Extracting Content for News Web Pages based on DOM

Hua Geng, Qiang Gao, and Jingui Pan
State Key Laboratory for Novel Software Technology
Nanjing University, Nanjing, P.R.China

Summary*

Nowadays, RSS is becoming a hot topic for Web applications. A lot of famous Web sites have provided RSS for users. However, making RSS files manually is boring, and so far, most sites haven't provided such a service. In this paper, we mainly describe the design, implementation and evaluation of HTML2RSS, a system to extract content from HTML Web pages based on DOM structure, and generate RSS files automatically with the extracted content. We introduce two algorithms to extract information from semi-structured Web data. The goal of HTML2RSS is to provide users with RSS files as a substitute of the HTML pages.

Keywords

Web information extracting, DOM, XML, time pattern, RSS

1. Introduction

With the huge amount of data available online, the World Wide Web has become the most popular and important way for people to obtain information. However, due to the complexity and hugeness of the WWW, the data on WWW is semi-structured and heterogeneous. Thus, mining useful information from Web is always a difficult and exciting challenge for researchers.

Much recent works have focused on extracting and mining useful information from semi-structured Web data. Hammer [4] describes extracting weather data form various WWW sites and converting the extracted information into database objects. Lin [7] first partitions a page into several content blocks according to HTML tag <TABLE>, and then uses entropy-based approach to discover informative blocks. Cai [2] advances VIPS algorithm, which extracts semantic structure of Web pages based on vision representation. By analyzing the hyperlink structure of the Web pages, two best-known algorithms, HITS [6] and PageRank [1], are proposed to rank Web documents. Some later works on topic distillation [3] try to analyze on HTML DOM (Document Object Model) structure. SoftMealy described in [5] is a well-known information extraction system that extracts the structural information from HTML documents based on manually generated templates. As the flexibility of HTML syntax and the difficulty of information extraction from HTML pages, XML is introduced. However, as we have observed, most of the web pages are written in HTML rather than XML.

RSS, which stands for RDF Site Summary, Rich Site Summary, or Really Simple Syndication, is an XML-based format that allows web developers to describe and syndicate web site content [9]. RSS is published in feeds or channels, and is read with a new category of software called news aggregators or readers. It is now becoming an efficient way to distribute information for publishers, and to obtain information for plain readers.

In this paper, we propose two algorithms (*SAMR* and *ATP*) to extract information for news Web pages based on DOM, and describe HTML2RSS, a system for generating RSS files from HTML files, using the two algorithms mentioned above.

2. Algorithms

Because of the flexibility of HTML syntax, a lot of web pages do not strictly obey the W3C HTML specification [10], which may cause mistakes in DOM tree structure. As a result, we do precleaning on HTML pages first of all, with a HTML parser. For every HTML file, the parser analyzes it and corrects mistakes. A DOM tree is generated after parsing, and the subsequent work is performed on it.

Given a news Web page, we abstract its structure into three levels: *page*, *subject*, and *item*.

Definition 1. An *item* α is a data object, which contains related information about a piece of news, including *item title* (the title of the news, key information), *item date* (the release time of the news), *item link* (the URL of the news), and *item description* (the brief description of the news). It can be represented by a set α = {*item title*, *item date*, *item*

*This work is supported by the National Natural Science Foundation of China under Grant Nos.60473113, 60533080

link, *item description*}.

Definition 2. A *subject* β is a data object, which contains related information about the current subject, including *subject title* (the title of the subject) and a set of *item*(s). It can be formulized as β = {*subject title*, *I*}, where *I* = { α^1, α^2,…, α^N} is a finite set of *items*.

Definition 3. Given a news page γ, *page* is composed by all the news information in the page, including *page link* (the URL of the page, key information), *page title* (the title of the document), *page date* (the last update time), *page encoding* (the encoding set of the document) and a set of *subject*(s). It can be formulized as γ = {*page link*, *page title*, *page date*, *page encoding*, *S*}, where *S* = { β^1, β^2,…, β^N} is a finite set of *subjects*.

2.1 Mapping Rule

By analysis on large amount of Web pages, we have found the following facts.

- The content of most pages on Web updates frequently, especially for news pages and BLOG (Web Log) pages.
- The structure of a page on Web usually doesn't change. That is to say, the DOM structures of most pages on Web are fixed.

