Charles University in Prague

Faculty of Mathematics and Physics

**MASTER THESIS**

Peter Kóša

# Structured Data Extraction from Unstructured Text

Department of Software Engineering

| | |
|---|---|
| Supervisor of the master thesis: | Mgr. Martin Nečaský Ph.D. |
| Study programme: | Computer Science |
| Specialization: | Software Systems |

Prague 2013

I dedicate this thesis to my girlfriend and my parents, for providing food, shelter and moral support during the time of writing. I would also like to express my gratitude to the supervisor, Martin Nečaský, for his input and time.

Název práce: Extrakcia štruktúrovaných dát z neštruktúrovaného textu

Autor: Bc. Peter Kóša

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: Mgr. Martin Nečaský Ph.D., Katedra softwarového inženýrství

Abstrakt: Posledných 20 rokov sa stále zvyšuje množstvo informácií na internete a v publikovaných textoch. Tieto informácie sú ale často v neštruktúrovanej podobe, čo spôsobuje rozličné problémy, ako napríklad nemožnosť efektívne vyhľadávať v rozsiahlych kolekciách textov (lekárske správy, inzeráty, atď.). Na prekonanie týchto problémov potrebujeme efektívne nástroje schopné texty počítačovo spracovať, vyextrahovať z nich dôležité informácie a následne ich v určitej podobe uchovať pre ďalšie použitie. Cieľom tejto práce je porovnať existujúce riešenia medzi sebou ako aj porovnať ich s riešením, ktoré vzniklo v rámci softwareového projektu SemJob. Projekt SemJob je zároveň podrobne predstavený a čitateľ tak získa informácie o jeho vnútornej štruktúre a použitých algoritmoch.

Klíčová slova: extrakcia štruktúrovaných dát, extrakčné pravidlá, ontológie, (semi)automatická indukcia wrapperov

Title: Structured Data Extraction from Unstructured Text

Author: Bc. Peter Kóša

Department: Department of Software Engineering

Supervisor: Mgr. Martin Nečaský, Ph.D., Department of Software Engineering

Abstract: In the last 20 years, there has been an ever-growing amount of information present on the Internet and in published texts. However, these information is often in a non-structured format and this causes various problems such as the inability to efficiently search in diverse collections of texts (medical reports, ads, etc.). To overcome these problems, we need efficient tools capable of automatic processing, extracting the important information and storing of these results in some form for later reuse. The purpose of this thesis is to compare existing solutions as well as to compare them with our solution, which was created in the scope of software project SemJob. The SemJob project is introduced and the reader can therefore obtain knowledge about its inner structure and workings.

Keywords: structured data extraction, extraction rules, ontologies, (semi)automatic wrapper induction

# Contents

# 1. Introduction

The amount of information freely available on the Internet is growing and this growth is unlikely to stop in the immediate future. Majority of the non-multimedia information is still provided in a plain text or HTML web page format. Even though there is fundamentally no problem with this format for human users of the Internet, the need for searching and categorization of published knowledge necessitated the creation of an Information Extraction (IE) research field.

Historically, the interest in the IE can be dated back to the year 1987, when this subject was discussed on the first Message Understanding Conference (MUC), funded by DARPA [3]. The focus of the IE research has been defined as [4]: "The identification and extraction of instances of a particular class of events or relationships in a natural language text and their transformation into a structured representation (e.g. a database)."

There are several possible approaches one can take to achieve the above-stated goal. In a very coarse division, we recognize different IE strategies for the IE from HTML (or generally any structured format such as XML) formatted text and from the plain text format.

The HTML extraction strategies focus mainly on identifying a structural pattern in the HTML tags present on the web page, which will then serve as an aid in extracting all the valuable data instances from the web page. Alternatively, a tool can rely on the delimiter role of the HTML tags and perform the extraction based solely on the markup structure.

On the other hand, the extraction strategies for a plain text document must rely on distinct tools — mainly natural language processing tools (NLP), regular expressions and/or some form of extraction ontologies.

Various approaches to the issue of data extraction provide different strengths and weaknesses such as the possible level of automation, the ease of use and the complexity of the objects obtainable via the given method of extraction. For this reason, many of the tools presented in this thesis combine several different strategies to optimize the recall and precision of their implementation. Some of these determinant features can be evaluated from the theoretical view point and qualitatively compared.

## 1.1 Motivation

This thesis is partially based on the work done for the software project SemJob. This project was designed to take a set of documents containing job description advertisements, parse these job descriptions and store them as RDF data [14] in a suitable database.

The project was successfully implemented and defended in front of a committee. Significant part of the project consisted of developing a way in which a user can configure the application to perform IE on a corpus of documents from a given domain with sufficient recall and precision. The implementation of this module was done without any prior knowledge of existing IE solutions and used only limited NLP capabilities.

After finishing the project, we were curious whether the task could have been handled better, and thus we began researching the state of art applications for the IE. The tools in this thesis were chosen with respect to what general approach they implement and they are categorized accordingly in the text.

This thesis aims to compare the existing solutions in the field of IE and to summarize the findings. We will take a look at several data extraction applications and frameworks dealing with this problematic. However, due to the high variance in approaches to the IE problematic, some of the analyzed tools cannot be directly compared to our solution, since they focus on slightly different areas in IE, but we will try to discuss all the similarities and differences.

## 1.2 Structure of the Thesis

Chapter 2 analyzes chosen set of existing solutions together with advantages, disadvantages and specific features of all of these projects. In the chapter 3 we present the solution implemented during the SemJob project with an in-depth description of technologies and approaches used. Chapter 4 contains a theoretical comparison of all the presented tools together with an analysis of some of the the individual tools' features and categories. The following chapter 5 supplements the comparison with actual tests performed using SemJob and several other tools. The last chapter concludes this thesis and summarizes all the findings. Furthermore, suggestions for follow-up work and research are presented.

# 2. Existing data extraction solutions

The process of extracting data from a source into a given output format can be characterized by creating a middle layer, commonly called wrapper. The wrapper is formally defined as [5]:

> Given a data source S containing a set of objects O, determine a mapping W which populates a repository R with the objects O. In addition to this, the mapping W must be able to perform this extraction on a source S', which is similar to the original source S.

The term "similar" in this definition does not have an absolute meaning and depends on the general approach used by a given tool. If a tool uses a data extraction algorithm based on extraction rules comparing the structure of input HTLM documents, then page A and page B displaying a different subset of a collection of items are similar and the tool must be able to extract data from both (since the structure of the page is the same, only data differ). However, if a tool uses ontology based approach, two entirelly different documents originating from the same domain are similar, as the ontological relations remain unchanged, independently of the actual document structure.

In the last two decades, there has been an increase in the quantity and quality of tools performing the IE on a text from various sources, ranging from web pages containing data listings from a database to a plain text document containing factual data. The different text sources condition which algorithms are viable for a particular data extraction task. In this chapter, we will try to present several of the most known tools used for the IE and to discuss their mutual features as well as their disparities.

## 2.1 Categorization

When approaching the problematic of the IE from text sources, there are several major ways to go. Each of these techniques has its own pros and cons, but they can still fit in one of the encapsulating categories. The question is what criteria can be used when comparing different tools for data extraction. Let us take a look at the several group features.

### Extraction rules language

First and foremost, all tools for the IE have to have a possibility of defining what to extract from the input text. This may be done in distinct ways. An important aspect to consider is whether the tool utilizes a standard language such as Perl, Java or XML, or whether it uses a language designed specifically for the purpose of characterizing a set of extraction rules. The standard format has an obvious advantage in its portability and the existence of various tools which can be used for editing the configuration files. The specialized languages developed solely for the purpose of describing extraction rules may provide greater expressive power and/or greater clarity and simplicity of markup.

## Wrapper induction

Creating extraction rules by hand comes with several liabilities. First of all, once the format of the source text changes, one has to manually update all the existing rules, which has been created before. It is not common for the data extraction tools to provide means for batch-editing the existing rules, therefore this task would most likely require the user to go through a vast number of rules and to update them in the text editor one by one. This repeating menial labor did pose a major inconvenience for some of the authors of the presented data extraction tools, who in turn tried to avoid it by either introducing an automatic wrapper induction, or not using the extraction rules in this form at all.

The automatic wrapper induction is usually done by providing a set of training documents (i.e. an input document with the structure corresponding to the real documents, from which should the data be eventually extracted) together with a set of corresponding sample outputs, which the user would expect to receive from the data extraction. As for the amount of manual labor included in this case, someone has to prepare the sample documents — but this task can be delegated to a person, who is not necessarily a member of the team of authors and does not have to have an understanding of what the extraction rules internally look like. Furthermore, there is a possibility that once this set of sample documents is created, it will be usable for far longer than the extraction rules themselves.

A different way of wrapper induction constitutes of the tool offering a set of automatically detected candidate record instances which could possibly be of an interest to the user and then prompting the user to mark the correct instances together with the desired fields to extract. However, this approach is of questionable usability, as it does not alleviate the burden of manual chore that must be repetitively performed after a source document change.

Alternatively, the induction can be done fully automatically by finding interesting data in the training documents (using some sort of heuristics and/or ontologies and concept models) and than generalizing the contexts in which the data was found. Nevertheless — as well as in the first case — someone has to prepare the correct answers for the sample document set, so the automatically detected data could be validated.

As with all decisions accompanying the design of the extraction tools, the wrapper induction has a setback. The tradeoff with the wrapper induction lies in a small (or nonexistent) possible customization of the extraction rules; the wrapper induction process works using a number of hard coded rules, therefore the resulting extraction rules will always be subordinate to these induction rules.

### Degree of automation

Does the tool require the user to write all the extraction rules by himself from scratch, or does it come with a set of default predefined extraction rules which are used in the extraction? Or maybe the tool can induce all the needed extraction rules by itself! Does it support the batch processing of a number of documents? Does the user have to input any information during the actual data extraction process? All of these questions contribute to the final usability and ease of use of a particular tool, having the same tradeoff as the previous feature - the more automation, the less customization; the less automation, the more work to be done by the user.

### Support for complex objects

Depending on the data extraction algorithm, a tool may be able to match data of various complexity. If the tool employs only simple regular expression extraction rules, it is most likely capable of extracting only single-instance data matching a single-slot pattern.

Single-instance means the extracted data fragments from the whole source text will be assigned to a single resulting object. This may pose a problem when trying to match complex text constructions or when there are several separate data record instances in a single source text document. Single-slot means one extraction rule will produce only one piece of knowledge per rule.

In contrast with the single-slot patterns are the multi-slot patterns, which can be commonly found in several tools using the natural language processing (NLP) and can extract more knowledge using one rule, such as "⟨subject⟩ was killed by ⟨object⟩" — this is made possible thanks to the part of speech tagging (POS tagging). The counterpart of the single-instance matching is the multi-instance matching, enabled by applying structured extraction rules such as the embedded catalog tree (ECT) used by STALKER [33]. In general, usage of these structured rules also contributes towards more complex (nested) result objects.

### Input format

The input format is a serious restriction when choosing a tool to use. Some of the tools were tailored specifically for a HTML source and are reliant on the HTML tags used in the document, using them for either the whole extraction process, or at least for the determination of what parts of the document belong to one data instance. Some of these tools even need to have specific HTML tags inserted into a web page for them to function property. These can be used when the user has a direct access to the source web page's code and an ability to modify it. The NLP tools require the input text to be in plain text format, with as grammatically correct sentences as possible, so the POS tagger have high precision rate.

It is worth noting, that many of the newer NLP tools can also make use of a HTML tags in the source document and will remove the unnecessary markup once they do not need it any more.

### Resilience and adaptivity

The resilience of a data extraction tool denote the ability of the given tool to react to changes in the input text. These changes can either mean the extraction rules need to be changed as well, if the resilience is low, or can be overcome by resilient and versatile design of these rules and/or extraction process. The change of the extraction rules can be further assisted by having a wrapper induction mechanism incorporated in the tool. In general, ontology based tools have the highest level of resilience, since they can be virtually independent of the source document structure. As always, there is an exception — some data is too constricted by the format used that even ontology-based tools will have problems with the change, e.g. date formats 2012/12/31 vs. 31.12.2012.

## 2.2 Extraction tools overview

### 2.2.1 Tools utilizing specific languages

This category contains tools which use a specific language for defining the extraction rules. Both presented tools are of older date and are simpler than tools from other categories. Nevertheless, the specific language could be abandoned in favor of standard XML without any harm to the actual expressiveness of the rules.

#### Minerva [6]

MINERVA is a formalism for writing wrappers for web sites and other textual data sources. Minerva does this by utilizing the benefits of a declarative, grammar-based approach, and a flexibility of a procedural programming. It enriches regular grammars with an explicit exception-handling mechanism, which is used to cope with irregularities of the web data. It uses a dynamic tokenization and a set of production rules; each of these rules representing one non-terminal symbol of the grammar.

Minerva has been used as a part of Araneus [7] system, together with a tool for document searching and restructuring called Editor for exception handling. Let us consider this source code taken from Wikipedia disambiguation page for keyword “data” [8]:

```
<p><b>Data</b> or <b>DATA</b> may also refer to:</p>
<ul>
<li><a href="/wiki/Data_(computing)" title="Data (computing)">...
<li><a href="/wiki/Data_(Euclid)" title="Data (Euclid)">...
<li><a href="/wiki/Data_(moth)" title="Data (moth)">...
<li><a href="/wiki/Data_(Star_Trek)" title="Data (Star Trek)">...
```

To extract the various uses of data from this web page using Minerva, one could define a small set of rules. Each string starting with the character $ denotes one non-terminal symbol of the grammar. The asterisks, plus signs and question marks have meanings as in regular expression patterns, same as plain text strings. The last non-terminal $TP has a special meaning and is used to insert the result of the matching into a database. Note that this example does not utilize the Editor exception handling, which could be performed if this matching failed.

```
PAGE AllMeanings
$AllMeanings  : *<ul> ( $OneMeaning )+ </ul> ;
$OneMeaning   : <li><a href="’$OneURL"’ title="’$OneTitle"’>*</li> ;
$OneURL       : *(?"’) ;
$OneTitle     : *(?"’>) ;
$TP : {
          $OneTitle           char(100)
          $OneURL             char(100)
}
END
```

Minerva can produce multi-instance and multi-slot patterns, however, it works only on the input in a form of HTML pages and relies very heavily on the fixed page structure.

