Example-Based Wrapper Generation†

Nitesh Shrestha, Ralph Busse, Gerald Huck
Fraunhofer Institute Integrated Publication and Information System (FHG-IPSI)
Darmstadt, Germany
{shrestha, busse, huck}@ipsi.fraunhofer.de

Abstract

Extracting specific information from the vast amount of documents in the World Wide Web is a very tedious task. Manual extraction has high quality output but cannot be automated. Programmed wrappers, on the other hand, suffer from the uncertainty of document structures. The generation of a more generic wrapper for whole classes of textual information, which can accommodate all kinds of document structures, is a crucial problem. Our graphical tool called the Intelligent Tagger allows user to create a grammar composed of rules and patterns which can parse plain text and html documents and retrieve desired information. Users are only required to have knowledge of the information type to be retrieved and its order and structure. With the Intelligent Tagger, grammar creation is performed in three steps: 1) a Graphical Schema Editor helps the user to create XMLSchema definitions with a visual drag and drop interface, 2) an Example Markup Tool allows to markup the desired information in a very simple way, and 3) a Grammar Generator takes the schema and the marked examples and generates a grammar for automatically extracting data from similarly structured documents. This paper focuses on this latter step.

1. Introduction

The increasing importance of the Internet as well as all the valuable information buried in legacy systems has brought considerable attention towards the generation of semi-automatic or automatic wrapper generation tools. These tools are used for generating suitable extraction scripts to extract the concrete information from World Wide Web pages and file system data. The desired data are not only extracted, but also presented in structured format by the wrapper tool, hence the extracted data can be queried further by standard query languages for many purposes, like, for example, event monitoring in news and stock markets or price comparison in various electronic markets.

Learning extraction patterns and composing them in well defined rules is the challenging problem of current wrapper generation techniques. Both manual information extraction and manual extraction rule creation suffer from two drawbacks: first, the effort may surpass the worth of information, and second, the task is likely to be error prone. Existing scripting languages like Perl or Jedi [1] are suitable layouts to write extraction scripts, but it is a rather complex and error prone task of manually creating textual scripts instructing them how to get the desired information out of the data sources. One of the objectives of this paper is to improve this task extensively.

Our wrapper generation approach is based on the observation that desired information from one data source shows commonalities in local and global structure. In this paper, we show a novel approach for a semi-automatic wrapper generation tool, which is called the Intelligent Tagger. Our idea is to use very simple markup and to generate extraction script (semi-) automatically. The Tagger tool is targeted for non-experts. It uses a simple underlying model and simple interaction paradigms, it allows for the reuse of controlled vocabulary (XMLSchemas), and it performs incremental generation, testing, and debugging of the wrappers. Machine learning techniques are applied to compensate lack of user expertise, to reduce or even eliminate the need for programming, and to achieve high wrapper quality and stability with minimal user interactions. As a consequence, the complete tool has been designed as a graphical tool where all actions can be done in an intuitive fashion. The required validation and extraction scripts are represented in a declarative language called Jedi, but the users are not required to have knowledge of that language.

The paper is structured as follows. Section 2 shows the system architecture and gives a brief description of first

† This work is partially funded by the CEC project XML-KM (IST 12030).
two phases of the grammar creation process. Section 3 is the main part of this paper and shows the pattern. Section 4 gives working examples, Section 5 shows related work carried out in this area, and Section 6 concludes the paper with a summary and an outlook.

2. Architecture

The architecture of the Intelligent Tagger system is depicted in Figure 1 and consists of three main components: the Schema Editor, the Example Markup Tool, and the Grammar Generator. In addition, the Jedi tool is used by the Grammar Generator to extract the desired information.

Our goal is to guide the user to a final parsing script with only minimum interaction and technical knowledge. As the first step, the user defines a so-called conceptual (XML) schema. This schema represents the output format and guides the extraction process. It defines the order in which elements should be expected and matched. Based on this schema, the Jedi parser tries to find the most specific assignment of text portions to terminal elements in the XMLSchema, i.e. text nodes. The result is a list of identified data elements together with a corresponding schema instantiation containing exactly these elements in the same order. From this point of view, the schema represents not only the output schema but also the input schema. Therefore, we call it more general the conceptual schema for the intelligent tagging process.