Hence, for a specified news page, we first create a Mapping Rule, and then use it to extract information for creating the RSS file. Thus, the total process to create RSS using Mapping Rules can be divided into two steps.

Step.1. Pre-clean the page, and get a DOM tree. Create a Mapping Rule based on the DOM tree.

Step.2. Extract information from the DOM tree, using the Mapping Rule. Then create the RSS file with the information.

When the page is updated, we needn't modify the Mapping Rule, but only need to perform the Step 2. As we need manual help in the Step 1, we call this method *SAMR* (**S**emi**A**utomatic-extraction based on **M**apping **R**ules).

2.1.1 Rule Format

As we can see in Figure.1, the Mapping Rule is an XML file, which specifies from which nodes of the DOM tree the information to be extracted, and which part of RSS the extracted information to be converted into.

All the occurrences of the word *path* in Mapping Rule actually are given in *XPath* (XML Path Language), which is a language for addressing parts in XML documents. But different from W3C Recommendation [11], we simplify it in our system as follows.

Definition 4. *XPath* is a path used for addressing elements in an XML document, supporting the following syntax only.

- / Selects the document root
- *para[i]* Selects the *(i+1)*th *para* child of the context node, (*i* = 0,1,2…)
- *para* Selects all the *para* children of the context node
- *path/para[i]* Selects the *(i+1)*th *para* child of the node specified by the *path* (*path* here is an *XPath*)
- *path/para* Selects the all the *para* children of the node specified by the *path* (*path* here is an *XPath*)

Thus, the XPath */HTML[0]/BODY[0]/TABLE* selects all the *TABLE* children of the first *BODY* of the first *HTML* under the root.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<page>
  <link>page URL</link>
  <date>path of page date</date>
  <encoding>page encoding set</encoding>
  <subject path="common path of all elements of the subject">
      <title>relative path of subject title</title>
      <item path="common path of all elements of the item">
        <title>relative path of item title</title>
        <date> relative path of item date</date>
        <link> relative path of item link</link>
        <description> relative path of item description</description>
      ……..
  </subject>
  <subject path="common path of all elements of the subject">
    ……..
  </subject>
  ……..
</page>
```

Figure.1 Form of Mapping Rule

2.1.2 Rule Generating

In order to generate Mapping Rules, we designed a semiautomatic method, which needs manual help. A GUI–based tool is developed to help the users[1] create Mapping Rules. With the tool, users can mark any HTML element (such as text and link) on Web pages, and assign it to any Mapping Rule element (such as *page date*, *subject title*, etc. except *page link*, *page title* and *page encoding* as the system will extract them automatically). We define *mark* as a binary mapping relation.

Definition 5. For a specified page, *mark* is a binary relation between the set of HTML elements named *A* and the set of Mapping Rule elements named *B*. If we define *T* as the set of all texts on the page, and *L* as the set of all links, then

$A = T \cup L$

B = {page date, subject title, item title, item date, item link, item description}

The Mapping Rule actually describes mapping relations between elements of the above two sets with XPath. We have designed a rule merging mechanism, so that the users needn't mark all the news in a page. As Figure 2 shows, in order to extract all the news under the subject title "BUSINESS", we only need to mark two *items*, says the first one and the second one. The absolute XPath of the first *item title* is

[1] Here the word "users" refers to the users of the GUI tool who create Mapping Rules, not the ultimate users.

*/HTML[0]/BODY[0]/TABLE[0]/TR[0]/DIV[1]/***A[0]***/TEXT[0]*, while the XPath of the second one is */HTML[0]/BODY[0]/TABLE[0]/TR[0]/DIV[1]/***A[1]***/TEXT[0]*. The system will merge the two paths into */HTML[0]/BODY[0]/TABLE[0]/TR[0]/DIV[1]/***A***/TEXT[0]* as the new *item title* path. The paths of *item date*, *item link*, and *item description* are processed in the same way. Thus all the news under "BUSINESS" will be extracted. Similarly, to extract all news in Figure.2, marking two *subjects* is enough.

Figure.2 An example of Marking

2.2 Time Pattern

On news Web pages, a news item is often published with the corresponding release time (Figure.3). This feature is a prominent and useful clue for locating and extracting the target news items. With this characteristic, we design an algorithm called *ATP* (**A**utomatic-extraction algorithm based on **T**ime **P**attern discovery).

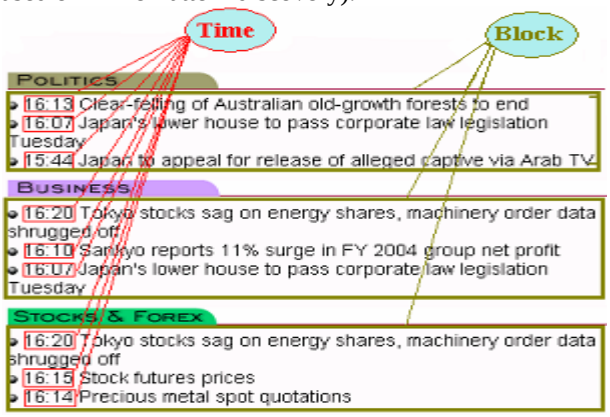

Figure.3 Time Information on News Pages

Since the formats of date and time are simple and limited, we can easily construct a database called Time Pattern DB for storing and managing *time patterns*, which are abstract representations of time formats (Table.1).

Table.1 : Some Typical Time Formats and the Corresponding Time Patterns

Time Format	Time Pattern
2004-07-09 21:08:03	*yyyy-mm-dd hh:ff:ss*
28 日 18：46	*dd 日 hh：ff*
Mar 15 2005	*mmm dd yyyy*
2005-March	*yyy-mmmm*
12 時 23 分	*hh 時 ff 分*
2005 年 5 月 15 日	*yyyy 年 mm 月 dd 日*

2.2.1 Discovery of Time Nodes

We use regular expressions to discover text nodes matching time patterns specified in Time Pattern DB (we call these nodes *time nodes*), when pre-order traversing the DOM tree.

We first generate a regular expression for each time pattern, e.g. *((0?[1-9]|1[0-2])/(0?[1-9]|[1-2][0-9]|3[0-1])* for *mm/dd*. For every text node, we try to match its text with the longest time pattern. That is, for text *2005-04-06*, we'll match it with time pattern *yyyy-mm-dd*, neither *mm-dd* nor *yyyy-mm*. After traversing the tree, we group the nodes matching the same time pattern. Sometimes there are multiple time patterns in a page, and we can select all groups, or just one group of a certain pattern selected by a heuristic rule. To refine the result, you can also specify certain time patterns. Thus, all nodes matching the specified patterns are selected.

2.2.2 Block-Dividing of Time Nodes

Before further extraction, we group the time nodes by their addresses. For easily processing, we use numbers to represent the addresses instead of *XPath*. The address consists of numbers joined by a dot, starting with '0', and followed by the order (index of the node in its parent's children nodes list) of the nodes which are ancestors of the current node. As Figure.4 shows, the address array can be divided into three blocks.

