4.2 shows a screen shot of the edit-view in PIEME.



content

```
le cellspacing="0" cellpadding="2" bordercol

    <td><span class="OverviewHeadlineSmall">
><tr>
    <td><table cellspacing="0" cellpadding="
            <tr class="header">
                    <td nowrap="nowrap" vali
            </tr><tr class="oddRow">
                    <td nowrap="nowrap" vali
            </tr><tr class="EvenRow">
                    <td nowrap="nowrap" vali
            </tr><tr class="oddRow">
                    <td nowrap="nowrap" vali
            </tr>
    </table></td>
>
n><br><br></span><BR>
```

Län	
Stad	
Område	
Storlek	
Adress	
Länk	
Rum	
Hyra	

viewState	__VIEWSTATE dDw...
eventTarget	__EVENTTARGET ...
eventArgument	__EVENTARGUME...
show	
area	
imgx	
imgy	

```
Post
http://www.bostaden.umea.se/portal/default.aspx
Match
name="__VIEWSTATE" value="(.*?)"
Match
<select name="((.*?):ddlArea)"
Post
http://www.bostaden.umea.se/portal/default.aspx
```

name="__VIEWSTATE" value="(.*?)"

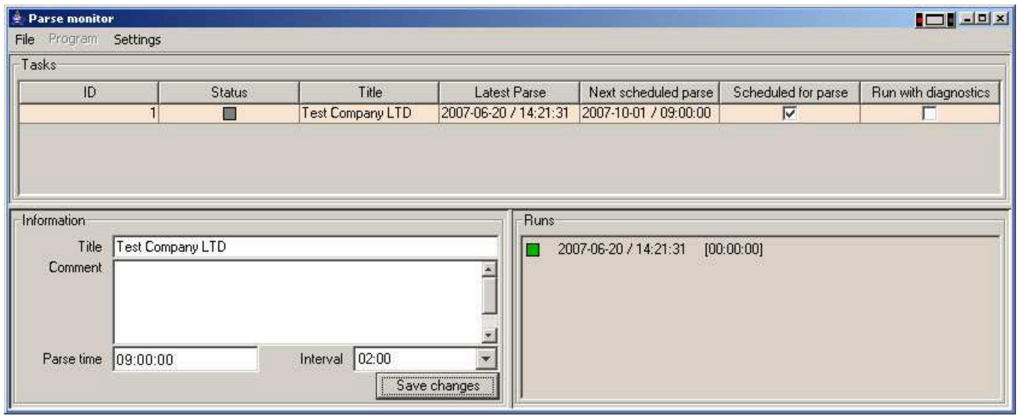
viewState	__VIEWSTATE $group1$
eventTarget	__EVENTTARGET

Figure 4.2: Edit Program View in PIEME

The view state consists of several panels to present the wrapper development. The upper panel to the right shows the sequence of operations (the program) and the properties of each operation below. The user can use these operations and assign the matching groups from the regular expressions to variables, either defined localy or variables designated to the database. The system gives the user the opportunity to step and re-step operations to test the functionality of the program. This view can also be used as a diagnostic feedback when programs fail to execute in the monitoring process, due to irregular or mismatching input from the web sites.

4.1.2 Monitoring the process

When the wrapper programs are defined it is necessary to monitor the executions in order to detect failure of extraction or when web sites fail to respond. Figure 4.3 shows a screenshot from the monitor view. Only one web site is entered at the presentation example, but the system is designed to display numerous sites and the aim of the functionality is to provide the ability to sort out and detect sites that failed last execution. When a failed execution is detected the user can use the view state to see which step in the program failed. If the option `Run with diagnostics` is set the system will save the environmental state of the program if the execution fails. This means that the user can inspect the state of the variables and current content from the extraction during the debug process. This functionality is optional since it's not always necessary and it requires more memory storage.



ID	Status	Title	Latest Parse	Next scheduled parse	Scheduled for parse	Run with diagnostics
1	■	Test Company LTD	2007-06-20 / 14:21:31	2007-10-01 / 09:00:00	☑	☐

Title: Test Company LTD
Comment:
Parse time: 09:00:00
Interval: 02:00

🟩 2007-06-20 / 14:21:31 [00:00:00]

Figure 4.3: Monitor View of the PIEME system.

Chapter 5

Automatizing the Extraction for This Project

5.1 Notes on Previous System

With observance of the study of previous system it can be concluded that none of the modern methods mentioned in chapter 3 is completely applicable on the problem stated in this project. The automated extraction methods require several pages of input and as stated this is not always the case, and as most of them don't take semantic meaning of the fields in consideration it will complicate matters when you are extracting a *specific* product group from a document.

The automated methods distinguish the fields by comparing edit-distance and have a heavy reliance on several threshold values, which will result in that the quality of the extracted information would be based on how you "turn the knobs". In order to extract correct information it is concluded that these methods are not sufficient since their approach is not based on prior knowledge (in this case, the semantics of the fields).

The aim of the implemented algorithm in this project is to generate some kind of wrapper based on the semantic meaning of some specific fields. Given a document and some semantic clues of some fields describing an element, the algorithm should derive extraction rules based on some self-learning principal. With this in mind the system with the most similar assumptions is actually the XWRAP-system, which is the earliest system described in chapter 3.