**TSIMMIS [9] [10] [11]**

TSIMMIS (The Stanford IBM Manager of Multiple Information Sources) is a project aimed to provide tools for accessing multiple information sources in an integrated fashion and to ensure that the information obtained from these sources is consistent. Taking this into account, it is not a single tool for data extraction. However, it contains wrappers configurable through specification files using a sequence of commands defining extraction steps.

Each command needs to be in a form *[variables, source, pattern]* where *source* specifies the input text to match, *pattern* defines the actual extraction rules and *variables* is a non-empty list of variables for storing the match results. The text stored in *variables* during the extraction process is not final and can be used as an input for subsequent commands.

After all the extraction commands are evaluated, given variables are packaged into one self-describing object exchange model (OEM) object. For the sake of simplicity, let us take the same web page as in the previous example:

```
<p><b>Data</b> or <b>DATA</b> may also refer to:</p>
<ul>
<li><a href="/wiki/Data_(computing)" title="Data (computing)">...
<li><a href="/wiki/Data_(Euclid)" title="Data (Euclid)">...
<li><a href="/wiki/Data_(moth)" title="Data (moth)">...
<li><a href="/wiki/Data_(Star_Trek)" title="Data (Star Trek)">...
```

Now we can define rules accomplishing the same goal as before:

```
[
  ["root",
   "get('http://en.wikipedia.org/wiki/Data\_(disambiguation)')",
   "#"],
  ["_meanings_tmp",
   "root",
   "*<ul>#</ul>*"],
  ["one_meaning",
   "*<li>#</li>*",
   "#"],
  ["one_meaning_url,one_meaning_title",
   "one_meaning",
   "<a href=# title=#>*"]
]
```

The hash symbol (#) denotes that the whole content should be assigned as an output of this rule. The underscore (_) at the beginning of any variable means this variable has only a temporary function and will not be included in the resulting OEM object. After evaluating these commands, we would get the following output:

```
root  complex  {
      one_meaning  complex  {
           one_meaning_url  string  "/wiki/Data_(computing)"
           one_meaning_title  string "Data (computing)"
      }
      one_meaning  complex  { ...
}
```

TSIMMIS can produce multi-instance and multi-slot patterns but works only on input in form of HTML pages and also needs a meticulous knowledge of the page structure.

**WebOQL [12]**

WebOQL is a tool aimed to provide a framework for complex web data management. It contains features to abstract, model and afterward query the data from online documents. The query language is able to convert between different data structure formats.

WebOQL is parsing the input documents by a generic wrapper, producing a transient structure called a hypertree. The parsing itself is similar to building a document object model (DOM [15]) of the web page and therefore there are no custom rules used in this step.

The produced hypertree is more complex than the OEM structure used by other tools [9], but it still contains less redundant code that traditional schema-based languages. It can roughly be compared to a format of an intermediate code produced by a high-level compiler, which is then used for code optimizations.

Formally, the hypertree is a edge-labeled tree with two types of edges - the inner edges are used for structural information, whereas the outer edges are used as links to other objects (similar to hyperlinks in HTML).

Using the WebOQL query language, it is possible to formulate queries over the hypertree and to locate and extract data of interest from the tree into another structure such as table. If we wanted to use the example from previous paragraph and extract the data from it, we could do so by evaluating the following query:

```
SCAN
"http://en.wikipedia.org/wiki/Data\_(disambiguation)"
USING
. . .
<li>
<a href=FullUrl title=FullTitle> ANY </A>
</li>
```

```
GIVING
<H2>"All meanings"</H2>
{
<span>FullTitle: FullUrl</span>
}
END
```

The WebOQL query language is able to express feasible queries, i.e., queries of polynomial complexity. As for the expressive power of WebOQL, it can simulate all operations in nested relational algebra and can also compute transitive closure on an arbitrary binary relations. One of the more advanced constructs are the navigation patterns — next query obtains all internal ⟨H2⟩ elements from the hypertree branch for papers.html (all examples taken from [12])

```
        select [ x.Text ]
        from x in "papers.html" via ^*[Tag = "H2"]
```

### 2.2.2 HTML-aware tools

This group of tools contains implementations, which take advantage of the valid HTML structure in their extraction process. If the provided HTML is not valid, they use a set of heuristics to repair or remove any invalid markup. Some of the presented tools implement their our language for the rule definitions and some use processes similar to those of the tools from wrapper induction group, but the main focus of these tools stays on the rigid HTML structure.

#### W4F [16]

W4F (World Wide Web Wrapper Factory) is a toolkit for semi-automatically building lightweight wrappers. The user can build a wrapper by specifying an URL of a document, describing what information to extract from this document and finally establishing the target structure where the results should be stored.

The W4F will then fetch the document via HTTP, clean up the HTML markup (the aim is to achieve well-formed tags and document validity) and build a DOM tree. Afterwards, the DOM tree is processed using the rules defined in HTML Extraction Language (HEL); each HEL rule expresses a navigation path along the DOM tree and specifies, what kind of information should be collected and how they should be combined.

The last step is mapping the preprocessed information to an output structure for later use. The authors provide a sample for extracting information about a movie from a IMDB web page [17]:

```
EXTRACTION_RULES
{
    title = html.body->h1.txt, match/(.*?) [(]/;
    year = html.body->h1.txt, match/.*?[(]([0-9]+)[)]/;
    genres = html.body->td[i:0].a[*].txt
        WHERE html.body->td[i].b[0].txt = "Genre";
    cast = html.body->table[i:0].tr[j:*].td[0].txt,
                match/(\S+)\s(.*)/
        WHERE html.body->table[i].tr[0].td[0].txt =~ "Cast"
        AND html.body->table[i].tr[j].getNumberOf(td) = 3;
}
```

From the above example, one can already see that the HEL language is heavily dependent on the structure of the source web page, since any modification of the source page will significantly change the produced DOM tree, thus invalidating any paths in the current extraction rules, which rely on one structure and firm order of the elements.

However, W4F provides a wizard application to help create these rules, but they still need to be put in manually. In spite of this low adaptability, the HEL provides means for extracting complex multi-instance, multi-slot patterns (e.g. the fork operator, #, which enhances a rule to match a following rule to the same instance).

## XWRAP [18]

XWRAP takes different approach to the semi-automatic wrapper construction - it provides a library of building blocks together with a GUI, so the user can select suitable blocks for the IE task at hand. It also employs inductive learning algorithms that enrich this library by discovering new patterns from sample pages.

The wrappers are generated in two steps. In the first step, the user needs to identify (using the provided GUI) the information to be extracted from the preprocessed source document (the document is checked for HTML validity, all invalid tags are removed and a parsing tree is created). The result of this is a set of information extraction rules, which are then combined with the blocks from the XWRAP library, resulting in a executable wrapper application specifically designed for the one provided source.

This two-phase code generation has several advantages, such as the ability to incrementally tune the produced wrapper and a user-friendly facade, shielding the user from writing actual Java code. The authors provide several examples of extraction rules — below is a sample construct with two nested loops for extracting a total of 6 values:

```
<ST_extract>
  ST_extract(String ST_name[], String ST_val[][])
    <? XG-Iteration-XG "Start" ?>
    <loop> integer row_i = 3, 4
      <loop> integer col_j = 0, 1, 2
        <rule_exp>
          extract ST_val[row_i, col_j] =
            ˜TABLE[2].TR[row_i].TD[col_j].getStoken()
          where
            ˜TABLE[2].TR[1].TD[col_j].getStoken() = ST_name[col_j];
        </rule_exp>
      </loop>
    </loop>
    <? XG-Iteration-XG "Start" ?>
</ST_extract>
```

XWRAP provides a considerable set of constructs to extract complex multi-instance, multi-slot patterns from the web pages. It states adaptivity to small changes in the source page structure, since the identification of the important page sections utilizes several heuristics. However, it is unable to apply one generated wrapper to the same page after a large-scale rebuilding of the web page.

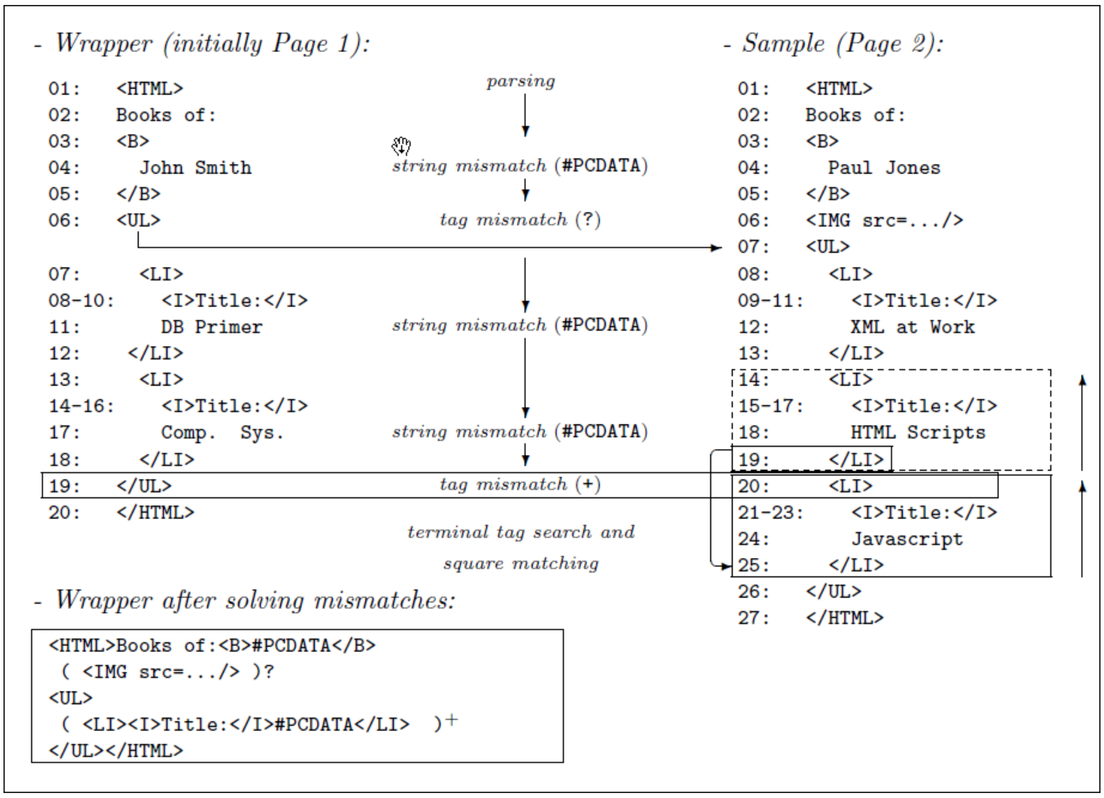
What stands out in the implementation of this tool is the overall ease of use, contributing to its real world feasibility. The user can inputs and debug all the rules using the provided GUI and XWRAP creates a tar archive with the finished wrapper afterwards. In spite of this advantage, XWRAP is still a tool with a significant strain on user due to the way the rule induction works. In its current state, it can only be used on smaller sets of web pages.

## RoadRunner [19]

RoadRunner is another tool making use of inherent HTML structure to perform wrapper generation and subsequent data extraction, thus trying to evade the need for a priori knowledge about the target pages.

The philosophy of RoadRunner is based on the notion that regular grammars cannot be inferred from the positive examples only [20]. Therefore the implementation tries to take an entirely different approach to the wrapper induction problematic by eliminating the need for user input (a set of user-provided sample pages) or knowing the page content before-hand. This is a step in the right direction when compared with the previous tools.

RoadRunner constructs the underlying dataset schema for the page data from the web page during the process of wrapper induction, utilizing any similarities and dissimilarities between two source HTML pages. These pages must originate from the same page class, which is defined as “produced by the same program/script”. The algorithm used for the schema construction is called ACME (Align, Collapse under Mismatch, and Extract) and it makes use of tag and string mismatches in the two web pages. The next figure is taken from the [19] and shows how finding a mismatch in the two pages helps with refining a resulting wrapper:


- Wrapper (initially Page 1):
- Sample (Page 2):
parsing
string mismatch (#PCDATA)
tag mismatch (?)
string mismatch (#PCDATA)
string mismatch (#PCDATA)
tag mismatch (+)
terminal tag search and square matching
- Wrapper after solving mismatches:

```
01:     <HTML>
02:     Books of:
03:     <B>
04:       John Smith
05:     </B>
06:     <UL>
07:       <LI>
08-10:      <I>Title:</I>
11:         DB Primer
12:      </LI>
13:       <LI>
14-16:      <I>Title:</I>
17:         Comp.  Sys.
18:       </LI>
19:     </UL>
20:     </HTML>
```

```
01:     <HTML>
02:     Books of:
03:     <B>
04:       Paul Jones
05:     </B>
06:     <IMG src=.../>
07:     <UL>
08:       <LI>
09-11:      <I>Title:</I>
12:         XML at Work
13:       </LI>
14:       <LI>
15-17:      <I>Title:</I>
18:         HTML Scripts
19:       </LI>
20:       <LI>
21-23:      <I>Title:</I>
24:         Javascript
25:       </LI>
26:     </UL>
27:     </HTML>
```

```
<HTML>Books of:<B>#PCDATA</B>
 ( <IMG src=.../> )?
<UL>
 ( <LI><I>Title:</I>#PCDATA</LI>  )+
</UL></HTML>
```