```
Block-Dividing (addrVect, n) {
  if(addrVect.size() == 0) return;
  if (addrVect.size() == 1) {
    blockVect.push(addrVect);
    return; }
  for(;n<MAXLEN; n++) {
    int count = GetDiffCount(n);
    if(n == 1) continue;
    if(n == addrVect.size()) {
      blockVect.push(addrVect);
      return; }
    //dividing otherwise
    int offset = addrVect.at(0)[n];
    vector newVect;
    for(int j=0; j<addrVect.size();) {
      if(offset == addrVect.at(j)[n]) {
        newVect.push(addrVect.at(j));
        addrVect.erase(j); }
      else {
        Block-Dividing (newVect, n+1);
        NewVect.clear();
        j = 0; }
      }
      Block-Dividing(addrVect, n+1);
    }
}
```

Figure.5 Block-Dividing algorithm (on Left)

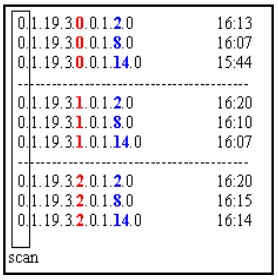

Figure.4 the Address Array of time nodes in Figure.3

Function *GetDiffCount(n)* computes the count of the different values in the column *n* of the address array. Only if the count is larger than 1 and smaller than the number of the addresses, we divide the address array.

2.2.3 Extraction from Time Nodes

For each block, news items in it often belong to a same *subject* (Figure.3). As a result, before extracting news items in a block, we discover the subject title of the block first. The pseudocode in Figure.6 describes the algorithm.