5.2 Assumptions of the Derived Algorithm

The chosen approach of this algorithm is to describe the document and the expected data with a tree-based regular expression that can recognize and distinguish fields from other fields and fields from noisy non-desired data. The reason for chosing a regular expression-based approach is because it's the intuitive method of extracting data when writing extraction programs manually, and by observing the repeated patterns of a document it is a method that is bound to have some possibility of automatization. There are however some serious drawbacks with generating an expression, not only in regards to the generation process (as it has been proved to be formally impossible with positive samples alone [8]) but also to the extraction process. Even if you have a correct regular expression describing a language it is not guarenteed that the expression could be used to correctly identify each field in an input. Given an example where there are three categories of data fields named $\{A, B, C\}$, and three categories of input values in the document named $\{x, y, z\}$. The idea is that each value in the input document should be mapped to a data field, that is $x \mapsto A$, $y \mapsto B$ and $z \mapsto C$. A regular expression containing the data fields could thereby be used to describe the input document and map each value to the correct data field. This works as long as the correct mapping could be identified, but if the mapping between input values can not be distinguished, the extraction will be impossible if the input has ambiguous interpretations in regards to the regular expression.

This is made clear with a more project specific example. Let's say a document consist of a nested definition with the values $\{x_{1-3}, y_1, z_1, y_2, z_2, y_3, z_3, x_4, y_4, z_4\}$. In this example the x-values specify a category and the y and z values specify properties of products belonging to a category. A correct interpretation and extraction is visualised in Figure 5.1, but this interpretation is impossible to guarentee even if you have the correct regular expression $(A(BC)*)*$, since it could just as easily be interpreted as Figure 5.2.

A	B	C
x_{1-3}	y_1	z_1
x_{1-3}	y_2	z_2
x_{1-3}	y_3	z_3
x_4	y_4	z_4

Figure 5.1: A correct interpretation of the string $\{x_{1-3}, y_1, z_1, y_2, z_2, y_3, z_3, x_4, y_4, z_4\}$

A	B	C
x_{1-3}	y_1	z_1
y_2	z_2	y_3
y_2	z_3	x_4
y_2	y_4	z_4

Figure 5.2: An incorrect interpretation of the string $\{x_{1-3}, y_1, z_1, y_2, z_2, y_3, z_3, x_4, y_4, z_4\}$

This is a rather extreme example and often the correct mapping can be derived by examining closely located constants and in most cases the tag structure for different elements are different enough to distinguish the elements. With the same example it is also demonstrated why optional and disjunctional fields are omitted in this algorithm. In the above example it is impossible to distinguish whether the field A is an optional field or (like the example) an indication of a nested definition of categories, and thus the learning process would have to deal with ambiguity. With that observation optional and disjunctional fields have been designated for future work in order to keep this project within the scope of a master's thesis. This means that it is assumed that each element value has a single way of representation in accordance to its tag structure.

5.3 The Algorithm

The implemented algorithm follows several stages in order to derive the extraction pattern and extract the data.

5.3.1 Generate a tag structure

The document is parsed and converted into a normal HTML tree tag structure with some exceptional divergence. Observations have shown that sometimes the desired data is located within the attributes of the tags, and therefor each attribute is interpreted as a new sub-tag with that attributes value as content. These new sub-tags are then located at the beginning of that tag's list of sub-tags. Since contents of the each tag can be located between sub-tags the following definition is made. A *tree-node* (*T-node* for short) is a representation of a HTML tag or an attribute tag, consisting of a description (`A`, `BR`, `TABLE`, `TR`, etc) and a list of sub-trees. A *data-node* (*D-node*) is a special leaf node representing data content of a tag. Several D-nodes can be children of the same T-node as long as they aren't consecutive. T-nodes generated by tag attributes have a single D-node per definition. Figure 5.3 shows a tag tree generated by some sample HTML code.

```
<html>
  <head>
    <title>Sample page</title>
  </head>
  <body bgcolor="black">
    <a href="http://www.cs.umu.se/">A sample link</a>
  </body>
</html>
```

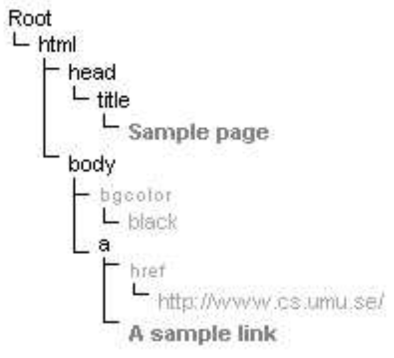


Figure 5.3: A tag tree generated by sample code. Each leaf node in this example are D-nodes. The light-gray nodes show T-nodes generated by tag attributes.

5.3.2 Define semantics

When the tag structure is generated the user prompts with a GUI where the semantics can be defined, which is done in a fashion similar to that of XWRAP discussed in section 3.5.1. The user highlights designated D-nodes, which can be one of the following: a *variable* from A to Z, a *constant* value, a *wildcard* or an *unknown* value. Each D-node is unknown from the start and the user can then observe the document and select which D-node should be interpreted as a variable or constant. Wildcard values are used when nodes contain data that isn't interresting for the extraction, but still is important to recognize and separate structures of different types of T-nodes.

5.3.3 Generate extraction template

The semantic clues given by the user is used to generate the extraction template. Firstly we can make some interresting conclusions based on earlier assumptions.