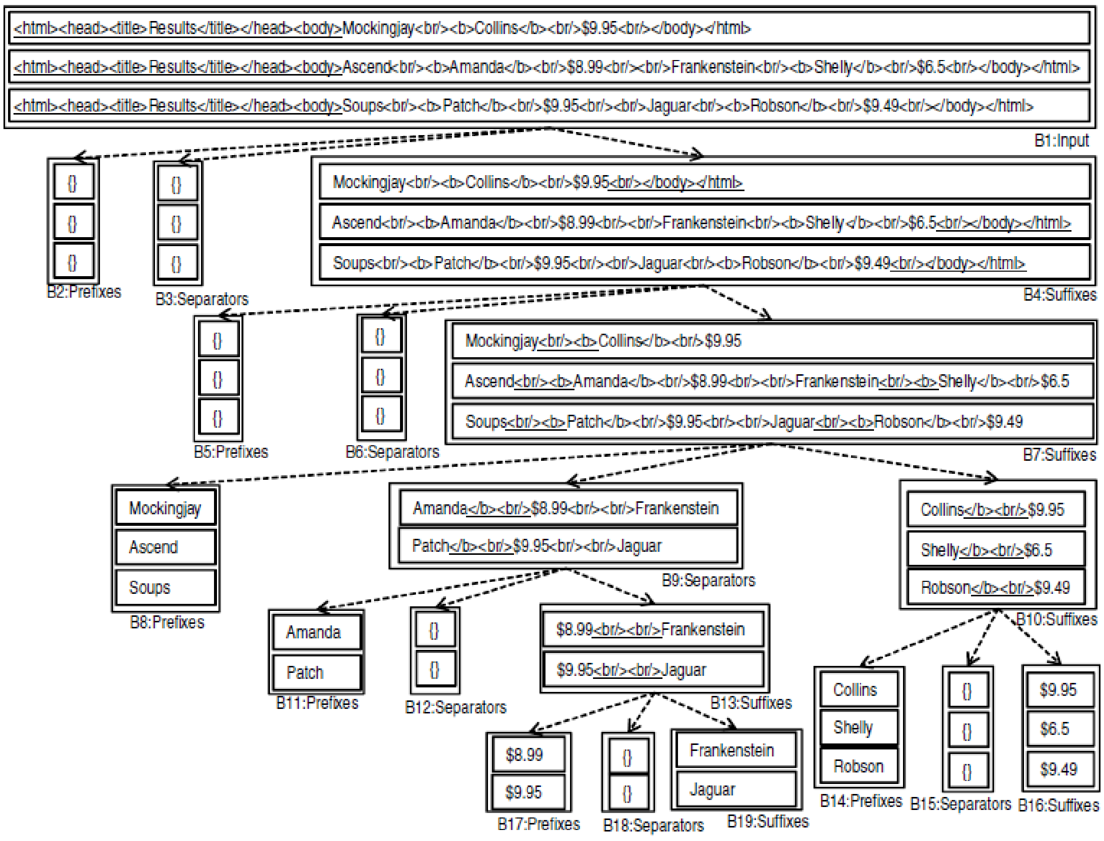
Figure 2.1: Using the mismatches in two pages to refine the induced wrapper

As depicted in the previous figure 2.1, all mismatches in the sample have been used to increase the quality of current wrapper. The mismatches on the lines 04, 11 and 17 were used to discover slots filled with instance data from the database behind the two web pages — these are the data that should be extracted after the wrapper is completed. The second mismatch, on the line 06, is a tag mismatch and is used to create an optional element ⟨IMG⟩ in the wrapper. The last mismatch on the line 19 introduces a repetition via the plus sign (+), as is it common in the regular expressions.

This advanced technique of wrapper induction is able to construct wrappers for complex multi-instance, multi-slot data extraction. What is more, the automatic nature of the tool makes it incredibly effortless to use.

**TEX [21]**

TEX is one of the latest tools in the field of web pages IE. It is designed to be fully automatic and therefore does not require any user interaction during the extraction process. Furthermore, since authors of this tool considered the observation that tools based on any set of extraction rules are less flexible and adaptive, they have chosen an approach to eliminate the need for these rules. The idea of the extraction algorithm used by TEX can be seen on the following image taken from their paper [22]:



<html><head><title>Results</title></head><body>Mockingjay<br/><b>Collins</b><br/>$9.95<br/></body></html>
<html><head><title>Results</title></head><body>Ascend<br/><b>Amanda</b><br/>$8.99<br/><br/>Frankenstein<br/><b>Shelly</b><br/>$6.5<br/></body></html>
<html><head><title>Results</title></head><body>Soups<br/><b>Patch</b><br/>$9.95<br/><br/>Jaguar<br/><b>Robson</b><br/>$9.49<br/></body></html>
B1:Input
B2:Prefixes
B3:Separators
Mockingjay<br/><b>Collins</b><br/>$9.95<br/></body></html>
Ascend<br/><b>Amanda</b><br/>$8.99<br/><br/>Frankenstein<br/><b>Shelly</b><br/>$6.5<br/></body></html>
Soups<br/><b>Patch</b><br/>$9.95<br/><br/>Jaguar<br/><b>Robson</b><br/>$9.49<br/></body></html>
B4:Suffixes
B5:Prefixes
B6:Separators
Mockingjay<br/><b>Collins</b><br/>$9.95
Ascend<br/><b>Amanda</b><br/>$8.99<br/><br/>Frankenstein<br/><b>Shelly</b><br/>$6.5
Soups<br/><b>Patch</b><br/>$9.95<br/><br/>Jaguar<br/><b>Robson</b><br/>$9.49
B7:Suffixes
Mockingjay
Ascend
Soups
B8:Prefixes
Amanda</b><br/>$8.99<br/><br/>Frankenstein
Patch</b><br/>$9.95<br/><br/>Jaguar
B9:Separators
Collins</b><br/>$9.95
Shelly</b><br/>$6.5
Robson</b><br/>$9.49
B10:Suffixes
Amanda
Patch
B11:Prefixes
B12:Separators
$8.99<br/><br/>Frankenstein
$9.95<br/><br/>Jaguar
B13:Suffixes
Collins
Shelly
Robson
B14:Prefixes
B15:Separators
$9.95
$6.5
$9.49
B16:Suffixes
$8.99
$9.95
B17:Prefixes
B18:Separators
Frankenstein
Jaguar
B19:Suffixes

Figure 2.2: The idea of data extraction from web pages used in TEX

The algorithm can be described as follows [22]:

> "The algorithm takes two or more web documents as input, and searches for shared patterns amongst them of size $s = max$ down to $s = min$, where $max \geq min \geq 1$.
> When a shared pattern *sp* is found, the text of each document is partitioned and three groups are created: prefixes, suffixes, and separators. Prefixes contain the text fragment from the beginning of each text until the start of the first occurrence of *sp* in this text; suffixes contain the text fragment from the end of the last occurrence of *sp* in each text to the end of this text, and separators include each separating text between every two consecutive occurrences of *sp* inside each text.
>
> Now that we have created three groups of text, the algorithm tries to search for a shared pattern of the the same size $s$ between the components of each group. If a group shares a string pattern, it is partitioned again; if not, $s$ is decreased, as long as $s \geq min$, and the algorithm starts again its shared pattern search on this group.
>
> When $s = min$ and no shared patterns are found, the proposal considers that the remaining non-empty text fragments inside each group can be considered as relevant text that should be extracted. The search for shared patterns is performed using a modified version of Knuth-Morris-Pratt's algorithm [23] in which all the occurrences of a string sequence are detected without overlapping."

Thanks to this design, the input documents do not have to be converted into DOM trees and therefore do not have to be XHTML compatible. According to the authors of this tool, it has been empirically proven on a body of approximately 2000 web documents from 55 real-world web sites that TEX can achieve mean precision as high as 96% and mean recall as high as 95%, with a mean execution time of 0.81 s. These numbers are quite astounding, considering the relative simplicity of the used algorithm.

To highlight the most convenient feature of TEX, the tool has very high level of automation — it was designed not to demand any user input throughout the whole process of extraction. The resulting ease of use coupled with the high recall and precision of this implementation represent an incredibly solid application.

### 2.2.3 NLP-based tools

This category covers the tools, which rely on a linguistic pre-processing of the input documents prior to the data extraction process. Due to this pre-processing step and the implicit complexity of the NLP problem, this tools are usually slower than their non-NLP counterparts. Despite this small flaw, the knowledge about the input text obtained from the NLP allows these tools to attempt to "understand" the input document (to some extent) and therefore empowers them to use algorithms, which would be otherwise unreachable.

Furthermore, some of the tools does not submit the while input file for the NLP processing, but perform this task only on parts of the input document selected by different heuristics, thus diminishing the performance bottleneck.

#### AutoSlog [24]

AutoSlog performs the data extraction on plain text documents and for this task it uses a dictionary of extraction patterns called concept nodes. This dictionary is constructed automatically from a set of training documents, consisting of example texts and associated answer keys for each document. The algorithm for dictionary building can be described in several steps:

- For a given answer key, find the first sentence containing this string
- Pass this sentence to lexical processor CIRCUS [25] for the conceptual analysis
- Identify the first clause in the sentence that contains the string
- Apply heuristics to the clause (see below)
- If the heuristics could not be satisfied, go to the first step of this algorithm
- Otherwise, generate a conceptual anchor point and a set of enabling conditions

The heuristics used in the algorithm consist of a predefined set of linguistic patterns (13 patterns are currently included). For example, if the algorithm received a text

```
the diplomat was kidnapped
```

and a keyword "diplomat", using the heuristic

```
<subject> passive-verb
```

AutoSlog would generate conceptual anchor point "diplomat" and enabling condition requiring a passive construction. Using additional knowledge supplied by the user in three domain specific configuration documents, AutoSlog would generate a following complete concept node

```
Name: victim-passive-verb-kidnapped
Trigger: kidnapped
Variable Slots: (victim (*S* 1))
Constraints: (class victim *S*)
Constant Slots: (type kidnapping)
Enabling Conditions ((passive))
```

However, the process of generating these concept nodes is not flawless, therefore a human input is required to approve or discard the proposed nodes. As seen in the example, AutoSlog can produce only single-instance single-slot rules and needs to have multiple rules, if the target of extraction can appear as different part of speech (e.g. target was kidnapped vs. kidnapped target).

**LIEP [26]**

LIEP (Learning Information Extraction Patterns) also creates a dictionary of IE patterns from a set of sample sentences and associated data to be extracted from each sentence. The extraction itself involves recognizing a group of specifically typed entities (usually noun phrases) and existing syntactic and semantic relationships between them.

This recognition is done by inputting a sample document to the LIEP tool, which then tries to use existing extraction rules on it. In case it does not succeed in covering all identified entities from the document using the existing rules, it will try to generalize a known pattern to accommodate for the new sample. If this effort fails as well, LIEP will construct a new pattern from scratch. To reuse the previous example, consider the sentence

```
the diplomat was kidnapped by terrorists
```

From this sentence, LIEP can generate a rule in the following form

```
TARGET-was-kidnapped-by-PERPETRATOR:
   noun-group(TRGT, head(isa(physical-target))
   noun-group(PERP, head(isa(perpetrator))
   verb-group(VG, type(passive), head(kidnapped))
   preposition(PREP, head(by))

   subject(TRGT, VG)
   post-verbal-prep(VG, PREP)
   prep-object(PREP, PERP)
==> kidnapping-event(KE, target(TRGT), agent(PERP))
```

As we can see, this extraction rule has multi-slot character, since both target and perpetrator will be extracted at the same time. LIEP calls this multi-slot extraction to be an extraction of event, as opposed to single-slot extraction of isolated facts. After using a sufficient number of sample documents, the extraction rules are adequately refined and the actual data extraction can be performed.

The extraction is done by a component called ODIE (On-Demand Information Extractor) which first tokenizes the text and then searches for possible occurrences of interesting keywords. If such keyword is found, the containing sentence is passed to the lexical analyzer for part of speech (POS) tagging [27].

If no keyword corresponding to an potentially interesting event is found, the whole sentence is discarded. After this step, the sentence is compared against a set of simple pattern matchers to identify all entities of interest (e.g. names of people, names of companies, noun groups, etc.) and finally, extraction rules are applied to the current preprocessed sentence.

If a match between the rule and the sentence is found, the event corresponding to the extraction rule will be logged. Finding a match incorporates verification of syntactic relationships between individual constituents participating in this match.

To save time and increase performance of the data extraction, this is done on-demand (thus the name ODIE) using a local syntactic constraints (for example, when considering a relationship "subject(noun-group, verb-group)", the noun-group must be to the left of the verb-group and the only tokens that may be between these 2 groups are right modifiers of the noun-group).

Authors of the tool claim, that during their experiments performed on a randomly selected sample from a corpus of 300 acquisitions-relates documents, LIEP has achieved recall and precision values ranging from 80 to 90 percent.

### PALKA [28]

PALKA (Parallel Automatic Linguistic Knowledge Acquisition) differs from the previous tools mainly in the structure of the extraction patterns. These are expressed as frame-phrasal pattern structures (FP-structures), each consisting of a meaning frame linked to a phrasal pattern.

The meaning frame represents an item, which should be extracted. This representation is done via a tree-like structure featuring a root and several slots for storing results of the extraction plus the semantic constraints for filler words. The phrasal pattern represents an ordered combination of lexical entities or concepts from a concept hierarchy together with a filler text. By linking the meaning frame slots to the phrasal pattern entities, one gets the final FP-structure, for example [28]:

```
Meaning frame:   (BOMBING
                     isa: (TERRORIST-ACTION)
                     keyword: (bomb grenade throw hurl ... )
                     agent: (ANIMATE)
                     target: (PHYSICAL-OBJ)
                     instrument: (PHYSICAL-OBJ)
                     effect: (STATE))

Phrasal pattern: ((BOMB) BE HURL AT (PHYSICAL-OBJ))
```

The final FP-structure is formed by connecting the “target” slot to pattern (PHYSICAL-OBJ) and the “instrument” slot pattern (BOMB):

```
FP-structure:    (BOMBING
                      target: PHYSICAL-OBJ
                      instrument: BOMB
                      pattern: ((instrument) BE HURL AT (target)))
```

The usage of semantic entities allows for transient matching through is_a relations (thus creating a structure similar to the common ontology structure), which leads to greater universality of the FP-structures. When creating a FP-structure dictionary, the semantic entities hierarchy can be used for generalization (moving up in the hierarchy; dynamite → explosive → bomb → instrument of crime → thing) or specialization (moving down). As seen from the example above, PALKA has multi-slot extraction rules.

## CRYSTAL [29] [30] [31]

CRYSTAL is a tool for building dictionaries of concept nodes (CN) from the free text. It utilizes the BADGER sentence analyzer, which takes the plain text document and produces a sequence of the CN. Each CN has an assigned CN definition specifying a set of syntactic and semantic constraints that must be satisfied for the definition to be applied.

A set of preprocessed and annotated training documents is passed as an input to the CRYSTAL tool, which then performs the following algorithm to induce as many new CN definitions as possible:

```
initialize the dictionary and training instances database
    for each initial CN definition in dictionary
        D := an initial CN definition
        loop
            D' := the most similar CN definition to D
            U  := the unification of D and D'
            test the coverage of U in training instances
            i  (error rate of U > tolerance) exit loop
            remove all CN definitions covered by U
            D  := U
        end loop
        add D to the dictionary
    end for each
return the resulting dictionary
```

The extracted CN definitions are in the form similar to the example below (taken from [29]):

```
CN-type: Diagnosis
Subtype: Pre-existing
Extract from Prep. Phrase "WITH"
Passive voice verb
Verb constraints:
    words include "DIAGNOSED"
Prep. Phrase constraints:
    preposition= "WITH"
    words include "RECURRENCE OF"
    modifier class <Body Part or Organ>
    head class <Disease or Syndrome>
```

After this algorithm finishes, the resulting CN-definition dictionary is tested for validity and can then by used for the actual data extraction.