The idea behind this approach is that the user, instead of defining explicit matching rules simply selects those portions of the input document he is interested in and assigns each of them an element of a given conceptual schema. This markup must be done with several input documents in order to allow detecting commonalities and generic patterns. As a result of this markup, the example documents are extended with meta-information. A complete set of such extended documents is then given to the Grammar Generator for the final script generation. We distinguish between three types of markup: element, prefix, and suffix markup.

The element markup is the default markup type and in the majority of cases it should be the only one to be used. It maps the selected character content to a terminal element of the conceptual schema. The Intelligent Tagger generation, unification, and optimization process will then try to identify patterns for all marked elements and, as well, patterns for the intermediate separators.

In some cases, however, it might be helpful to give hints about keywords or specific structures that initiate or terminate a specific element. The prefix markup identifies the selected text as an initiating hint for a corresponding element; the suffix markup identifies it as a terminating hint. In a consistently marked document, each prefix markup must be followed by a corresponding element markup, and each suffix markup must be preceded by one.

Three types of nodes can be noticed here. The first one is the terminal node which captures user’s desired information. The second one is the indicator node which captures hint information (represented by prefix and suffix) for terminal nodes. The third one is the separator node to capture all remaining spare data.

The validation grammar generated from the conceptual XML Schema checks the correctness and completeness of markup must be followed by a corresponding element markup, and each suffix markup must be preceded by one.

Three types of nodes can be noticed here. The first one is the terminal node which captures user’s desired information. The second one is the indicator node which captures hint information (represented by prefix and suffix) for terminal nodes. The third one is the separator node to capture all remaining spare data.

The validation grammar generated from the conceptual XML Schema checks the correctness and completeness of
each document’s markup. It analyses every marked document and tries to find a schema instantiation for it. If it is not possible to find a schema instantiation then the extended document has to be rejected. Reasons are, for example, missing markup for mandatory elements or schema incompatibility due to wrong element order etc. When validation succeeds, all markup contained in that document can be annotated to the corresponding rules of the validation grammar. These annotations partition the complete document as each piece of text is either attached to terminals (element markup), indicators (prefix and suffix markup), or separators (unmarked parts). These annotated strings are the sources for detecting and generating the final extraction grammar.

The next section of the paper discusses the pattern generation approach from annotated strings.

3. Pattern Generation

The outcome of the example validation process is a tree of terminal, indicator, and separator nodes, with sets of example strings attached to them. In the generation process we convert each set of strings attached to a node into a pattern that generalizes all these strings. A pattern is defined as a sequence of characters, character classes, and repetition indicators. The original characters can be categorized as letter, digit, white space, punctuation and so on. These categories can be combined again to get the more general categories. All the strings can be represented in the form of these categorized characters (of different level) and then can be unified to get the single pattern string.

3.1 Pattern Elements

Each pattern consists of a sequence of three basic elements: characters, character classes, and repetitions. A single character in a pattern matches exactly this character in any input string. A character class matches any character belonging to a certain type. In our system, we use four basic character classes:

- LETTER: represents all alphabetical characters [a-zA-Z]
- DIGIT: represents all digits [0-9]
- WHITE: represents all white space characters [\t\n\r]
- PUNCT: represents punctuation and selected symbol characters [!"#$%&/()=?…]

By combining these four basic character classes, we have 15 different character classes representing different subsets of the complete character range. Every character class can be followed by a repetition indicator, specifying optional occurrence or unlimited repetition.

3.2 Distinguishing Different Contexts

The information attached to different types of nodes (i.e. terminal, indicator and separator) differs in perspective. Users are only interested in information belonging to terminal nodes. The contents of indicator and separator nodes are only used to identify terminal node contents in the bunch of input data. The strings attached to terminal nodes are assumed to be rather short and little specific. As a too tight pattern could result in information loss, we do not keep the original characters but start pattern generation directly from the sequence of character classes. The indicator nodes, on the other hand, serve as hints for the terminal nodes and have a high chance of holding equal content in different examples. We have to keep the pattern as specific as possible. In principle, the same is true for separator nodes, but they might be much longer than indicator nodes. As a consequence, we have to take care of complexity reduction for this type of nodes. In the following we detail the different pattern generation approaches.