– Since disjunctions are omitted, each type of variable, constant or wildcard will share the *same* unique path from the root to the leaf.

– Since each type of variable, constant or wildcard have a unique path, *every* D-node in the input document that doesn't share the same path can't belong to any of the selected types. Thus, it can be removed in order to improve and simplify the process, which is an effective way of locating data-rich regions discussed in 3.2.

– Although all non-applicable D-nodes can be discarded, it doesn't mean that all remaining nodes should be interpreted as any of the labeled D-nodes. Some D-nodes share the same path as labeled D-nodes but they reside outside any repetitive patterns inferred by the node inference process. This will be explained with examples in section 5.3.4.

In light of these observations we will make the following definitions. A *data tree* (*D-tree*) is the original tree structure parsed from the document. A *intersection template tree* (*I-tree*) is a tree generated by merging all selected D-nodes into a minimal tree only consisting of merged path-ways from the root to each selected D-node. This I-tree describes each labeled D-node and can be used to make an intersection between it and the original D-tree, to sort out and discard non-interresting information. Figure 5.4 examplifies a labeled D-tree with the corresponding I-tree in figure 5.5. A *match tree* (*M-tree*) is the final pattern used to extract the data, which consist of a regular expression-type tree that describes repeated patterns.

Since the I-tree shows the merged path from the root to each labeled D-node, the goal now is to convert this I-tree into a complete M-tree. An I-tree can be generated in linear time, but as illustrated by the example the variabels B, C and D share the same path, and thus the I-tree alone isn't enough to distinguish the variables from each other. As noted the I-tree is used to create an intersection which will result in a simplifed D-tree without the non-interresting D-nodes. In this case the tree-nodes `colspan` in figure 5.4 will be discarded, as they are not interresting for the process.

5.3.4 Node inference

In order to create a M-tree each ambiguous node in the I-tree must be examined in their context of the D-tree in order to make a correct interpretation of their pattern of occurance.

This is done by taking the corresponding node in the D-tree and create a matrix of all possible interpretations of that nodes children. Each labeled sub-node in that node is compared with each other nodes to see which node can be mapped to which. Any variable or wildcard can be mapped to an unknown node as long as they share the

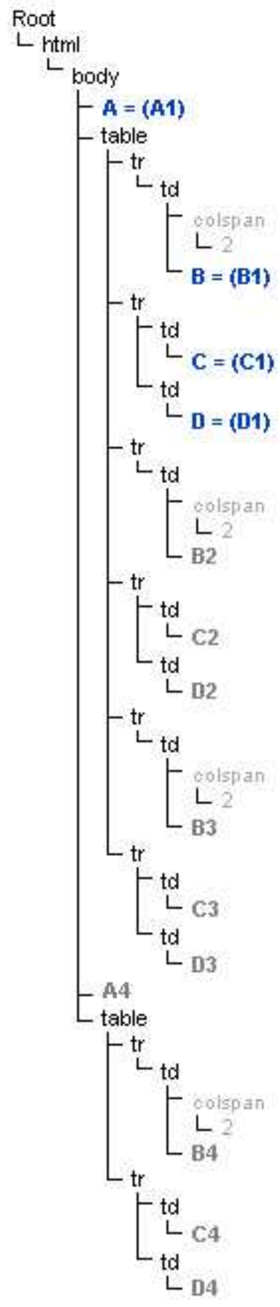


Figure 5.4: A user labeled D-tree

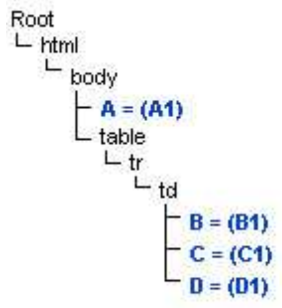


Figure 5.5: The corresponding I-tree from the D-tree in figure 5.4

same structure. Constants can be mapped to unknown if their data content match a string comparission. This method creates an $O(n^2)$ complexity in resemblance with the method employed in MDR (section 3.6.5), but in addition to the discovery of repeating patterns, it also labels the nodes semantically based on the nodes labeled by the user.

To examplify this we take another example beyond the one stated above. Let's say we have a node with three different types of D-nodes labeled A, B and C. We also have several unlabeled D-nodes a, b and c that should be mapped to the corresponding variable. In this example we say that a is distinguishable from b and c by tree-structure (let's say that the each D-node actually resides in sub-trees distinguishable by structure, as variables never are distinguishable without heuristics). If the content of the node is the following set of nodes $\{A, B, C, b, c, b, c, a, b, c\}$, it would generate the matrix shown in figure 5.6.

The nodes in the row header specify the sub-nodes in the D-tree and the nodes in the left-most column specify the sub-nodes containing labeled D-nodes in the same order has they occur in that branch of the D-tree. Note that labeled nodes never map to other labeled nodes as they are defined by the user.

-	A	B	C	b	c	b	c	a	b	c
A	A	·	·	·	·	·	·	a	·	·
B	·	B	·	b	b	b	b	·	b	b
C	·	·	C	c	c	c	c	·	c	c

Figure 5.6: A combinatorial matrix generated by the nodes $\{A, B, C, b, c, b, c, a, b, c\}$.