When coupled with Webfoot, a tool for pre-processing web pages using cues from each page's layout, CRYSTAL is also capable of operating on web pages, which would be otherwise impossible. However, the CN definition dictionary obtained from plain texts differs from the one that is needed to process for web page input.

### WHISK [32]

WHISK is an extraction rule based system capable of extracting information from either structured or plain text documents. The extraction rules are in a form of regular expression with special placeholders for digits, numbers, etc. WHISK is capable of extracting either single- or multi-slot rules. A common example for WHISK is from the rental advertisement domain; each instance in the input must be enclosed in the @*S* tags.

```
@S[
  <br>
  BALLARD - 1 Bedroom <br>
  new cpts, drapes & paint, nr hospital & bus.
  $535.  <br>
  1519 NW 65 th, Apt #4 <br>
  206-542-4600 <br>
]@S
```

For its learning algorithm to function properly, WHISK needs a set of training examples to learn the extraction rules. If the above text was fed as a training exampel to the application, the following lines would be added at the end and the user would be prompted to fill in the data which should be extracted from this particular text (here shown with the values "1" and "535" already filled in).

```
        @@TAGS Rental {Bedrooms 1} {Price 535}  @@COVERED_BY
        @@ENDTAGS
```

Once the updated sample with correct values filled in handed to WHISK for a re-processing, it will generate an extraction rule in a form of

```
Extraction rule: * (<Digit>) 'Bedroom' * $(<Nmb>)
Output: Rental {Bedrooms @1} {Price @2}
```

The capturing groups in the extraction rule on the first line are assigned to the slots numbered @1 and @2 in the output pattern. The string "Bedroom" could be replaced by a semantic class Bedroom ::= $(br\|brs\|bdrm\|bedrooms\|bedroom)$ to cover all possible occurrences in different texts.

For free texts (i.e. plain text documents with no visual clues to help with partitioning), WHISK relies on syntactic analysis performed by the BADGER [31] lexical analyzer. Afterwards, it can utilize some special constructs such as PObj, @Passive, Person, etc. in its extraction rules, since these information will be available in the pre-processed source text.

In general, WHISK suffers the same disadvantages as any tool with forced user interaction and manual creation of training samples does. To lessen the manual burden on the user, WHISK does support batch mode processing for more samples, but the initial rule creation still has to be done manually and the rules need to be updated to accommodate for any change in the text structure of the documents.

### 2.2.4 Wrapper Induction tools

The category of wrapper induction tools represents tools, which focus mainly on the task of inducing the wrapper based on a provided sample input documents. This means that these tools are capable of developing delimiter based rules, which do not require lexical pre-processing.

Note that there is a small functionality overlap with the tools presented in the previous section, since some of the implementations presented there are also able to induce the extraction rules without the need for prior lexical analysis.

#### WIEN [35]

WIEN is the first wrapper induction system and as such is the pioneer of utilizing the (semi-)automated wrapper induction to circumvent the manual creation of extraction rules. WIEN paper also formally states the definition of wrapper induction problem, namely

> For a given wrapper class $\mathcal{W}$ and an input set $\epsilon = \{..., \langle P_n, L_n \rangle, ...\}$ of examples, where each $P_n$ is a page and each $L_n$ is a label, generate a wrapper W $\in \mathcal{W}$ such that $W(P_n) = L_n$ for every $\langle P_n, L_n \rangle \in \epsilon$.

As for the induced extraction rules, WIEN uses one multi-slot pattern and rules in a form of regular expressions with capturing groups (LR extraction rule class), e.g.:

```
* '.' (*) ':' * '(' (*) ')'
```

WIEN features several other different rule classes, such as HLRT, OCLR, HOCLRT, N-LR or N-HLRT, which are more complex than the basic LR class and can utilize concepts such as document head/tail delimiters or tuple delimiters.

#### SoftMealy [36]

SoftMealy is an advanced wrapper representation formalism based on finite-state transducer (FST) and contextual rules. Utilizing FST allows it to perform on web pages, which would not be processable using techniques relying on valid HTML structure, like DOM tree construction. Furthermore, it allows for processing pages containing attributes with missing or multiple values, variant attribute permutations and typos.

The algorithm works on a pre-processed web page, which has been tokenized into a sequence of tokens in form Class_name(token_value), such as Num(123) or Html(⟨ul⟩). Before performing the actual data extraction, a set of training documents has to be supplied for SoftMealy to learn the contextual rules and determine the separators.

Separators in SoftMealy are abstract and have the same function as delimiters used in other tools' extraction rules (e.g. WIEN). They can be defined as two sets of tokens, one for context preceding the separator, and one for the context following it. To give an example of separator using the previously described tokens, consider a piece of HTML code

```
<LI> <A HREF="http://www.sample.com">John Smith</A>,
     <I>Professor of Computer Science</I> </LI>
```

The separator between tag ⟨I⟩ and string "Professor" can be described as

```
    separator_prev = ... Punc(,) Spc(1) Html(<I>)
    separator_next = C1Alph(Professor) Spc(1) 0Alph(of) ...
```

where the tokens (from left to right) are for punctuation character, space, html tag, string with first capital letter, space and string without capital letter. However, this separator is unique to the previous sentence (tuple) and must be later generalized by similar structures found in another documents, so in the end, the rule will look similar to this:

```
    separator_prev = Html(</A>) Punc(,) Spc(_) Html(<I>) |
                     Punc(_) NL(_) Spc(_) Html(<I>) |
                     Punc(,) Spc(_) Html(<I>)
    separator_next = C1Alph(_)
```

with underscore (_) being a wildcard character. Using the separators unique for each tuple, one FST will be constructed for each type of tuple. These FSTs will be then used in the actual data extraction. One drawback of this approach is the fact that context rules can utilize only the immediate surrounding for any given information to be extracted.

### STALKER [33] [34]

STALKER is one of the more recent and more advanced tools and as such is capable of hierarchical data extraction. It is loosly based on the wrapper induction techniques used by WIEN and SoftMealy.

The inputs for STALKER algorithm consist of a set of tokenized training examples and a description of the pages' structure, called an Embedded Catalog Tree (ECT). The root of the ECT is a node representing the whole web page, the leaves of the ECT represent individual data records to be extracted and the inner nodes of ECT contain embedded lists. A sample ECT from authors' paper is below [33]:

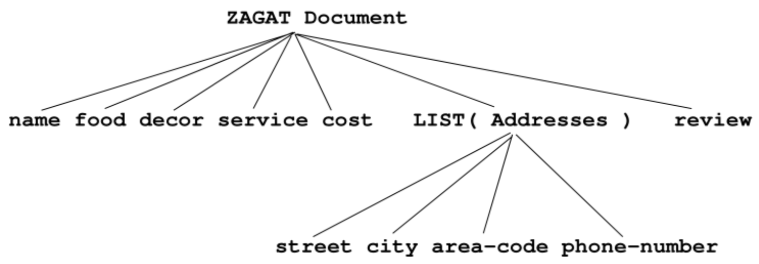


Figure 2.3: ECT description of a web page

As for the extraction rules, STALKER relies on user participation during the learning process. The user has to specify a set of web and highlight all the data of interest using the provided GUI. Afterwards, STALKER will use this information with conjunction with the provided ECT to generate and refine extraction rules. Each extraction rule corresponds to one node of the ECT.

### 2.2.5 Modeling-based tools

**NoDoSE [37]**

NoDoSE (Northwestern Document Structure Extractor) is a modular interactive tool for semi-automatic data extraction. It was designed as a test bed to study the data extraction problem and takes a different approach to the wrapper induction task, using a graphical user interface (GUI), so the user can hierarchically decompose the source document and highlight regions containing important data. After the initial decomposition, the user can add different documents, similar to the original one, and further refine the induced internal model. Afterwards, the resulting model is passed to the data mining module for interpretation and grammar creation. The grammar resulting from the last step is finally used to extract the data from actual documents. The use of NoDoSE is not limited to the web pages, as it can try to process semi-structured documents such as various reports.

**DEByE [38]**

DEByE is a modeling tool capable of extracting data from web pages. It requires the user to specify a set of training examples, in which he needs to highlight the data of interest in form of nested tables. This structure was chosen for being simple, intuitive and expressive enough, so the user does not need to know anything about HTML, special delimiters or automata. Same as NoDoSE, DEByE also provides GUI for the user interaction.
DEByE then takes the decomposed samples and constructs object extraction patterns (OEP; OEP objects are organized in tree structure), which are used for communicating with the extractor module.
The extractor module has two available strategies at its disposal, a top-down and a bottom-up strategy. The top-down strategy combines all the OEP objects, creating one encapsulating OEP object for the whole document. The extractor will then use this one object to recognize and obtain data objects from input web pages.
The bottom-up strategy is more complicated, more resource consuming and arguably more efficient (considering recall and precision of this type of matching), as it consists of matching atomic regular expressions in the leaf OEP objects and assembling the matched ones to create a resulting structure.
However, the computational complexity of the bottom-up strategy can be viewed as a major flaw to DEByE extraction phase. What is more, even the authors admit this approach possibly produces numerous false-positives and false-negatives (i.e. matching instances that should not have been matched and not matching instances that should have been matched).

### 2.2.6 Ontology-based tools

#### Ontology definition

An ontology can be defined as [42] “a formal, explicit specification of a shared conceptualization”. In other words, ontology is a structural framework for storing and organizing information about a certain domain. Each ontology can contain several types of elements. The most basic of these elements are individuals (instances), classes (concepts), attributes and relations (properties). Furthermore, ontologies can contain some more complex elements such as function terms, rules and restrictions, axioms, and events. Let us explain the basic elements:

- INDIVIDUALS represent the most elemental objects from the domain. An example of an individual can be “dog” in an animal ontology, or “Golden retriever” in a dog ontology.
- CLASSES represent a type, a category, a kind of things. Classes are more abstract than individuals and may contain individuals, other classes or both. An example would be a class “Thing” for all things, or a class “Person” for all people. Classes are the building stones for any ontology and have a lot of theoretical consequence attached, but for the purpose of this thesis, none of this is of major significance.
- ATTRIBUTES are used to attach additional properties to objects in ontology. An example would be a dog class having the attribute “number of legs” with value 4.
- RELATIONS express the relationships between object in ontologies. One of the most used relations is the is_a relation (a.k.a. is_a_subclass_of, is_a_subtype_of) for describing a hierarchical relationship between two objects, e.g. “Dog” is_a “Animal”.

#### Ontos [39]

ONTOS (Ontology Extraction System[1]) represents one of the earlier ontology-centered tools for the data extraction. It was released by Data Extraction Group [40] at Brigham Young University (BYU) in Utah. The ontology-based approach is suitable for documents, which contain a vast amount of data and are narrow in ontological breadth. This means that a document has to have a number of identifiable constants (dates, names, identifiers, monetary values, etc.) and the knowledge about its domain can be described by a relatively small ontology focused on this domain of interest. Furthermore, the document should have information concerning one entity in a somehow concentrated regions, such as a sequence of single advertisements or reports, which do not overlap. None of these assumptions is unrealistic, since all three conditions express the way web documents are usually structured. The data extraction can be divided into two phases, a pre-processing phase and an actual processing/extraction phase.

[1]This actual name of the tool is not mentioned in the cited document, but the authors of the tool renamed it to Ontos with inclusion to the Ontos framework. It is also referred to as OntoES in the following paper [44]

The *first phase* consists of two steps:

1. developing an ontological model for the domain, from which will the documents originate
2. parsing this model (ontology) to generate a database schema and rules for constant/keyword matching

For the ontology to have a sufficient complexity and validity, its construction must be done manually by an expert. This is by far the most time-consuming step in the whole matching process, however, once a domain ontology is completed, it can be used for matching any documents from this domain without a need for change, independently on how much the source documents change. Furthermore, there is currently a number of ontologies made publicly available on the Internet, some of which could be used for this purpose.
The *second phase* — the actual matching phase — can be divided into several steps as well:

1. obtain documents of interest and separate the text sections containing individual records from it. In this step, the decomposition is done by analyzing HTML tags and constructing a tag-tree
2. apply two heuristic analysis: one to determine which branches of the tree represent separate data instances and one to determine the separators used to delimit the individual data fields in each instance.
3. strip the HTML tags from the parts of text corresponding to single instances and store the resulting unstructured text for the further processing
4. extract the data objects from the text using a set of recognizers administering the rules developed in the first phase
5. assign the extracted objects to the actual records in database schema (using other heuristics)

All the components involved in the compound matching process are general and do not need to be changed when using the tool for data extraction from documents originating from another domain — with the exception of the ontology. This is a serious advantage against the other wrapper generation tools, which demand their extraction rules are rewritten every time the web page structure changes.
Some other noteworthy features implemented in OntoES include the set of heuristics used for identification of separate instances in one monolithic document. This set currently contains 5 different strategies of ranking HTML tags, which could possibly be instances delimiters. These strategies are

- Highest count ranks the candidate tags depending on the number of occurrences in the web page, assuming high number of occurrences for a certain tag corresponds to a high number of records present in the document
- Identifiable separator uses predefined lists of tags, which are most commonly used to separate instances, such as ⟨td⟩, ⟨p⟩ or ⟨h1⟩

- Standard deviation assumes the instance records are of similar size and therefore it tries to identify real delimiter tags by calculating a standard deviation of plain text enclosed by two identical tags
- Repeating pattern assumes the records are divided by the same sequence of tags, such as ⟨br⟩⟨hr⟩
- Ontology matching uses the list of record-identifying fields, i.e. fields occurring exactly once per each instance. This heuristic counts the number of occurrences on such fields on the page, calculates an average of these occurrences and then compares these numbers to the numbers of would-be instances produced by the candidate tags.