```
News-Extracting (blockVect) {
  for each block in blockVect
  {
    SubTitle = FindSubject(CurrentBlock);
    for each time node in CurrentBlock
    {
      ItemDate = GetText(TimeNode);
      UpBounds = FindUpBound(TimeNode);
      DownBounds = FindUpBound(nextTimeNode);
      for each node between UpBounds and DownBounds:
      {
        ItemTitle = ItemLink = ItemDes = "";
        if(tag name of Node is "a") {
          if(ItemTitle is not empty) {
            AddItem(SubTitle, ItemDate, ItemTitle, ItemLink, ItemDes);
            ItemTitle = ItemLink = ItemDes = ""; }
          ItemLink = GetLink(Node);
          ItemTitle += GetText(Node); }
        if(Node is a text node) ItemDes += GetText(Node);
      }
      if(ItemTitle is not empty)
        AddItem(SubTitle, ItemDate, ItemTitle, ItemLink, ItemDes);
    }
  }
}
```

Figure.6 News-Extracting algorithm

3. Implementation

3.1 Feed Server

We constructed a feed server with Apache Tomcat, on Red Hat Linux platform. In the database, we stored all the URLs and the related information including Update Frequency, Last Update Time, Update Algorithm (Mapping Rule or Time Pattern), Rule File (the Mapping Rule content) and RSS File (the RSS file content).

To help the ultimate users access RSS files conveniently, we wrote a JSP program, which fetches and analyzes users' requests, then queries in the database to get the required RSS files and returns to users. The user only needs to open his browser and type the request URL, which may like this, *http://AddressOfOurFeedServer/RssRequest?url=YourURL* .

3.2 Update Crawler

The crawler is a Java application, which updates the RSS files in the database constantly. It maintains a thread pool, in which the number of threads can be specified when the crawler is started. When it works, it collects all the URLs from database and assigns the same amount of URLs to each thread. Before a thread updates the RSS for a URL, it first compares the value of Last Update to the current system time. Only when the interval between them is larger than the Update Frequency, the RSS will be updated. As the module for extracting information and generating RSS was implemented in C, to use it in the crawler, we adopted a technique called JNI (Java Native Interface [8]).

In practice, we also applied some additional strategies to the crawler, in order to improve the system performance. For example, a fresh URL, which is newly inserted to the database, has a priority to those that have been in the database for a period of time when the crawler updates the database. For an advanced option, you can also designate how many threads to update RSS for fresh URLs, and how many for normal URLs.

4. Experiments

We conducted several experiments to measure and evaluate the performance of the HTML2RSS system.

4.1 Efficiency

We selected 125 news pages as the test set, in which there were 50 pages from Chinese news Web sites, 50 pages from Japanese news sites, and 25 pages from English sites, all with time information around the news items. Pages from English sites were much less because news is released without time information on most English news pages.

The experiments were performed on a PC with Intel P4 1.6G CPU and 512M RAM. The operating system was Microsoft Windows 2000. Firstly, we tested the efficiency of the two algorithms discussed in Chapter 2.

We inserted the URLs of the 125 pages, together with the rule files, into the database, and started the Update Crawler. As fresh URLs, they would be updated immediately. The following figure shows the test result. The abscissa indicates the number of the threads used for updating, and the ordinate denotes the total time (in second) for updating 125 pages.

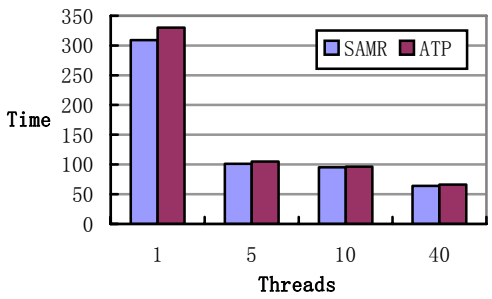

Figure.7 Efficiency Test for the Two Algorithms

Actually, for each page, downloading it from the Web site takes considerable time, and RSS generating usually takes less than one second. From Figure.7, we can see that if we run the crawler with multithread, the efficiency will increase greatly. But as the number of threads increases further more, the performance improves much less. We consider that it was due to the bandwidth limitation of the network.

4.2 Accuracy