The matrix show how each labeled node could be mapped to an unknown node, and the trick now is to make a path from left to the right and include as many of the labeled nodes as possible. To do this you start from the top left corner and step down-right until you find an empty cell or the bottom of the matrix. When this occurs you step to the top of that column and continue. When doing this in the above example a couple of sub-paths will emerge as shown in figure 5.7.

-	A	B	C	b	c	b	c	a	b	c
A	**A**	·	·	·	·	·	·	**a**	·	·
B	·	**B**	·	**b**	-	**b**	-	·	**b**	-
C	·	·	**C**	-	**c**	-	**c**	·	-	**c**

Figure 5.7: Illustrating the steps taken in figure 5.6.

The following observed sub-paths are $\{A, B, C\}$, $\{b, c\}$, $\{b, c\}$ and $\{a, b, c\}$, which can be used to recognize repetitive patterns. These patterns can be used to create a simplistic automaton, which would yield the transitions $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow B$ and $C \rightarrow A$. To finally create the regular expression an iteration from A to C can be made (in the same order as they occur in the left-most column) and each transition *to* the current node is interpreted as the start of a repetitive pattern, and each transition *from* the current node (except for the transition to the consecutive node) is interpreted as the end of a repetitive pattern.

This would in the example generate the regular expression $(A(BC)*)*$. This might seem like a fairly over-complex method of extracting the pattern, but the algorithm must take consideration to many exceptional cases. Recall that all D-nodes shouldn't necessarily be interpreted as a labeled D-node. It is possible that some node in the matrix doesn't match any of the labeled D-nodes, but they are still there since they match a tag-path with some other variable (say variable D) that resides in an other node in the original D-tree. If a node doesn't have any mappings it will cause an empty column in the matrix, and should thereby be discarded in the inference process.

This method doesn't hold for general purposes when generating regular expressions since it relies on several assumptions, but it is sufficient when considering typical repetitive patterns when dealing with product pages. It assumes that the whole collection of repetitive patterns is available in the pattern of the labeled nodes, and it also assumes there are no conjunctions or optionals. If conjunctions are found it would generate an incorrect expression.

5.3.5 Creating the M-Tree

The convertion from the I-tree to a complete M-tree is made with a depth-first recursion. The function starts at the D-nodes and erase ambiguity by inference until each labeled D-node has been inspected in regards to its context, an thus the entire tree contains no ambiguous interpretations. The steps of converting the I-tree from figure 5.5 is illustrated in figure 5.8 to 5.12.

In the first step it is recognized that the D-nodes B, C and D are ambiguous, but since each D-node is the single child of that parent (the `TD`-trees) no inference is necessary as there can only be one interpretation. In the second step each `TD` node is still ambiguous and therefor a new inference process is conducted. This time a pattern is discovered and the tree-nodes are illustrated as a paranthesis-looking lines indicating that a structural order is defined. The tree containing C *must* be strictly followed by the tree containing D. This process is continued until the M-tree is fully defined, in this case until the `BODY`-tag is found. The paranthesis-looking lines are supposed to illustrate an iterating pattern and should be considered as a Kleene-star when interpreting the tree as a regular expression.

5.3.6 Extraction

When the M-tree is applied to some other input in order to extract the content, the input is first parsed into a new D-tree. The corresponding I-tree from the original document is used to create an intersection between the I-tree and the new D-tree, and thereby removing nodes that per definition can't contain desired information. The M-tree is then applied to the D-tree in a recursive fashion with a greedy approach in order to match as many iterative patterns as possible. Since data elements can be nested in the fashion illustrated in figure 5.1, a compilation of the data is necessary in each recursive step. The recursive extraction-function returns a table of data from each call, and when consecutive calls are made in the same step the data is merged into a table where all columns contain the same number of elements. When two tables of data are merged, it is secured that each column that contains at least one element but less then the largest column, the last element in the smaller column is duplicated until each column contains equal number of elements. This merging process is motivated by the fact that tables in that fashion are interpreted from left to right and top to bottom, which is most common in european web pages. This method works in similarity with the data alignment in DeLa (section 3.6.3, figures 3.3 and 3.4), but it also successfully handles situations when multiple records are contained within the same node, as presented by DEPTA in section 3.6.7 (figure 3.5).

5.3.7 Normalization

The extraction process is based on a basic assumption that each data-element is contained within a tag-node. This assumption is far from obvious in the general area of data extraction, since properties like news content and book reviews often contain HTML code. This is a limitation of this algorithm but it is often sufficient enough since tag structures of product properties (prices, description, product-names) often are very plain. Even if each desired element is contained in a single tree node a normalization process is still necessary. This is vital in order to convert the extracted data into the correct format (dates, etc) and can be done by applying a manually written regular expression for each data type. It is also necessary since sometimes several elements are contained within the same tag, like the string “Book B by Author A” which contain both the author and the book title.