Each of these heuristics presents a list of ratings for the candidate tags in the web page. These ratings are then used in the formulas from *Stanford certainty theory* [43] to determine the correct separator tags. Authors state that the empirical results of using this technique on a corpus of 120 web pages resulted in correct delimiter identification in 100% of the test cases.
The second interesting feature of Ontos is aimed at the disambiguation of data of the same data type present in one instance. First of all, it hypothesizes that the important information is presented closer to the beginning of the record, e.g. if the tool encounters a name of a person in a obituary text, it can safely guess this is the name of the deceased and not the name of one of his relatives. Secondly, it uses a concept of keyword proximity — if there are three dates in the above-mentioned obituary, it searches for keywords like "born","passed away","last goodbye" or "funeral service" and determines that these dates are in fact date of birth, date of death and the funeral date, depending on the distance of each found keyword from the instance of a date.

**FROntIER [44]**

FROntIER (Fact Recognizer for Ontologies with Inference and Entity Resolution) represents a framework for an automatic extraction and organization of data from historical documents using extraction ontologies and organization rules. On the top of this, this framework is capable of fact disambiguation and additional fact inference. Even though this framework was developed for a very narrow usage purposes, it has been included in this thesis as an example of novel work concerning data extraction — namely utilizing several external tools instead of relying solely on user-provided modules. What is more, after supplying the extraction ontologies for different domains, this tool could arguably be able to operate on these domains as well.

The actual data extraction is handled by Ontos [39] and the output is converted from XML to RDF after the extraction, creating an OWL ontology. This ontology is later passed to Jena reasoner [45] to infer additional implied facts, such as if B is a son of A, then B is a child of A. Also, if C is a child of B, C is a direct descendant of A. The output from Jena reasoner is converted to OSM data instance model, which is subsequently converted to comma-separated value file, which is then used as an input for Duke deduplication engine [46] for instance disambiguation.

# 3. SemJob Implementation

## 3.1 Initial motivation and specifics

The SemJob project [47] was developed as a complex solution to allow a classification of documents from a firmly defined domain, with an option to change this domain by providing updated configuration files. Being bound to a single domain of interest is a significant influence factor for the final design of this application, since it predetermines certain approaches and technology choices, such as the usage of ontologies for storing knowledge about the domain and being able to precisely specify the extraction rules to be used for IE. The initial domain consisted of documents with job description advertisements (JD), therefore any examples in the text will be also based on this domain. The choice of the domain was not entirely arbitrary, but it has proven to be a reasonable decision due to the usual content and structure of these documents.

### 3.1.1 Performance specification

Performance requirements imposed on the system were divided into two parts according to different stages of document processing. In the first stage, document must be pre-processed using a set of linguistic tools designed for the specific language (i.e. Czech). The pre-processing is not a full NLP, since SemJob only relies on tokenized and lemmatized input, as opposed to full POS tagging. Afterwards, IE is performed on the document and the results are stored in the database for later usage. In the second stage, documents stored in the database are being queried and the results of these queries are displayed to the user via suitable graphical interface. This part is very performance reliant, as the users will not be willing to wait for the results longer than a few seconds, whereas the first part is not a subject to this restriction (administrator can run it in a batch during the night). With this in mind, the pre-processing and following IE does not have to be optimized for run time, but can rather focus on delivering more precise results using algorithms with extensive processing. Furthermore, the second part does not concern the actual data extraction and therefore will not be discussed in this thesis.

## 3.2 Input example

A common JD contains several sections (which tend to be even divided into separate paragraphs), each of them having a constricted set of information it can contain. Below is an example of a sample JD containing 3 sections:

```
Requirements:
- 4-year degree in field
- experience with working for IT company, leading a team
- knowledge of network architectures, network security
- good knowledge of HW and SW (servers, backup, Windows XP/W7 etc.)
- excellent communication skills, reliability
- driving license

About the position:
- managing several servers, backups
- managing IT processes of the company
- managing of a computer network (approx. 100 PC)
- optimization of current IT processes
- communication with customers

We are offering:
- salary EUR 50k pa
- benefits: cell phone, car, 13th salary
- career growth opportunity
```

As we can see in this structure, all of these sections can be identified by an unique header. The assumption about multiple types of text sections had to be done to be able to efficiently extract information from the text, as the same keywords can attain different meaning in the context of different sections, e.g. for a keyword "server" this may mean the difference between "knowledge of servers" in the requirements section and the "server management" in the section with position information. The text format shown here can be considered semi-structured text, not as structured as a HTML or XML file and not a plain text either. Furthermore, the documents usually adhere to this bullet list format.

## 3.3 Output example

The IE being performed on the document needs to have a reasonable output, so that the extracted data can be later used for other purposes. Because of this, it has been agreed that the output data will be in a RDF format [14], which can in turn be stored in a suitable database able to handle this format, such as Virtuoso database used by this implementation. This also enables precise classification of all the information found during the IE which could not be possible if the implementation was utilizing only simple text strings. The inclusion of ontologies in this process follows the same goal. First ontology is the domain ontology containing classes for each of the items which could be found in the IE process. If a matching process finds a match in the source document using one of the defined rules, it outputs a RDF statement using the assigned ontology class. To clarify, consider this clause

```
excellent knowledge of Java is required.
```

Utilizing the fact that this sentence would be found in requirements section of the JD and extracting the data into RDF format, we would arrive at this output (namespaces omitted for simplicity; each row is one RDF triple)

```
jd01 a :JobDescription.
jd01 :requiresKnowledge knowledge01.
knowledge01 a :Knowledge.
knowledge01 :ofWhat :Java.
knowledge01 :withLevel :Excellent.
```

Note that all items prefixed by : are already defined in the class ontology as either classes (first letter is capital), or properties (otherwise)[1].

This way it is possible to extract and archive as much information as possible, while also allowing for ontology-based search tools to perform any searches and further process the data. The usage of ontology output format has an influence on other parts of design not directly connected to the output itself, such as configuration. It is sensible to have the extraction rules encoded in a way similar to the output, since these rules are the templates used to extract the information in the first place.

## 3.4 Implementation

The implementation of SemJob consists of several main modules. Even thought, for the purpose of this thesis, the main focus lies on the extraction module, high level structure of the tool is provided in the following figure:

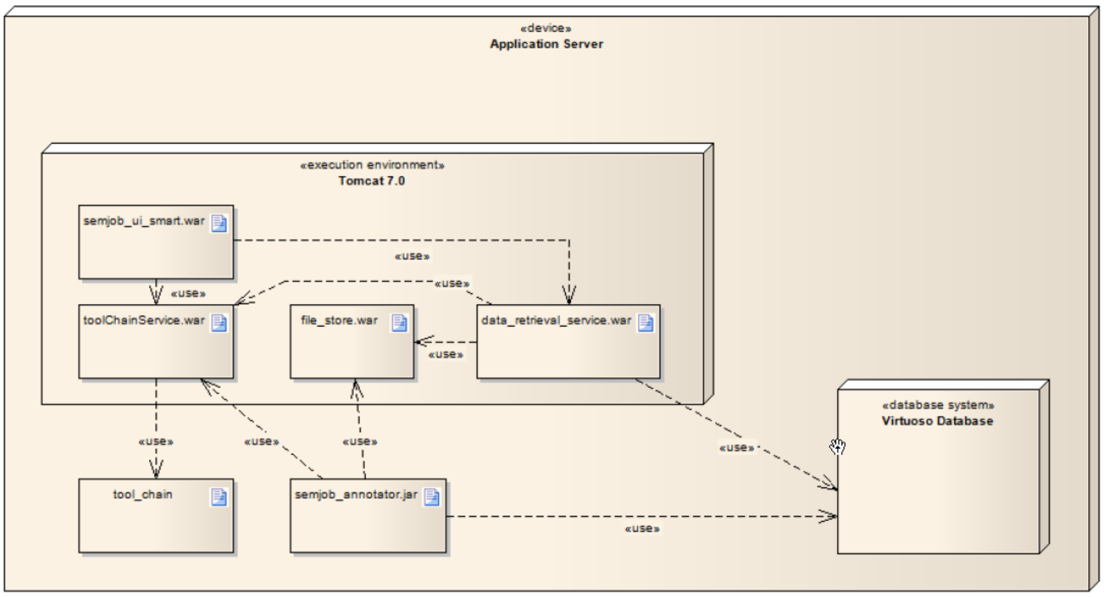


Figure 3.1: High level structure of SemJob tool

As outlined in the previous chapter, SemJob implementation divides the process of IE into two parts, the pre-processing and the actual extraction. This tasks are performed by the two modules at the lower left corner of the previous figure, tool_chain and semjob_annotator.

### 3.4.1 Pre-processing

The document from which we want to extract data is first submitted to a lexical analyzer (in this case, the tool_chain tool). Lexical analyzer is a tool designed for converting a plain text document into a sequence of tokens. The plain text document by itself is basically one long string of characters, numbers and interpunction. A token is a much shorter string of characters, numbers and interpunction

[1]This is an arbitrary naming convention used for clarity and not an ontology requirement.

having an atomic meaning. One token can represent for example a name, a date or an end of a sentence. The process of creating a sequence of tokens is called tokenization. Once again, even though the tool_chain would be capable of POS tagging, we do not utilize this functionality in SemJob, since it is not guaranteed the results would be correct for semi-structured text without complete sentences. After tokenization, the output is further processed to create a in-memory encapsulating object representing the whole document, internally called a document tree. The document tree has several helper methods for handling the text, since in reality, different levels of granularity are needed and "whole text" or "one token" simply do not suffice.

## 3.5 Configuration

For the matching algorithm to be as general as possible while still allowing for a vast number of adjustments and special rules, there is a need to have complex configuration structure — in case of SemJob, two configuration files are used. These two files together are used to define the extraction rules to be used in the matching process. One of them is the RDF configuration file and the second one is a XML configuration file. As mentioned before in the section about output 3.3, the configurations are interconnected tightly with output, meaning the same individuals and ontology classes are used both in configuration and subsequently in the output. This way, the output is automatically linked to the ontology which was used in the extraction process.

### 3.5.1 The RDF configuration

The RDF configuration contains extraction rules for document chunks, each chunk being affiliated with a certain class or property from the ontology. This way the user has an option to specify an individual matching strategy for each single piece of knowledge, such as address or language skill required for a given working position. Currently, there are 3 different strategies available:

- `Regexp` strategy
- `Taxonomy` strategy
- `DecisionTree` strategy

These strategies have a massive impact on the way the matcher behaves and can be combined together to tailor a very specific set of requirements which will in turn preordain how and when a part of the document will be matched. All the examples in this section use the Turtle syntax.

Please note that even though the usage of the name `DecisionTree` strategy, the strategy is in fact not a decision tree in a meaning used in other literature.

## Regexp strategy

This is the most basic strategy and uses regular expressions for finding a suitable match in the document. The user can specify a regular expression which corresponds to a certain item he wants to extract from the text in the source document.

Example: Let us consider an `:Address` class in the ontology. Now consider there is a property called `:hasStreet` with a domain being `:Address` and range being class `:Street`. The class `:Street` has further a property called `:hasStreetName` with domain being `xsd:string`. If we wanted to create an extraction rule for matching a street given the above-mentioned structure, we would need to create a new ontology class of type configuration. This class will need to have an assigned strategy and this strategy would be of type `:RegexpStrategy` with one (or more) associated regular expressions.

```
:hasStreetNameConfiguration
    :type :Configuration;
    :isConfigurationOf :hasStreetName ;
    :hasStrategy [
        :type :RegexpStrategy ;
        :hasRegexp "([\\p{L}\\p{Nd}]+)"^^xsd:string
    ] .
```

This configuration contains only one strategy – the `:RegexpStrategy`. Furthermore, this strategy contains only one `:hasRegexp` property with the regular expression specified (if there were several regular expressions, matcher would try all of them in order to find a match).

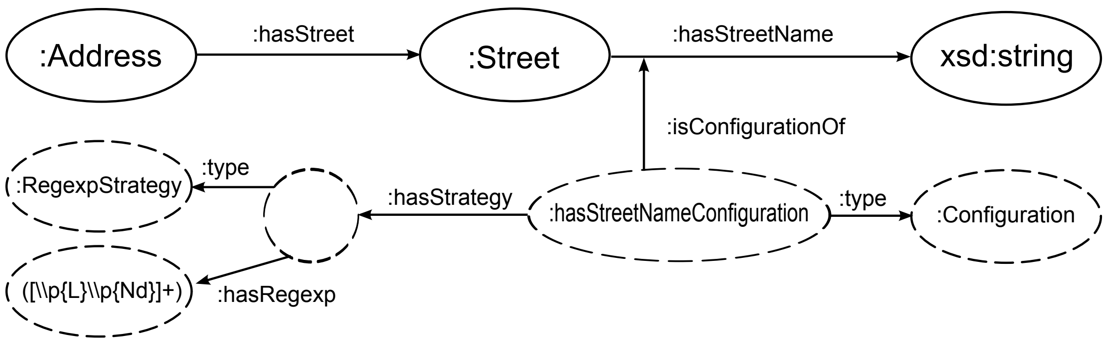


Figure 3.2: Ontology relations between :Address and :Street classes, together with the Regexp strategy for the :hasStreetName property

The Regexp strategy allows for an extraction of a specific group of the provided regular expression by using a :groupNo property. Group with index 0 is the whole matched regular expression, groups numbered above this are assigned parts of matched text in order from left to right. For example, if we wanted to match two numbers representing an interval, the regular expression would be `([0-9]+)-([0-9]+)`.

### Taxonomy strategy

This is an advanced strategy for finding matches and relies on a dictionary-like approach using taxonomies. These taxonomies are in fact also using the ontology format and contain additional knowledge about the domain of interest. Using the class Address from previous example, we can now write another example, in which we extract the name of a town in an address, given a taxonomy of towns.

Example: We will use new class `:Town`, which is connected to the `:Address` class via a `:hasTown` property.