3.2.1 Terminal Pattern Generation

The initial pattern is generated from the first example string by replacing all letters with character class LETTER and all digits with character class DIGIT. The rest is kept as it is. This initial pattern can have two character classes LETTER and DIGIT; their repetitions; and remaining original characters. For example, the input string “Premium·Vcr”, taken from an auction item description, is converted to the initial pattern “LETTER+ · LETTER+”. This represents a sequence of repeated letters followed by a single white space and repeated letters.

This initial pattern is the starting point for unification. In a subsequent loop every other input string for this terminal node is unified with this pattern. The resulting pattern will then match all input strings. The unification process itself is identical for all node types and is described in Section 3.3.

3.2.2 Separator Pattern Generation

The strings of this node type can be quite long and finding a general pattern like in the case of terminal nodes could end up with long repeated patterns which are little specific and time consuming in evaluation. Therefore, we try to combine specific and general pattern sections for optimum performance. The first idea is to keep the leading and trailing portions of the pattern as specific as possible in order to find good boundaries to the adjacent nodes. Furthermore, we assume that all strings attached to a separator node have some common contents. Therefore, we try to find the common longest sequence of tokens
(CLST) among them and preserve it in the final pattern. Each separator pattern consists thus of five sections: 1) a specific initial part, 2) a generic initial part, 3) a specific central part, 4) a generic trailing part, and 5) a specific trailing part.

The final pattern is generated in five steps:

1. We identify the common longest sequence of tokens CLST in all input strings. Due to lack of space we cannot present the algorithm within this paper.

2. We take the initial portions from all example strings (from the beginning up to the CLST) and apply the unification process to these substrings.

3. We perform the unification process with all trailing substrings.

4. We apply the pattern optimization techniques described in Section 3.4 to the separate patterns. The initial pattern is reduced from right to left in order to keep the left part as specific as possible. The trailing pattern is reduced from left to right to keep its end most specific.

5. The resulting partial patterns are re-combined to a single pattern matching the complete separator.

### 3.2.3 Indicator Pattern Generation

In order to keep these patterns rather specific, we try to preserve the original text characters as long as possible. In general, the strings attached with this node type are supposed to be rather short and more specific than terminal nodes, but they may as well become large. For this reason, we apply the same algorithm as for separator nodes. The CLST algorithm will guarantee a very specific content, and generalization will only take place when the pattern is getting too long.

### 3.2.4 Specific Treatment for Leading and Trailing Garbage

We have found that normally the contents (Leading Garbage) before the first Indicator/Terminal node and the contents (Trailing Garbage) after the last Indicator/Terminal node of the documents are quite long and treating them like separator nodes could cause the system to waste time in keeping unnecessary details. Only a limited amount of trailing characters is needed to distinguish the leading garbage from succeeding nodes. The opposite is true for the trailing garbage. On this basis, measures are taken to keep the length of the first and last nodes’ patterns below a defined constant $k$.

**Leading garbage:** If the length of strings attached to this node is more than $k$, only the $k$ trailing characters are kept for pattern creation. All leading characters are consumed by a .* pattern. The unification and optimization process is performed like for trailing portions of separator nodes, keeping the trailing parts as specific as possible.

**Trailing garbage:** The trailing garbage is processed like the leading garbage, but with different direction. I.e., the leading $k$ characters are preserved and unification follows the leading parts of separator nodes.

### 3.3 Pattern Unification

Pattern unification is always performed in a loop. The previous sections described the creation of initial patterns from the first example strings. Every pattern is now generalized by merging it step by step with the remaining example strings. The current pattern string $P$ is unified with the next example $N$ such that the unified pattern string subsumes both input pattern strings. This algorithm iterates character by character over both strings and tries to check whether pattern element $P[i]$ subsumes the next string character $N[j]$ or not.

If $P[i]$ subsumes $N[j]$ then two possibilities of pattern character $P[i]$ are considered: If $P[i]$ is an original text character, the values of $i$ and $j$ are incremented to get the subsequent character. Otherwise, $P[i]$ is a character class and should not only subsume the current character, but also all subsequent characters belonging to the same character class. Therefore, we have to increment the string counter $j$ until the first character not belonging to $P[i]$ is encountered. Whenever more than one string character has been skipped, the character class is made repeated (if not already) in order to reflect this.