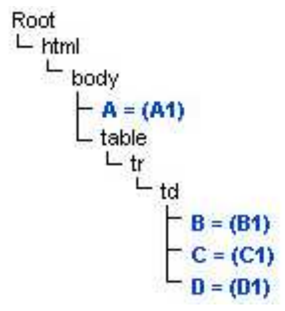


Figure 5.8: Step 1: Initial I-tree

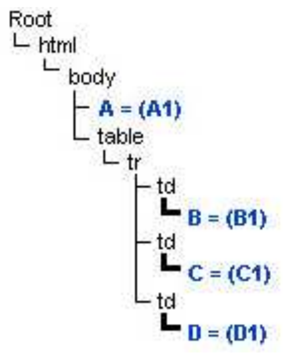


Figure 5.9: Step 2

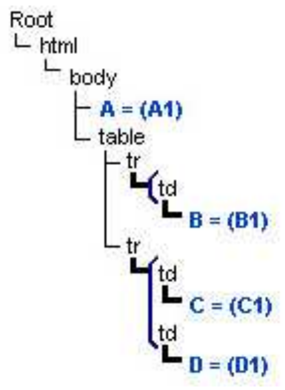


Figure 5.10: Step 3

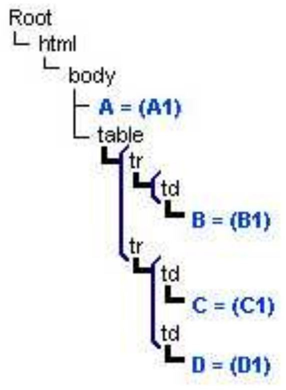


Figure 5.11: Step 4

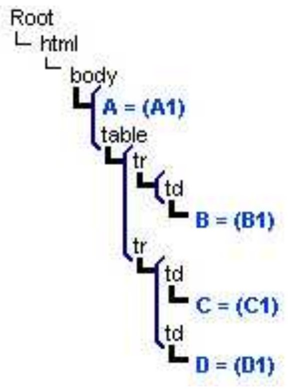


Figure 5.12: Step 5: The final M-tree

Chapter 6

Results

6.1 Usage of PIEME

The PIEME system has been tested for basic functionality. Web pages have been inserted into the system using the basic method of manually interpreting the documents with a proper sequence of operations. The system works and the data is extracted into the database, but it shows that the whole process is very tedious and complicated. It is hard work to design and write the proper regular expressions, and large regular expressions applied on large documents shows a very massive requirement of computing time. The non-surprising discovery is that this method scales terrible in regards to performance and human labour, but it works as a last resort.

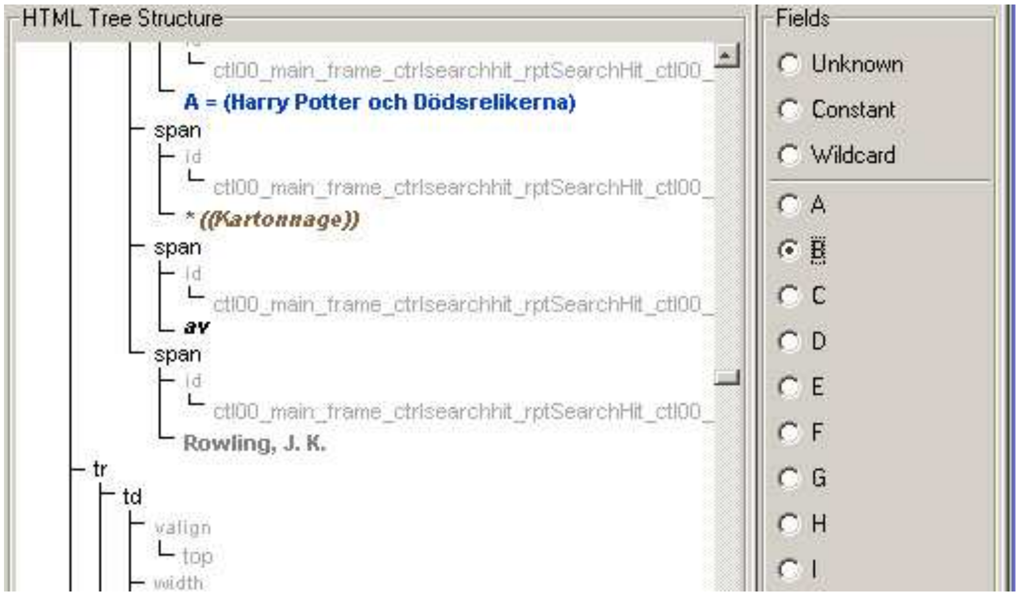
6.2 The Automated Extraction Algorithm

The algorithm for automatizing the extraction process has been tested independantly from the PIEME system. Intensive testing hasn't been proved necessary since the limitations of the algorithm is explained quite thoroughly in chapter 5, but testing have been conducted to indicate how these limitations affect the extraction process in real world situations.

The algorithm has been applied to several typical online stores displaying products, and the extraction works very well as long as the layout shows no disjunctional fields. Figure 6.1 shows a screenshot of a user labeling properties of a book from the online bookstore at `www.adlibris.se`. The resulting M-tree is displayed in figure 6.2 and was generated successfully without any trouble. The M-tree was applied on other pages from the same site and the data was extracted in good quality.

In regards of performance the inference and extraction process shows good results. The bottleneck is in fact the process of parsing the web page and generating the D-tree. Once the D-tree is generated the generation of M-tree and the extraction process completes in far less than a second. The parsing of the web documents hasn't been properly optimized in this implementation, as it contains many obscure regular expressions in order to clean up faulty HTML. The main focus of the implementation has been on the extraction algorithm, but it is quite confident that the code for parsing can be further optimized when comparing with the performance of standard web browsers of today.