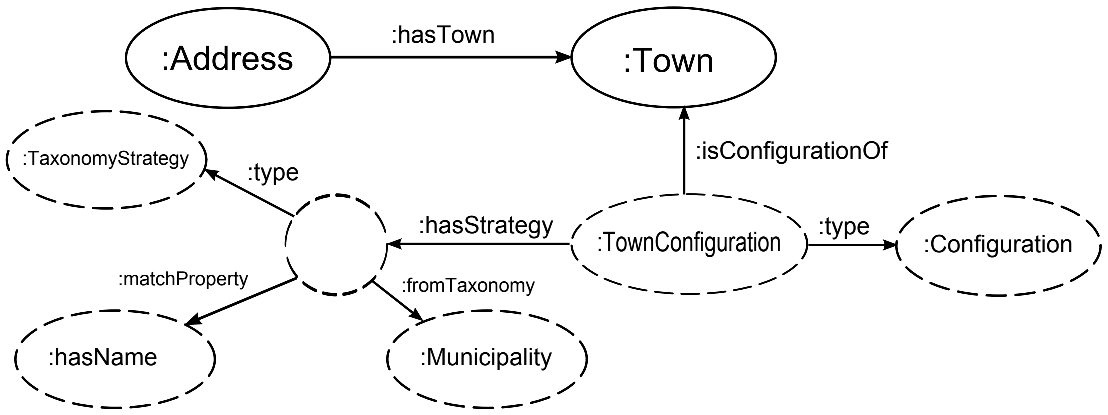


Figure 3.3: Ontology relations between `:Address` and `:Town` classes, together with the Taxonomy strategy for the `:Town` class

```
:TownConfiguration
    :type :Configuration ;
    :isConfigurationOf semjob:Town ;
    :hasStrategy [
        :type :TaxonomyStrategy ;
        :fromTaxonomy :Municipality ;
        :matchProperty :hasName
    ] .
```

The instances in the municipality taxonomy can contain several properties, which can be fairly useful when further processing the results of matching. Below is an example of an instance of the Prague area. Note that the property specified in `:matchProperty :hasName` is present in this instance and therefore the matcher will use it in the matching process:

```
:area-praha
    :type :Area ;
    :hasName "praha" ;
    :includesLocation :municipality-32767 ;
    :isIncludedByLocation :county-praha .
```

As we can see, the instance `:area-praha` is of type `:Area`, thus denoting this is not the city of Prague itself. However, this area includes the location `:municipality-32767`, which is an auto-generated instance representing the city of Prague. The instance also states it belongs to a parent instance, in this case the `:county-praha`. Utilizing this transitive search features in conjunction with this tree of instances, user can later search for all objects in the `:county-praha` and receive this instance among other results.

### DecisionTree strategy

This strategy is the final and most complex strategy and is used for constructing hierarchical tree structures using the previous two strategies. This strategy can be pictured as a tree template with empty nodes, which can be filled by a certain `Regexp`, `Taxonomy` or `DecisionTree` strategy. Let us look on an real life example of an address.

```
Address: Malostranske nam. 25, 118 00 Praha 1
```

In this example, we can see the street name followed by the street number, a comma, ZIP code and finally a town name. This would be impossible to match using any of the two simpler strategies, since the address format can slightly change depending on the source, and this change would invalidate any regular expression tailored specifically for this format. With the use of `DecisionTree` strategy, we are able to construct a tree structure, which will be much less dependent on the precise format.

```
:AddressConfiguration
    :type :Configuration ;
    :isConfigurationOf :Address ;
    :hasHeaderRegexp "Address"^^xsd:string;
    :hasStrategy [
        :type :DecisionTreeStrategy ;
        :hasNode [
            :property :hasStreet ;
        ] ;
        :hasNode [
            :property :hasTown ;
            :hasNode [
                :property :hasZipCode ;
                :close-to :hasTown ;
                :distance "2"^^xsd:integer ;
                :isOptional "true"^^xsd:boolean
            ] ;
        ] ;

    ] .
```

Some interesting new properties appeared in this example, these are the relationship properties. Firstly, we have implemented `:close-to`, `:before` and `:after` properties, all of them usable with the `:distance` property. In the above example, the combination of `:close-to :hasTown` and `:distance "2"^^xsd:integer` means that the property `:hasZipCode` can be found around the `:hasTown` property with the maximum distance being 2 words before or after the `:hasTown` property. The other relationship properties behave accordingly. Next, there is a new `:hasHeaderRegexp` property, which can be used also in previous strategies. This means that the text to be matched using given strategy must contain this regexp, which is useful in this case, as we can further restrict matching of addresses only if the actual address is prefixed by a string `"Address"`.

### 3.5.2 The XML configuration

The XML configuration is used to describe the "bigger picture"; it gives the user a tool to describe the high level structure of how the document will be matched. It also utilizes the tree structure and divides the document into hierarchical structure of sections. Furthermore, this configuration is used to define what the output of the matching will look like by assigning an element from the ontology to each section.

The configuration follows a simple structure depicted by the following grammar:

```
CONFIGURATION -> SECTION
SECTION -> HEADER | HEADER, BODY | BODY | empty_section
HEADER -> EXPRESSION
BODY -> SET | CHOICE | SEQUENCE
SET -> SECTION | SECTION, SET
CHOICE -> SECTION | SECTION, CHOICE
SEQUENCE -> SECTION | SECTION, SEQUENCE
EXPRESSION -> regular_expression
```

The beginning non-terminal symbol (NT) of the grammar is `CONFIGURATION`. This NT can be expanded into several levels of nested NTs `SECTION` which must ultimately be replaced by one of the two possible terminal symbols – either `regular_expression` or `empty_section`. The former is used for including an actual regular expression representing the text which must be found in the input document, whereas the latter is usable for linking this `SECTION` to an ontology class from the RDF configuration, thus allowing for use of one of the three matching strategies.

Now we can characterize the XML format in which the configuration has to be written. The top-most root of the XML document is element `:Configuration`, similar to the `<HTML>` tag of HTML documents. This element usually contains only one section, which corresponds to the whole matched document. If we were to match a job description, our XML configuration so far would look like this [2]:

```
<?xml version="1.0" encoding="UTF-8"?>
<:Configuration>
    <:section :scope="DOCUMENT" :outputElementType=":JobDescription">
    </:section>
</:Configuration>
```

[2]Namespaces are omitted for clarity

What we can see here are the two required attributes of each section. First one, `:outputElementType` is the above-mentioned link between this section and an element in the ontology, in this case `:JobDescription`. The second attribute is `:scope`, which restricts on which part of the document should the matching of the described section be performed. The `:scope` attribute has several firmly defined values implemented, namely

- **DOCUMENT** this section corresponds to the whole document
- **PARAGRAPH** this section corresponds to one paragraph of text
- **SENTENCE** this section corresponds to one sentence of text
- **WORD** this section corresponds to one word of text
- **NEXT_HEADER** this section corresponds to previously unknown amount of text - up until another section can be identified

These scope attributes are influenced by the nesting, therefore if a nested section has wider scope than its parent section, it will adopt the parent sections' scope.

Every section can have either have two child elements (header and body), only one child element (body) or no child elements. In the header element, user can provide a regular expression characterizing this section (similar to `"Address"` string used in RDF configuration to denote the occurrence of an address in the text). However, this is optional and user can omit this element.

In the body element, the user must specify a collection of sections, choices currently are a set and a sequence. If the user states that a section contains a set of other sections, the child sections can be matched in any order. The sequence collection means the child sections must be matched in the defined order. This order is checked after the matching and if it does not correspond to the configuration, matching of this section will be invalidated.

Last but not least, if the user does not include neither header nor body elements, the value of `:outputElementType` attribute is used to find a configuration for this section in the RDF configuration file. This is the preferred way of structuring the XML configuration, since it makes editing of this configuration much more straightforward and transparent.

We can expand the previous example to provide a simple XML configuration for matching a document containing a job description:

```
<?xml version="1.0" encoding="UTF-8"?>
<:Configuration>
    <:section
      :scope="DOCUMENT"
      :outputElementType=":JobDescription">
        <:body>
            <:set>
                <:section
                  :scope="SENTENCE"
                  :outputElementType=":hasSalary">
                    <:header>
                        <:expression>(Salary)</:expression>
                    </:header>
                    <:body>
                        <:expression>(\d+)\s(EUR|CZK)</:expression>
                    </:body>
                </:section>
                <:section
                  :scope="SENTENCE"
                  :outputElementType=":hasAddress" />
            </:set>
        </:body>
    </:section>
</:Configuration>
```

## 3.6 Data extraction

After the document is pre-processed and the document tree is generated, it is passed to the matching module. Here the actual matching takes place. Alongside the document tree, both configurations from previous section are included as inputs for the matcher. They influence all aspects of the matching such as division of text into different sections and forms of extraction rules. The main matching algorithm is recursive and fairly straightforward.

### 3.6.1 Matching algorithm

1. If the section to be matched is not a leaf section in the configuration (it has at least one child section defined)

   - Match the child sections headers in the text satisfying the scope assigned to each section. If the collection for child sections is defined as a sequence, child sections headers must be matched in the precise order in which they are defined.
   - Choose the sections with a scope defined as "next header". These sections need to be matched first, as the allocated text depends on the other section headers - some of which may not be found in the text yet, thus designating this matching order. Once the algorithm matches a child section on the allocated text in the source structure, this part of the structure is removed, since it is presumed that one block of text can only belong to a single section.
   - Pick and match one section with header from the remaining sections.
   - Pick one section without header from the remaining remaining sections. This is the last resort of the matching algorithm, since it cannot determine, where exactly should these sections begin and end. Therefore, the algorithm iterates through all the remaining segments in the document tree (which were not assigned by previous steps) and tries to match this section using a depth-first search.

2. If the section to be matched is not a leaf section in the configuration (no child sections defined) and has an assigned matching strategy

   - Use the assigned matching strategy of this section on the text allocated for this section according to the scope restrictions. The matching will stop after a first successful match. If the matching fails, this section cannot be matched. Furthermore, if the section is defined as a required section for the whole document, the whole document matching will fail.

Note that there is a significant difference between trying to match a section with included child sections and matching a leaf section. While the sections with included child sections have a function of inner nodes in the virtual section tree, the leaf sections serve as the final segments and are the ones performing the actual matching. The inner sections are similar to the decision tree strategy mentioned in the configuration section, since they act as guides in the matching process.

### 3.6.2 Matching a section with `Regexp` strategy

This strategy has a list of defined regular expressions and the matching consists of simply comparing the input string with these regular expressions. If the regular expression belongs to a header part of a section and therefore is used to determine the start of a whole section, first successful match will end the matching process. However, if the regular expression belongs to the body part of a section, the algorithm will continue matching the other sentences in this section even after a first successful match. This way it is possible to match multiple data records using one regular expression.

### 3.6.3 Matching a section with `Taxonomy` strategy

This strategy uses a taxonomy and tries to find instances from this taxonomy in the input. The actual matching is performed by a call to an internal taxonomy matcher, which will in turn connect to the data store, obtain a list of instances and perform the matching. The result of this form of matching is a list of taxonomical instances found in the text.

### 3.6.4 Matching a section with `DecisionTree` strategy

This strategy serves for more complex matching using the previous two strategies. With a enhanced section nesting structure, is able to utilize features such as word distance and direction between two matches in the text. It uses a depth-first search to find the matches in the input text, following the tree-like ancestor-successor structure.

# 4. Theoretical comparison

In this section, we briefly recapitulate the presented tools with their characteristic features. A comparison between two or more tools will be presented, if it is in order with respect to the commented feature. The analysis will be divided into several sub-chapters for greater clarity.

## 4.1 Degree of automation

The real world usability depends greatly on the overall automation of a given tool. The automation directly translates into an amount of time needed for a user to prepare the system for real world deployment and data extraction from actual web pages.

The group of tools utilizing a specific language for the extraction rules is the clear underdog in this category, as each of these tools require the user to analyze the source documents by himself and to write the actual code for the extraction rules. Even though the languages usually provide some means for shortening the code, the amount of time the user has to spend looking at the HTML code and determining which tags are the correct separators for the data extraction task is very high.

HTML-aware tools are better off, since they rely on predefined heuristics, which aid them in the partitioning of the web page. A possible pitfall here is the quality of HTML code of the source web page — if the code contains too many invalid HTML tags, the whole result of matching may be in jeopardy. To combat this, most tool employ at least basic heuristics to fix the invalid code (e.g. W4F, XWRAP). Other tools, such as RoadRunner and TEX are able to "ignore" the invalid HTML tags, as their algorithm does not necessarily need the valid HTML code to work properly. Furthermore, both the latter are designed to be fully automatic — neither of them needs user interaction during the extraction phase or during the preparation phase.

The tools based on NLP, wrapper induction and modeling are said to be semi-automatic, since they require the user to prepare a set of training examples, which are then used during the training step. Afterwards, they are able to construct the extraction rules by themselves. The amount of time needed for the preparation of the training examples varies between individual tools - as a rule of thumb, the more interaction the tool demands from the user, the lower is the provided automation and the more time the user has to spend preparing the examples.

Ontology-based tools can achieve feasible levels of automation, once an expert user constructs a suitable ontology. If the constructed ontology sufficiently represents the given domain, the actual data extraction is performed fully automatically. However, constructing and validating a sizable ontology is incredibly time-consuming.

## 4.2 Ease of use

The ease of use of a tool is very dependent on the automation level provided by an individual tool. Fully automatic tools, such as TEX and RoadRunner, are very easy to use, since they do not require the user to participate in the matching process. On the other hand, tools using specific language for wrapper development expect the user to input all the extraction rules by himself (and by hand, since they lack any kind of GUI), making them fairly difficult to use.

Majority of the more recent tools include some sort of GUI — at least so the user can specify training examples. Modeling-based tools take the GUI incorporation to higher level, since their extraction algorithm depend on the decomposition of sample documents done by the user in the GUI. Furthermore, in NoDoSE, user can debug the induced extraction rules using the provided GUI. The GUI is essential part of extraction process also for the XWRAP tool, since it allows user to progress through single algorithm steps and in the end also generates the resulting wrapper.

Ontology-based tool Ontos has a GUI for creating the extraction ontology together with the associated extraction phrases. SemJob does not feature any kind of GUI for ontology specification, but this is not necessary, as there are several third party applications for ontology creation, such as Protégé [49].

## 4.3 Supported input formats

Most of the presented tools rely on one exact input format of the documents for data extraction. In reality, this basically means the tool expects either a web page, or a plain text document.

The web pages contain semi-structured data — the individual data instances are delimited by the HTML markup and the process of data extraction consists of collecting these pieces and putting them together, forming an output object.

The plain text documents do contain some structure, but this structure is much more benevolent than the one expected from the HTML documents. Therefore, the data extraction process is also more complicated, as the tool needs to identify chunks of text containing interesting data and only then can proceed to the extraction.

The languages for wrapper development, HTML-aware tools, wrapper induction tools and modeling-based tools all rely on the structure of the input document to extract data and thus require the input to be in a form of a web page. OntoES, in spite of being a ontology-based tool expects a web page structure as well, since it uses the HTML markup to separate individual data record instances (groups of data records corresponding to one entity) present in the text. If there was only one instance in the whole input document, then if could process a plain text document. FROntIER can process plain text by itself — this is one of the main strengths of ontology-based approach.