If $P[i]$ does not subsume $N[j]$ the following three steps are performed:

i) Replace $P[i]$ by the character class resulting from merging $P[i]$ and $N[j]$. Merging is the function to get a higher character class by combining two characters or character classes.

ii) Compress pattern $P$

The new $P[i]$ must be of higher character class than the old $P[i]$ and may also subsume subsequent pattern elements $P[i++]$. All these pattern elements must be removed. Otherwise, $P[i]$ would already consume the characters which are supposed to be consumed by its subsequent pattern – possibly resulting in a parsing failure. If the pattern is compressed and $P[i]$ is not yet marked repeated it should be done so.

iii) Skip $N$

All subsequent characters of $N[j]$ are skipped, which are subsumed by $P[i]$, otherwise the final pattern strings might have equal pattern elements
appearing consecutively, resulting in inefficient pattern. If some N[j] is skipped then P[i] is made a repeated pattern (if it was not already).

This process is repeated until one of either P or N is finished. If P is finished before N then the smallest character class (optionally repeated) subsuming all remaining characters of N is appended to P. If N is finished before P then all remaining patterns of P are replaced by their smallest equivalent character class (optionally repeated). The following listing shows the algorithm in pseudo code.

```plaintext
pattern unifyPattern(pattern P, string N)
    i = 0, j = 0 //iterators for P and N
    while(i<length(P) and j<length(N))
        if(not subsumes(P[i],N[j]))
            P[i] = merge(P[i],N[j])
        if(subsumes(P[i],P[i+1]))
            make P[i] repeated
            remove all P[i+1]..P[i+n] that are
            subsumed by P[i]
        if (P[i] is original character) then j= j + 1
        else if(subsumes(P[i],N[j+1]) and P[i] is
            non-repeated class)
            make P[i] repeated
            while subsumes(P[i],N[j]) do j = j + 1
        i = i + 1
    end while
    if P is finished during the while loop
        find smallest class C subsuming all remaining
        characters of N and append repeated C to the
        end of P
    else if N is finished during the while loop
        find smallest class C subsuming all remaining
        patterns of P and replace all these patterns
        with a single repeated C
    return P
```

Listing 1: Algorithm for Pattern Unification

In the above example we have generated the initial pattern “LETTER+ ‘·’ LETTER+” from the input string “Premium-Vcr”. The result of unifying this pattern with another item description, “Blue-Ford·Mustang”, results in the following pattern: “LETTER+ ‘·’ LETTER+ (LETTER | WHITE)” . In this example, the initial pattern is finished after matching all three pattern elements (repeated letter, single white space, and repeated letter) with “Blue-Ford”. All remaining characters of the new string, i.e. “Mustang”, are now merged into a single repeated pattern element, matching both letter and white space.

3.4 Pattern Optimization

By keeping the patterns as specific as possible, the output of our unification process tends to be rather long and complicated. This may lead to performance problems during pattern matching and data extraction. In order to find a compromise between efficiency and specificity, we perform two kinds of optimization that try to optimize the patterns without losing much specificity (i.e. accuracy).

3.4.1 String Length Reduction

We have found that for some input sources, there are very long sequences of original characters resulting in long and inefficient patterns. In this case we reduce the amount of consecutive original characters. If more than a defined number i of consecutive original characters are found in the pattern, only the first i-2 characters are kept and the rest is reduced to one single repeated character class. This process continues until the overall pattern length is less than a defined constant k or all the pattern elements are scanned once from left to right.

3.4.2 Middle Portion Optimization

After string length optimization, if the pattern length is still more than k, then final optimization is done by dividing the pattern into three parts. The first part contains the leading (k/2 - 1) characters, the third part contains the trailing (k/2 - 1) characters, and the second part contains the remaining middle pattern elements. The contents of the second part are merged into one single repeated character class. The new reduced pattern contains the original first part, the new second part, and the original third part. This way, the leading and trailing portions of the pattern are still as specific as before while the complete pattern has been reduced.

The terminal node patterns do not need to go through string length reduction since their letters and digits have already been converted to character classes and it is very unlikely that they contain long sequences of pure punctuation characters. However, the middle portion optimization is applied to it. For separator and indicator nodes, the left and right side patterns of the CLST are optimized first by string length reduction and then by middle portion optimization, if necessary. The CLST itself, however, is optimized only through middle portion optimization method in order to keep its specificity. The leading and trailing garbage patterns of the document do not need further optimization since they have already been reduced in the pattern generation phase.