The testing has however shown that the occurance of disjunctional fields is far more



HTML Tree Structure

```
      └ ctl00_main_frame_ctrlsearchhit_rptSearchHit_ctl00_
    └ A = (Harry Potter och Dödsrelikerna)
  ├ span
  │ ├ id
  │ │ └ ctl00_main_frame_ctrlsearchhit_rptSearchHit_ctl00_
  │ └ * ((Kartonnage))
  ├ span
  │ ├ id
  │ │ └ ctl00_main_frame_ctrlsearchhit_rptSearchHit_ctl00_
  │ └ av
  └ span
    ├ id
    │ └ ctl00_main_frame_ctrlsearchhit_rptSearchHit_ctl00_
    └ Rowling, J. K.
├ tr
  ├ td
    ├ valign
    │ └ top
    ├ width
```

Fields

Unknown / Constant / Wildcard

A [✓ B] C D E F G H I

Figure 6.1: The user labeling properties of a book at `www.adlibris.se`. The title is labeled as variable A, the book cover is labeled as a wildcard and the word "av" (of) is labeled as a constant since it helps to identify structure in the document.

frequent than orignially expected. The algorithm has been applied to the bookstore `www.amazon.com` which has a far more complex layout, and the generation of the M-tree proved far more difficult. The page layout require the user to define lots of constants or wildcards in order to identify the different elements, and they have different representations for the price of each book depending on whether the book can be shipped directly, has to be ordered or if it is on second-hand sale. Several trial and error testing had to be made in order to create the correct M-tree, and even after it was only able to extract a small portion of the books (2-3 out of 12) due to the disjunctional price field.

To test the functionality of extracting nested patterns in the form from figure 5.1 the algorithm was applied to the housing firm at `www.bostaden.umea.se`, where the apartment information is listed in several tables preceded by a label indicating the location of those apartments. The nested extraction works very well, but the algorithm has a poor requirement that the iterative pattern must occur within the labeled nodes, and the labeled nodes must be in the first occurance of that iterative pattern. This means that if the first occurance of a location-label is followed by a table that only contains one apartment, the iterative pattern won't be discovered, resulting in that only the first apartment from each location will be extracted. This site also contains optional fields (some apartments don't include electricity in the rent, for example) which causes problems for the learning process. The good aspect of the algorithm though is that it has no problem of dealing with optional fields as long as they are defined in the M-tree. In practice this means that if the first apartment in this example contains that optional remark on electricity, that field could be labeled as a wildcard and the rest of the apartments could thereby be extracted successfully. This works if the optional remark is just a D-node and not a tree structural option. The problem could be solved