As the evaluation of RSS files is subjective, to improve the reliability, we invited 30 persons to help us. Among them, there are college students, company staffers, and plain users who often browse news on Web. The participants were asked to browse both the HTML pages and the corresponding RSS files, and then give a mark for each RSS file (the mark ranges from 0 to 5, and the higher, the better) in their own opinions (Table.2).

Table.2 : the Average Marks by Users

Algorithm / Page	SAMR	ATP
Chinese (50)	4.71	4.60
Japanese (50)	4.62	4.62
English (25)	4.33	4.15
Total	4.60	4.52

Table.3 : the Average Occupancy of RSS

Algorithm / Page	SAMR	ATP
Chinese (50)	96%	85%
Japanese (50)	94%	88%
English (25)	90%	73%
Total	94%	84%

Table.3 : the Average Accuracy of RSS

Algorithm / Page	SAMR	ATP
Chinese (50)	96%	96%
Japanese (50)	91%	92%
English (25)	95%	95%
Total	94%	94%

We also computed the *occupancy* and *accuracy* of each RSS file (Table.3 and Table.4), by defining them as follows.

Definition 6. The *occupancy* of a RSS file is the ratio of useful news in the RSS file to all useful news contained in the Web page.

i.e. *occupancy = number of news items in RSS / number of all news items in page*

Definition 7. The *accuracy* of a RSS file is the ratio of useful news items in the RSS file to all items in the RSS file.

i.e. *accuracy = number of news items in RSS / number of all items in RSS*

Compared to *ATP*, although *SAMR* is more accurate and can be applied to almost any news Web page, it needs manual work during creating Mapping Rules, which will take much more time than RSS generating. Sometimes it's troublesome to mark on a page with complex structure. As a result, the two algorithms have their respective advantages and disadvantages. Choosing which algorithm for every news page is a problem.

5. Conclusion and Future Works

Now HTML2RSS has been put into practice and it performs well, according to the users' feedback. As Web pages have diverse contents and structures, HTML2RSS has its limitations and still needs further improvement.

At present it mainly deals with pages from news and BLOG sites, but the application fields can be extended as the system improved. The idea of time pattern discovery can be extended to mine other distinct format patterns, such as *currency patterns*, which can be used to extract information about products (such as the name, price and description of the products) from pages on e-commercial sites.

Besides *SAMR* and *ATP*, We have designed another algorithm *ARP* for extracting target information from Web pages (**A**utomatic-extraction based on **R**epeated tag **P**attern discovery) and now are testing it in our system. We will discuss the details of *ARP* in future.

References

[1] S. Brin, and L. Page, The Anatomy of a Large-Scale Hypertextual Web Search Engine, In *the 7th International World Wide Web Conference*, 1998.

[2] D. Cai, S.P. Yu, J.R. Wen and W.Y. Ma, Extracting Content Structure for Web Pages based on Visual Representation, In *the Fifth Asia Pacific Web Conference (APWeb2003)*, 2003.

[3] S. Chakrabarti, Integrating the Document Object Model with hyperlinks for enhanced topic distillation and information extraction, In *the 10th International World Wide Web Conference*, 2001.

[4] J. Hammer, H. Garcia-Molina, J. Cho, R. Aranha, and A. Crespo, Extracting Semistructured Information from the Web, In *Proceedings of the Workshop on Management fo Semistructured Data*, 1997, pp. 18-25.

[5] C.N. Hsu, and M.T. Dung, Generating Finite-state Transducers for Semi-structured Data Extraction from the Web, *Information Systems*, 23(8):521-538, 1998.

[6] J. Kleinberg, Authoritative Sources in a Hyperlinked Environment, *Journal of the ACM*, 46(5):604-632, 1999.
[7] S.H. Lin and J.M.Ho, Discovering Informative Content Blocks from Web Documents, In *Proceedings of ACM SIGKDD'02*, 2002.
[8] Sun Microsystems, Inc. *Java Native Interface*, http://java.sun.com/docs/books/tutorial/native1.1/.
[9] O'Reilly Media, Inc *What is RSS?* http://www.xml.com/pub/a/2002/12/18/dive-into-xml.html
[10] World Wide Web Consortium. *HTML 4.01 Specification*, http://www.w3.org/TR/1999/REC-html401-19991224, 1999.
[11] World Wide Web Consortium. *XML Path Language (XPath)*, http://www.w3.org/TR/xpath, 1999.