In the group of NLP-based tools, AutoSlog, LIEP, Palka and Crystal are able to process plain text input by utilizing lexical analysis. In addition to this, Crystal with help from Webfoot is able to operate on HTML formatted text, but the extraction rules created for the plain text data extraction are not portable.

One special case is WHISK, which is able to operate on both plain text or structured text, but in case of the plain text document it has to be annotated with the tags from lexical analysis.

## 4.4 Complexity of extracted objects

The data occurring in the input documents for any kind of the data extraction usually represents a more complex structure than a simple statement — there is a high chance it has a hierarchical structure with multiple levels of nesting and a certain level of variation. Here we take a look at how well do various extraction tools cope with these challenges.

This is the first category in which the tools with specific languages perform well. Minerva with its Editor exception handling mechanism is able to restructure any irregularities in the input document if an extraction rule fails. TSIMMIS implements the Object Exchange Model (OEM) which was directly designed to serve for exchanging semi-structured data between object-oriented databases, making it ideal for the task of representing nested data. WebOQL is the last tool in this group and is also capable of handling nested objects with variations in their structure. This is made possible by the hypertree structure used for data representation and the flexible query language.
SoftMealy uses finite state transducers with multiple outgoing edges between their states, thus enabling to describe variations in input data structure. However, it is unable to process nested structures. STALKER employs different approach, utilizing the embedded catalog tree (ECT) to describe the tree-like structure of input documents, allowing it to process both structure variation and nested constructs.
In the group of HTML-aware tools, W4F provides a HEL language for defining extraction rules. This language contains the fork operator (#), making it possible to construct more complex structures by creating a tree-like structure similar to ECT used by STALKER.

## 4.5 Resilience and adaptivity

Resilience is based entirely on the matching algorithm used by a tool once the extraction rules are created (either induced, or manually). In general, if a tool's extraction algorithm consists only of searching a match between source text and a regular expressions, any changed in the structure will invalidate all current extraction rules. This is the case with most of the specific language tools (Minerva, TSIMMIS) and some HTML-aware tools (W4F, XWRAP). The modeling-based tools suffer from the same. On the other hand, tools without rigid extraction rules, such as TEX or ontology-based implementations tend to be much more resilient.

Wrapper induction tools may not be able to adapt to changes in the source formatting, but they make up for this by the ability to construct new set of extraction rules from the scratch or by refining the current set.

## 4.6 SemJob features analysis and possible improvements

SemJob has a special place among the other data extraction tools. This is mainly due to the fact the initial requirement was for it to be able to match semi-structured documents. The problem with this type of documents is rather obvious — a tool cannot rely HTML syntax, since there is none present, and it cannot rely on NLP strategies, since there is no guarantee that the NLP algorithms designed to work on whole sentences can successfully process sentence fragments, which are often present in reports and advertisements. Also, since these documents exhibit very high level of variations in their content, our approach with SemJob had to be ontology-based.

The ontology-based tools — as already presented in this thesis — offer favorable properties for texts with high element variance, if they are restricted to a narrow ontology domain, which can be represented in peculiar detail. The tradeoff we discovered immediately after choosing this approach is the need for constructing this ontology by hand, seeing that automatic construction of ontologies does not yield satisfactory results. What is more, initial SemJob domain was in Czech language, thus lowering the chances of automatic ontology construction even more.

The tool designed to work on a domain similar enough to ours is WHISK, but this implementation does not support the extraction of exceptionally complicated data objects, which was also in the prime places of SemJob's priority list. To add to this flaw, WHISK extraction rules take into consideration only the immediate surroundings of data fragment that should be extracted — this may not be an obstacle for the use the authors of WHISK presented in their paper (separated data instances of rental advertisements), but is unacceptable when the goal is to be able to match lexical constructs from real world advertisements like the one below:

```
Requirements:
    - excellent knowledge of Java
    - driving license A, B
```

The second closest is the FROntIER tool, utilizing the ontology-based data extraction from OntoES. Even though authors state, that OntoES can take advantage of HTML tags in the early stage of matching (for partitioning the text into chunks of text, each of them containing one complete data record), judging from the description of the extraction algorithm the authors of OntoES present, it may be able to perform well on the semi-structured plain text.

One interesting fact to note is that the configurations used by SemJob resemble the embedded catalog tree (ECT) used by STALKER, however, we used these configurations to incorporate more structural logic than the ECT.

One important missing feature, which would greatly benefit the usability of SemJob, is a GUI for the user to easily construct extraction rules, or at least to generate some kind of template, which could later be modified by hand. This is the handicap that becomes obvious when comparing our tool to a tool featuring user-friendly interface (e.g. NoDoSE, DEByE).

## 4.7 Condensed featurewise comparison

For quick reference of tools' features and condensed factual survey we present three tables, A.1 for Minerva, TSIMMIS, WebOQL, W4F, XWRAP, RoadRunner and Tex, A.2 for AutoSlog, LIEP, PALKA, CRYSTAL, WHISK, WIEN and SoftMealy and last but not least A.3 for STALKER, NoDoSE, DEByE, OntoES, FROntIER and SemJob. All the values in these tables were filled in to our best knowledge from cited papers. Please note that fields with a value N/A either do not make sense (such as ease of preparation of training examples when no are needed), or could not be found in the information provided by authors (only implementation languages of specific tools).

The selected features for this comparison belong to an extended set of characteristics, which could be deemed useful and/or descriptive for any extraction tool. Many of them are described also in this chapter under the specific feature section.

# 5. Practical comparison

## 5.1 Choice of tools

The choice of tools for an experimental comparison was influenced by several factors. First of all, since the tools reviewed in this thesis are not always recent, there is an issue with the availability of their source code. The second criterion is more performance oriented – since our tool, SemJob, is designed for extracting information from semi-structured documents, the tools with which SemJob will be compared must be capable of operating on this kind of input documents as well. This automatically disqualifies two tool groups, namely the specific language tools and the HTML-based tools.

From the tools accommodating the above-stated conditions, two competitors for SemJob have been chosen – WHISK and Ontos. WHISK relies on relatively simple extraction rules and should be able to operate on any kind of input text, whereas Ontos utilizes the complex ontology-based approach and should perform well on the semi-structured information rich texts. These tools will allow us to evaluate the results achieved by a simpler or a more complex extraction strategy.

## 5.2 Data corpus

The data corpus consists of job description advertisements. This domain was chosen for consistency with the original domain, for which the SemJob tool was developed. The corpus consists of approximately 1700 documents in Czech language, each document having between 5 and 50 kilobytes in size.

The documents were retrieved from a web portal for hosting job advertisements and therefore were initially in a HTML format. Afterwards, they have been pre-processed into the semi-structured text format as the input is supposed to be in this format. Furthermore, all the compared tools do require some form of initial pre-processing.

Ontos did not state any pre-processing request in available documentation. However, after some initial testing, it became obvious that the tool is unable to process input dictionaries and/or documents containing special Unicode characters [1]. Therefore all the input files had to be stripped off letter accents for the tool to operate normally.

In case of WHISK, all documents must be merged into one file named UnTagged. This file must follow a strict format – each original file must be enclosed in a `@S[ ]S@ ID` structure. The documentation states that the tool is unable to

[1]Specifically, even after modifying the framework's input/output methods to be able to handle the UTF-8 encoding, the creation of regular expressions and building of in-memory dictionaries would throw an exception and exit the matching process. Modification of these methods would be out of scope of this thesis, as it would mean a greater change in the original tool

handle characters "+", "?" or "*" and these should be replaced by a string token. An unstated requirement, as was the case with Ontos, is that the files use only the ASCII charset, without any special Unicode characters, which will otherwise cause the rule-learning process to fail.

SemJob requires lemmatized text on its input. Formerly, this task was performed by the tool_chain natural language processing framework as a pre-processing step during the matching process, but this framework has become unavailable by the time of the practical testing. Therefore the set of documents was processed using an alternative NLP framework, Treex [48], instead. This has enabled a significant speed-up of the matching process itself as a welcome side effect.

All the data in the above-mentioned formats can be found compressed in the `data` folder on the attached CD.

## 5.3 Configuration and operation of the tools

### 5.3.1 WHISK

WHISK distribution consists of one file containing the WHISK tool itself, implemented as a Perl script. There are also several accompanying configuration files. For the purpose of our experiment, the configuration in TagSet was adjusted to mirror the information that will be of interest during the evaluation phase. The input documents are merged into the UnTagged file, as mentioned above.

First of all, the tool needs to build up a database of training examples. This is done by running in an interactive mode and selecting the `I` option. The tool then selects several data instances from the `UnTagged` file and moves them to `ToMark` file. The user then has to manually append a set of tags to each of these selected instances, filling in the information to be extracted from the given instance. Afterwards, the `R` option can be specified, which will cause the tool to read the `ToMark` file, move the residing data instances to the `Tagged` file and allow for `L` option to be chosen. This final option will cause the tool to evaluate the data instances in `Tagged` file and derive extraction rules, which should ensure the extraction of suitable data from remaining untagged instances. The rules will then be saved in the `Rules` file. This whole process is fairly cumbersome and repetitive.

### 5.3.2 Ontos

Ontos distribution can be downloaded from DEG BYU web page [40] and comes in one archive. This archive contains the source code for the whole extraction framework. For the experiment, we are interested in the OntologyEditor and Ontos packages. The Ontos package is responsible for the actual data extraction, whereas the OntologyEditor is suitable for creating a valid configuration file, which will be used for this extraction.

The package does not contain any ontologies usable for data extraction by default, however, sample ontologies can be extracted from defunct web demos on the DEG BYU web page [41]. The actual configuration of the tool is done by editing the OntosConfig.properties file, where all the paths to various input files and tool modules need to be stated.

The configuration ontology file is the main mean of influencing the extraction process without modifying the tools code. Using the OntologyEditor module, user can specify which dictionaries to use and how to construct expressions for each information to be extracted. The relationships between ontology concepts are constructed using the main canvas:

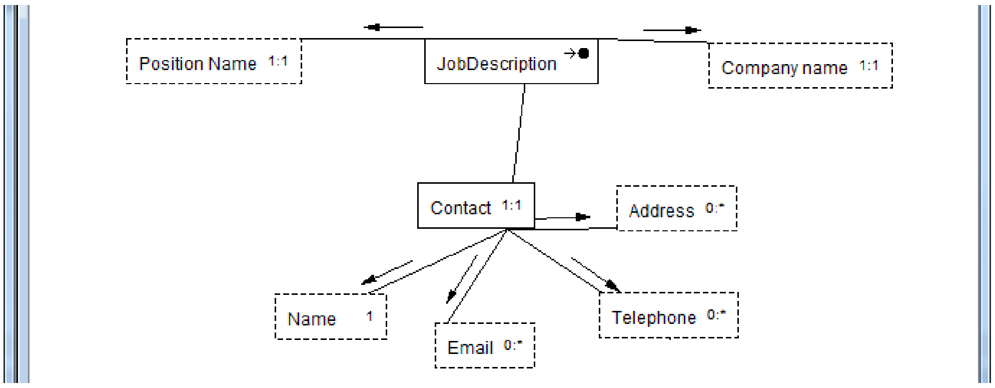
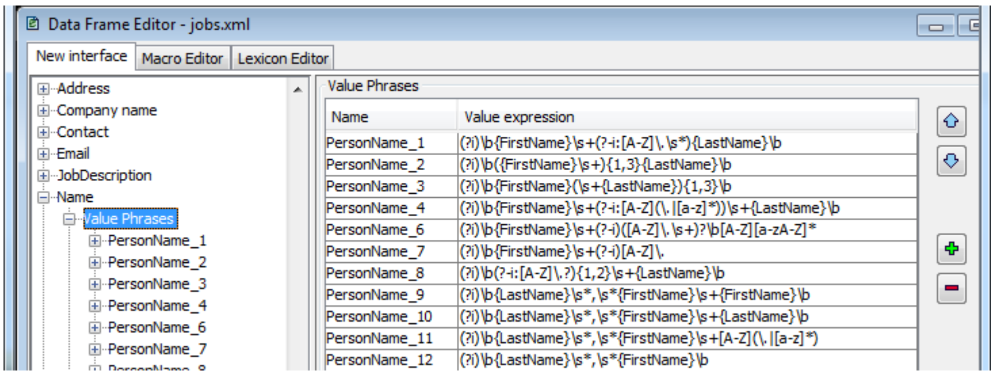


Figure 5.1: Construction of extraction ontology for job description documents

The extraction patterns can be set up using a data frame editor feature. In this editor, the user can specify a "data frame" for each object – that is, a set of conditions used in the matching, which will lead to extraction of a text from the input document and assignment of this text to the ontology object. The data frame consists of value expression, exception expression, left/right context expressions and keyword phrases. Furthermore, there is an option to specify a canonicalization method and extraction methods, but no documentation was available for any of these features. Each of the expression in the data frame represents a regular expression. There is a possibility to include an phrase from the provided dictionaries and to combine these phrases with the regular expressions. On the following figure, we can see the configuration for a personal name, using the `{FirstName}` and `{LastName}` dictionaries containing lists of names:



| Name | Value expression |
|---|---|
| PersonName_1 | (?i)\b{FirstName}\s+(?-i:[A-Z]\.\s*){LastName}\b |
| PersonName_2 | (?i)\b({FirstName}\s+){1,3}{LastName}\b |
| PersonName_3 | (?i)\b{FirstName}(\s+{LastName}){1,3}\b |
| PersonName_4 | (?i)\b{FirstName}\s+(?-i:[A-Z](\.\|[a-z]*))\s+{LastName}\b |
| PersonName_6 | (?i)\b{FirstName}\s+(?-i)([A-Z]\.\s+)?\b[A-Z][a-zA-Z]* |
| PersonName_7 | (?i)\b{FirstName}\s+(?-i)[A-Z]\. |
| PersonName_8 | (?i)\b(?-i:[A-Z]\.?){1,2}\s+{LastName}\b |
| PersonName_9 | (?i)\b{LastName}\s*,\s*{FirstName}\s+{FirstName}\b |
| PersonName_10 | (?i)\b{LastName}\s*,\s*{FirstName}\s+{LastName}\b |
| PersonName_11 | (?i)\b{LastName}\s*,\s*{FirstName}\s+[A-Z](\.\|[a-z]*) |
| PersonName_12 | (?i)\b{LastName}\s*,\s*{FirstName}\b |

Figure 5.2: Configuration phrases for personal name

### 5.3.3 SemJob

The configuration of SemJob can be performed by following the programmer and administration documentations included in the original SemJob distribution. A modified version of SemJob is included on the DVD for the purpose of this testing – in this version, all configuration files have been adjusted to mirror the changes from the original version. Mainly, the tool will now expect the Virtuoso server running locally and the input files will be expected to be in their lemmatized forms. Furthermore, all connections to `tool_chain_service` have been eliminated, since the tool will be operating on the above-mentioned pre-processed input files with no need of the additional NLP.