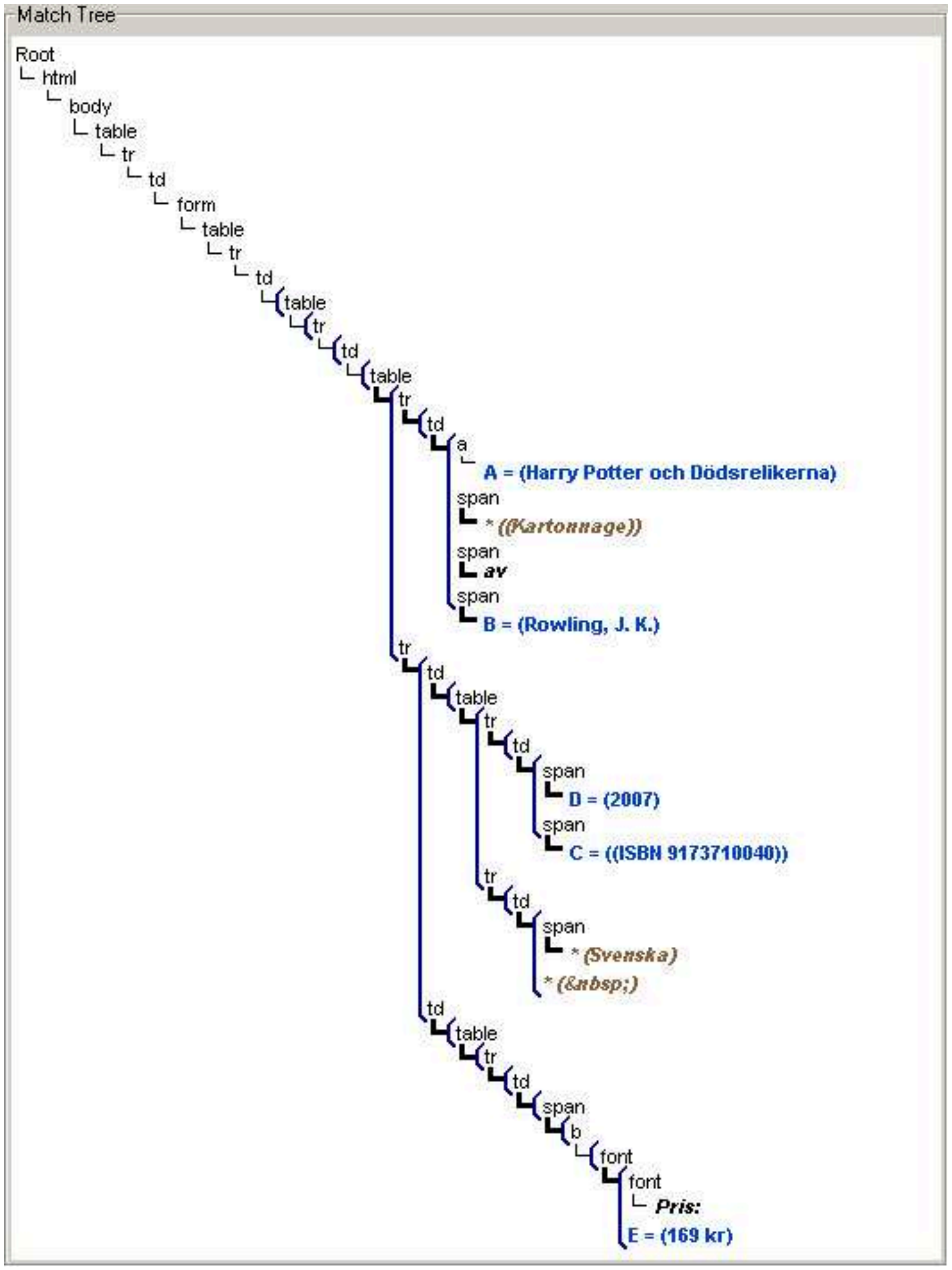


Figure 6.2: The generated M-tree from `www.adlibris.se`.

if a further implementation could give the user the possibility to insert empty D-nodes during the labeling process in order to correct the generation of the M-tree.

Chapter 7

Conclusions

It is quite clear that the algorithm suffers from severe limitations due to the lack of ability to handle disjunctional ways of representation. It is also concluded that it's impossible to create a system that is guarenteed to work on every site, since there will always be exceptions and special cases when dealing with product web sites. Even if a system could learn the patterns and create a correct extraction template for every page, there is still the possibility that disjunctional fields are embedded but not visible at the time of the learning process.

The algorithm works very well when extracting from uniform pages and tables, and it can actually handle documents with a large amount of varying non-interresting information as long as that information isn't interpreted as D-nodes.

In regards of the system in context it is now clear that the extraction part is just a small part of the problem. It has been revealed during the testing that the navigation is a far more complex issue than just following links specified in the `<A>`-tags. Many documents are packed with JavaScript code that overrides the reference and uses functions to perform HTTP-post requests with individually defined parameters. An example of this is that the tag might contain a line something like this: `href="javascript:openMyFunction('someParameter')"`. These function must sometimes be interpreted semantically (which in this case means *manually*) since the script code formats the parameters before they are sent to the server. The process of navigating and retrieving the documents has been proved to be as time consuming as writing wrapper programs manually. The evolvement of new technologies on the web makes the navigation far more complex, especially when considering Web 2.0 and Ajax-technologies. The representation of the information is also beginning to grow in complexity as many companies fill their data tables with scripts to add effects to their representation.

The problem of normalization will also be a problem that is hard to solve, even for a human. The quality of the extracted information will never be better than the source of the information.

As shown in the study there are several good tools and methods to extract information from the web if you are searching for something, but it's infeasible to make a fully automated extraction in order to present the information as a product. The need of human interaction will be necessary even with good tools that guides the hand, and as shown in the brief survey in section 3.1 this is the case on the market today.

The only way to really insure the extraction of correct information is to collaborate with the owners of the web sites, which means to work with people instead of computers.

The PIEME system is as a result of these conclusions at present state not sufficient for effective commercial use.

Chapter 8

Future work

8.1 The Algorithm

Due to the lack of possibility to deal with optionals and disjunctions the algorithm should be further improved to deal with these situations. One possibility is to make the user define which fields are optional and thereby apply more semantic clues to the inference function.

A serious limitation to the solution is as mentioned the presumption that each field share the same path from the root, which is not always the case. The path could be described with optional/disjunctional fields within the tag structure, but it would complicate the process of locating the fields tremendously. One way would be to define suffix path instead of prefix path, that is, the longest common path from the field to the root instead of the whole path from the root to the field. This would however include non-data fields in a much more thorough fashion.

8.2 Form Crawling and Page Navigation

It has been observed that the process of actually finding the documents is just as a complicated problem as parsing them for product information. There has been research conducted in the subject of navigation through the "Deep Web", which as concluded in this thesis is the natural next step towards the goal of an automated extraction process. [4] presents a technique of using components of the web browser Firefox[1] in combination with Lixto [5] to generate wrappers.

Sadly and with great resignation, the paper on Lixto was surprisingly not discovered during the in-depth study of this thesis until the final steps of writing the conclusions in this report. This is a great abortivation to this project since Lixto was presented at the VLDB conference as early as 2001, and the methods (if studied earlier) would probably had made great innovations to this project. The Lixto system uses a logic programming language for trees called *Elog*, which is a further development of Datalog, a query language for deductive databases. Lixto uses a GUI to display the web page and the user can then click on interresting regions to define logic rules, which is later used to extract the information.