4. Intelligent Tagger Validation

We have tested the Intelligent Tagger successfully with pure ASCII data and various web sites like, for example, e-bay or the CIA fact book. At least two example files are required to generate the pattern. The number of required example files depends highly on the degree of difference in the contents of the nodes (mainly separator nodes). It can be observed that if more examples are given, then more general patterns are generated, and as a result
accuracy goes down. The extracted information is found 100 percent accurate if the prefix and suffix nodes are specified properly for each terminal node. Figure 2 shows screen shots of two input examples. The contents of example files are marked as terminal nodes and corresponding prefix nodes. The validation script, depicted in the left side of Figure 3, parses the marked example files and annotates the corresponding strings to the respective nodes of the grammar. After pattern generation and unification, the extraction script is generated the Jedi script shown in the right side of Figure 3. This Jedi script can be applied to all files with similar structure to get the output in XML format as in Figure 4.
5. Related Work

Extracting useful information mainly depends on the extraction rules. A lot of research has been done in the area of generating extraction rules and can be broadly categorized in three areas:

1. **Natural language processing** for information extraction is carried out in AutoSlog-TS [2] and Snowball [3]. The former creates dictionaries of extraction patterns using only untagged text while the later generates patterns and extracts tuples from plain-text documents. Another work in this stream is CRYSTAL [4], which processes input by a syntactic analyzer and semantic tagger. It uses a bottom-up covering algorithm that begins with the most specific rule to cover a seed instance, and then generalizes the rule by merging it with similar rules. Since these approaches use natural language processing methods for learning extraction rules, they are fundamentally different than ours.

2. **Semantic** meanings of input documents are interpreted in many wrapper generation approaches. By semantic meaning of input documents, we mean that these wrapper generation techniques are based on the meaning of HTML tags having information like font size to guess a page structure. Therefore, they work for web pages only.

   Stalker [5], Lixto [6], IPEAD [7], XWrap [8], and W4F [9] are based on semantics of source documents. Stalker generates extraction rules based on user-labeled training examples that are expressed as simple landmark grammars. Lixto has used a declarative language (Elog) for data extraction. IPEAD automatically discovers extraction rules from web pages and does not require users training examples. XWrap and W4F use supervised interactive wrapper generation methods. The former one uses a procedural rule system and provides limited expressive power for pattern definitions while the latter uses an SQL-like query language called HEL. All of these approaches, except IPEAD, generate extraction rules based on training examples, while IPEAD can automatically identify record boundaries by repeated pattern mining and multiple sequence alignment. The PAT tree data structure is used to realize repeated patterns. Though these works in some respect are similar to our providing visual support for training the system but significantly differs on interpreting source documents.

3. **Non-Semantic** approach like ours which is not based on semantics of input documents is WIEN [10] and NoDoSE [11]. WIEN uses inductive learning techniques to build a program that extracts information from a web page based on a set of pre-defined extractors. The NoDoSE approach is quite close to our approach, which uses prefix/postfix patterns and offers an interactive environment for semi-automatic extraction of data based on outlining of interesting regions in documents. Though NoDoSe provides visual support for user to train the system to learn the extraction rule, it is more complex than our approach.

6. Summary and Future Work

Our extraction rules are expressed in Jedi scripts which are very expressive and fault tolerant. With the help of the Intelligent Tagger, user can generate XMLSchema definitions with a visual drag-and-drop interface and mark samples of desired information in input documents. Then...
it is the task of the Intelligent Tagger to generate the validation grammar, unify the examples, find the patterns, and generate the final Jedi script. All these tasks are transparent to the user. The Intelligent Tagger has used quite a novel approach for pattern detection and extraction. By applying different mechanism to different kinds of text, it makes the final pattern more specific and efficient to extract highly accurate information.

In the future, we plan to improve the Intelligent Tagger in different directions. In order to better support tagging of more complex input sources with repeated and nested elements, we plan to add a third indicator node for infix patterns, identifying separators for repeated elements. Furthermore, indicator nodes should be specified not only for terminal nodes but also for non-terminals. Another planned extension is the support for negative examples: Any unwanted elements in the output can be marked as faulty, giving some feedback to the generation process. Another change concerns the unification algorithm. Investigations showed that our current algorithm depends on the processing order of the example strings. This shortcoming can be removed by treating all input strings in parallel. Another plan is the support for basic data types, like date, number, etc., with pre-defined extraction patterns.

7. References