[1]http://www.mozilla.com

The Lixto project [10] has in fact emerged into a commercial product that deals with the whole concept of product retrieval in the same fashion that was aimed for PIEME. Although Lixto doesn't use any self-learning principals, it has proved very efficient in defining wrapper functions since defining logic rules with a GUI is far more user friendly and effective than writing regular expressions. If PIEME was to be patched with these methods of logic languages (which is very expressive) instead of using standard regular expressions as a fundamental condition, the outcome would probably be more decent. The work on PIEME is not entirely in vain though since these concepts could be patched and implemented as operations in the PIEME programs. The reason for missing this branch of research during the study is because the focus was too concentrated on examining the fully automatic methods, which proved to be deficient for this purpose.

Chapter 9

Acknowledgments

I would like to thank my supervisor Michael Minock for his wise and down-to-earth advice regarding the realization of this project. I would also like to thank Johanna Högberg and Frank Drewes for their short but interresting discussion on tree-based grammar inference, as well as P-O Östberg and Anders Broberg regarding the discussions on web 2.0 and Ajax. A big thanks goes to Emanuel Dohi for his proofreading and friendly support.

References

[1] Xwrap: An xml-enabled wrapper construction system for web information sources. In *ICDE '00: Proceedings of the 16th International Conference on Data Engineering*, page 611, Washington, DC, USA, 2000. IEEE Computer Society.

[2] A fully automated object extraction system for the world wide web. In *ICDCS '01: Proceedings of the The 21st International Conference on Distributed Computing Systems*, page 361, Washington, DC, USA, 2001. IEEE Computer Society.

[3] Arvind Arasu, Hector Garcia-Molina, and Stanford University. Extracting structured data from web pages. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 337–348, New York, NY, USA, 2003. ACM Press.

[4] Robert Baumgartner, Michal Ceresna, and Gerald Ledermuller. Deepweb navigation in web data extraction. In *CIMCA '05: Proceedings of the International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce Vol-2 (CIMCA-IAWTIC'06)*, pages 698–703, Washington, DC, USA, 2005. IEEE Computer Society.

[5] Robert Baumgartner, Sergio Flesca, and Georg Gottlob. Visual web information extraction with lixto. In *The VLDB Journal*, pages 119–128, 2001.

[6] Chia-Hui Chang and Shao-Chen Lui. Iepad: information extraction based on pattern discovery. In *WWW '01: Proceedings of the 10th international conference on World Wide Web*, pages 681–688, New York, NY, USA, 2001. ACM Press.

[7] Valter Crescenzi, Giansalvatore Mecca, and Paolo Merialdo. Roadrunner: Towards automatic data extraction from large web sites. In *Proceedings of 27th International Conference on Very Large Data Bases*, pages 109–118, 2001.

[8] E.M. Gold. Language identification in the limit. *Information and Control*, 10:447–474, 1967.

[9] Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. New indices for text: Pat trees and pat arrays. pages 66–82, 1992.

[10] Georg Gottlob, Christoph Koch, Robert Baumgartner, Marcus Herzog, and Sergio Flesca. The lixto data extraction project: back and forth between theory and practice. In *PODS '04: Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–12, New York, NY, USA, 2004. ACM.

[11] Chun-Nan Hsu and Ming-Tzung Dung. Generating finite-state transducers for semi-structured data extraction from the web. *Inf. Syst.*, 23(9):521–538, 1998.

[12] Nickolas Kushmerick, Daniel S. Weld, and Robert B. Doorenbos. Wrapper induction for information extraction. In *Intl. Joint Conference on Artificial Intelligence (IJCAI)*, pages 729–737, 1997.

[13] Bing Liu, Robert Grossman, and Yanhong Zhai. Mining data records in web pages. In *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 601–606, New York, NY, USA, 2003. ACM Press.

[14] Donald R. Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.

[15] Ion Muslea, Steve Minton, and Craig Knoblock. A hierarchical approach to wrapper induction. In *AGENTS '99: Proceedings of the third annual conference on Autonomous Agents*, pages 190–197, New York, NY, USA, 1999. ACM Press.

[16] Justin Park and Denilson Barbosa. Adaptive record extraction from web pages. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 1335–1336, New York, NY, USA, 2007. ACM Press.

[17] Sriram Raghavan and Hector Garcia-Molina. Crawling the hidden web. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 129–138, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[18] D. C. Reis, P. B. Golgher, A. S. Silva, and A. F. Laender. Automatic web news extraction using tree edit distance. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 502–511, New York, NY, USA, 2004. ACM Press.

[19] Binyamin Rosenfeld, Ronen Feldman, and Yonatan Aumann. Structural extraction from visual layout of documents, 2002.

[20] Jiying Wang and Fred H. Lochovsky. Data extraction and label assignment for web databases. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 187–196, New York, NY, USA, 2003. ACM Press.

[21] Jiying Wang and Frederick H. Lochovsky. Data-rich section extraction from html pages. In *WISE '02: Proceedings of the 3rd International Conference on Web Information Systems Engineering*, pages 313–322, Washington, DC, USA, 2002. IEEE Computer Society.

[22] Yalin Wang and Jianying Hu. A machine learning based approach for table detection on the web. In *WWW '02: Proceedings of the 11th international conference on World Wide Web*, pages 242–250, New York, NY, USA, 2002. ACM Press.

[23] Peter Willett. Recent trends in hierarchic document clustering: a critical review. *Inf. Process. Manage.*, 24(5):577–597, 1988.

[24] Yanhong Zhai and Bing Liu. Web data extraction based on partial tree alignment. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 76–85, New York, NY, USA, 2005. ACM Press.