After setting up the Virtuoso database, the tests can be launched by running `cz.SemJob.test.integration.RealExampleTest`. This test also contains all path settings to the input files.

## 5.4 Observations and results

### 5.4.1 Ease of use (revised)

After having a hands-on experience with the reviewed tools, the ease of use property discussed in chapter 4 can be evaluated further. WHISK requires the user to manually construct training examples by going through randomly selected data instances and tagging them by hand. This task has shown to be extremely tiring. In spite of this, the overall operation of the tool is straightforward, but leaves the user with little to none means of influencing the created extraction rules.

Using the OntologyEditor provided with Ontos has proven to be the most comfortable, as it alleviates the necessity to write XML code by hand. Furthermore, it allows for a simplified construction of relationships between ontology objects, since it handles all the internal identifiers. The interface itself is not as polished as it could be, but this is acceptable as the tool is provided without any warranty and mainly for research purposes. The only disadvantage of this tool is its occasional misbehavior, which causes the produced output ontology to have some of the values reset to their default states, without notifying the user.

The difference between SemJob and Ontos configuration ontology creation is that since SemJob does not have an inbuilt ontology editor and the user must keep track of all the ontology properties and their names. This approach can be problematic when editing an unknown ontology with no prior knowledge of concepts present in this ontology, but once the user familiarizes himself with the structure, he can control the extraction expressions with more precision. The inbuilt/accompanying ontology editor should in any case be implemented and released for future versions of SemJob.

### 5.4.2 Strength of extraction rules

WHISK is an odd one out among the other tools considering the strength of extraction expressions, as the generated rules are only in a form of primitive regular expressions, without any possibility to include any dictionary/taxonomy look up. However, the operation of WHISK tool is accompanied by bigger and more serious problems.

First of all, the work-flow – chose examples from untagged set, tagged them, learn rules, repeat – does not allow for any sensible testing. In the process of learning rules, some untagged instances are moved to the tagged set, since they became covered by the newly adapted rules. Some of the rules may be discarded, if the current set of learning examples suggests there are better rules available (even though this may not be the case, as is demonstrated in the `results\whisk\` folder on the attached CD), causing the existing and already tagged examples to become invalid. There is no way of getting tangible result set in a consistent form similar to the Ontos/SemJob one.

Second and even more grave problem appeared after a while of testing. The algorithm used by WHISK for creating new rules is insufficient for extracting information with even a tiny bit of hierarchical structure, such as an email. The rules generated by WHISK were simply a copy of the string specified as an email address in the learning examples. This outcome of email matching tests after several iterations can be found in the `results\whisk\` folder as well. At this point, we have concluded that this tool is not suitable for trying to match such complicated documents as the job descriptions.

SemJob and Ontos, on the other hand, provide the means for both the dictionary look-up and the context specification. The dictionary utilization in Ontos is done by including the dictionary file and then using a reserved expression in a form of `{dictionary_name}`. In SemJob, the user can make use of the TaxonomyStrategy construct, which will lead to a database query obtaining the desired instances from the taxonomy stored in the Virtuoso database.

As for the context specification, Ontos provides the user with an opportunity to define regular expressions for both left and right context of the desired phrase, whereas SemJob has this option included in the `DecisionTreeStrategy`. Due to the nature of the `DecisionTreeStrategy` construct, this approach is more general and allows for matching of more complicated text structures than the simple left/right context. What is more, `DecisionTreeStrategy` does support reusability – once an child strategy object is defined, it can be easily used as a bounding condition for a number of other `DecisionTreeStrategy` objects. Last but not least, due to the implementation of the properties like `:close-to` and `:distance`, the extraction rules definable by SemJob are strictly more expressive than those definable by Ontos.

At this point, we should restate that the Ontology Editor supplied with Ontos has means of specifying the canonicalization method and extraction methods for each ontology element, but neither has been used in the available samples. The provided documentation does not state anything about their actual usage either. Nonetheless, the canonicalization is included by SemJob by default (due to each input document undergoing NLP), so the only slight edge Ontos would have on SemJob in the field of the extraction rule strength is the specific extraction method.

### 5.4.3 Matching results

The quality of matching results from SemJob and Ontos is vastly dependent on the complexity and preciseness of the extraction ontology the user is able to create. For our experiments, we have tried to devote the same amount of time to perfecting both of these ontologies. The ontologies were configured based on a small sample of documents stemming from the corpus, therefore some of the false negatives appearing in the "incorrect" rows in the result table can be attributed to the information format, which did not occur in the sample (i.e. telephone number in a form of `420123456789` when the ontologies were configured to recognize only numbers in a form of `(+420)123( )456( )789()`; the brackets mark optional parts of the expression). The control data were obtained by searching the input files for some distinctive patterns, which indicate the presence of a given piece of information (i.e. searching for a general regular expression `[\w]+@[\w]+\.[\w]+` will yield the number of all the possible email addresses in the corpus)[2]. The patterns were generally similar to those used by either SemJob or Ontos.

| Tool name | | Extraction of Information | | | | |
|---|---|---|---|---|---|---|
| | | contact email | contact phone | contact person | position town | position name |
| Control (grep) | found | 860 | 784 | 1072 | 1490 | all (1783) |
| Ontos | found | 857 | 779 | 1045 | 902 | N/A |
| | incorrect | 0 | 0 | 316 | 0 | N/A |
| | precision | 100% | 100% | 69.7% | 100% | N/A |
| | recall | 99,6% | 99.3% | 97.4% | 60.5% | 0% |
| SemJob | found | 488 | 725 | 1011 | 1057 | 1201 |
| | incorrect | 0 | 10 | 0 | 0 | 3 |
| | precision | 100% | 98.6% | 100% | 100% | 99.7% |
| | recall | 56.7% | 92.4% | 94.3% | 70.9% | 67.3% |

Table 5.1: Results of the practical comparison

The means of obtaining the actual data for the table5.1 were specific to each of the tools. The testing was easiest for SemJob, since its original design made it easy to implement a way to filter desired matches to external file, therefore the test evaluator's job was to examine these files and calculate the recall and precision measures are readily available.

Ontos saves all the matching results to one ontology object, which is then used to generate a human-readable output in a form of HTML file. However, for any kind of statistical analysis, the test evaluator would have to either parse this file,

[2]All the expressions stated in this paragraph are only examples to illustrate the particular point, not the actual regular expressions. These can be found in the appropriate configuration files for each tool.

or to use the batch output provided in a form of XML file. The XML file is not as straightforward as it would seem though, since it contains data grouped by individual ontology properties and not by data instances to which they belong. Therefore the HTML file was parsed and recall measure was obtained from it. The precision measure had to be determined by manual examination of the HTML file and counting any incorrectly matched values (e.g. string "Student Agency" matched as a contact person's name).

Considering the actual numbers in the table, there are several specifics worth commentary. First of all, the "position name" was unmatchable using the Ontos tool – there was no context available for this information, since Ontos cannot handle multiline regular expression patterns. This was also the case with contact person, where the available context was ambiguous, resulting in a much lower precision.

SemJob, on the other hand, did not have any problems with precision – once a piece of information is matched it is almost certain that this match is correct. However, the matching approach consisting of dividing the input document into several sections and matching different properties only on a chosen section[3] takes its toll on the resulting recall values. This fact is most prominent with the "contact email" matching, as different job advertisements present the email in different sections (e.g. contact, offer, introductory text), but the configuration counts on this information to appear in the contact section only.

The final remark about the test results is that even though both compared tools were ontology-based and therefore should be able to extract information from any semi-structured or unstructured text, this is not entirely true. Both of these tools require the user to create the extraction ontology based on the previous knowledge about the input documents. The more initial knowledge the user has, the more detailed and efficient will the resulting ontology be during the actual extraction.

[3]The finer details of this matching process are described in the chapter 3

# Conclusion

To summarize this thesis, we have examined several data extraction tools chosen according to the general extraction approach they took. Nowadays, there are six major ways of solving the data extraction problem, namely using a specific language, using a wrapper induction algorithms, HTML-aware data extraction, NLP-based, modeling-based and ontology-based approach. Each category has at least two representatives in this survey, but the categorization is not definitive, as several tools utilize practices from multiple categories.

We have presented a short description for each of the implementations, where we briefly assessed its strengths and weaknesses. The tools were presented in order resulting from their membership in a given category. The form of extraction rules or the extraction rule learning algorithm were mentioned as well, so the reader can gain some basic understanding of how a given tool accomplishes the data extraction task.

Afterwards, we have presented our implementation of extraction module used in SemJob project, which was developed from scratch and without prior knowledge of the state of art applications. This project was designed to perform on a specific domain of semi-structured documents and this has had a significant influence on the extraction methods and capabilities. The extraction strategies used by SemJob were presented in fine detail.

At this point in the thesis, the reader should have a general knowledge of all the presented tools and can follow a theoretical featurewise comparison. In this part we have analyzed the performance of the extraction tools in several areas, such as the ease of use, the supported input format and the resilience to change of the input document structure. The qualitative analysis is concluded by a short study of how well can SemJob rival the other tools based on the presented characteristics. For quick reference, the tables A.1, A.2 and A.3 containing a condensed comparison are provided.

As a follow-up to the theoretical comparison, we have conducted a practical comparison. In this part of the thesis we have selected two tools, configured them and evaluated the results gained by executing their extraction algorithms. These results were then compared with the performance provided by our tool, SemJob. It is worth noting that each of the three tools required modifications in its source code to be able to partake in the testing (both the original and modified code are included on the disk accompanying this thesis). However, even after expending significant effort, one of the tools had proven to be insufficiently proficient for the chosen domain and had to be excluded from the qualitative testing. In the end, we are glad to conclude that the design of SemJob has allowed it to achieve solid results on the presented data corpus.

To conclude the findings discovered in both the above-mentioned comparisons, there is no universally good and correct way of performing the data extraction and therefore there is no clear “winner” among the presented tools (in both theoretical and practical comparison). Each category has its pros and cons in the real world deployment and there are always tradeoffs. If a tool performs a thorough analysis of the source text using lexical analysis and conceptual ontologies, it will inevitably perform slower than a tool extracting data based on structure of the HTML tags. Similarly, if a tool works automatically without any need for user intervention during the extraction process, it will be easier to use, but without the opportunity to customize the extraction rules or influence the extraction process in any way.

One important thing to remember when trying to use any of the presented tools is that all of them have been implemented as means of researching various data extraction techniques and therefore the user must be prepared for the lack of user-friendliness and polish he/she may expect from commercial applications. Therefore it is our future goal to create an improved version of SemJob, utilizing all the experience and knowledge acquired during the work on this thesis, which would provide the above-mentioned qualities.

# Bibliography

[1] Charles University web page [online], 2013-03-30 [cit. 2013-04-10]. http://www.mff.cuni.cz/toISO8859-2.en/fakulta/budovy/kampus/ mala_strana.htm.

[2] Grishman, Ralph, Sundheim, Beth, Message Understanding Conference - 6: A Brief History. *Proceedings of the 16th International Conference on Computational Linguistics (COLING)*, I, Kopenhagen, 1996, 466–471.

[3] Defense Advanced Research Projects Agency (DARPA) web page [online], [cit. 2013-04-10]. http://www.darpa.mil/

[4] Grishman, Ralph, Information Extraction: Techniques and Challenges. *In Information Extraction A Multidisciplinary Approach to an Emerging Information Technology*, edited by MariaTeresa Pazienza, 1299:10–27. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1997.

[5] Laender, Alberto H. F., Ribeiro-Neto, Berthier A., Da Silva, Altigran S, Teixeira, Juliana S. A brief survey of web data extraction tools. In: *SIGMOD Rec. June 2002. Vol. 31, no. 2*, pp. 84-93. DOI 10.1145/565117.565137.

[6] Crescenzi, Valter, Giansalvatore, Mecca. Grammars have exceptions. *Information Systems 23.8* (1998): 539-565.

[7] Atzeni, Paolo, Giansalvatore, Mecca, Masci, Alessandro, Sindoni, Giuseppe, Merialdo, Paolo. The Araneus Web-based Management System. *In ACM SIGMOD Record*, 27:544–546, 1998.

[8] Wikipedia disambiguation page for a search keyword "data" [online], 2013-03-12 [cit. 2013-04-10]. http://en.wikipedia.org/wiki/Data_(disambiguation).

[9] Chawathe, Sudarshan, Ireland, Kelly, Hammer Joachim, Garcia-Molina, Hector, Papakonstantinou, Yannis, Ullman, Jeffrey, Widom, Jennifer. The TSIMMIS Project: Integration of Heterogenous Information Sources. *In: Information Processing Society of Japan (IPSJ 1994)*, October 1994, Tokyo, Japan.

[10] Stanford InfoLab Publication Server [online], 2009-01-14 [cit. 2013-04-10]. http://ilpubs.stanford.edu:8090/66/

[11] TSIMMIS web page [online], 1998-04-04 [cit. 2013-04-10]. http://infolab.stanford.edu/tsimmis/tsimmis.html.

[12] Arocena, Gustavo O., Mendelzon, Alberto O., WebOQL: restructuring documents, databases and Webs, 1998.

[13] W3C web page for XML [online], 2012-01-24 [cit. 2013-04-10]. http://www.w3.org/XML/.

[14] W3C web page for RDF [online], 2013-03-02 [cit. 2013-04-10]. http://www.w3.org/RDF/.

[15] W3C web page for DOM [online], 2009-01-06 [cit. 2013-04-10]. http://www.w3.org/DOM/.

[16] SAHUGUET, Arnaud, AZAVANT, Fabien. Building intelligent web applications using lightweight wrappers. *Data & Knowledge Engineering 36.3* (2001): 283-316.

[17] International Movie DataBase web page [online], [cit. 2013-04-10]. http://www.imdb.com.

[18] LIU Ling, PU, Calton, HAN, Wei. XWRAP: An XML-enabled Wrapper Construction System for Web Information Sources. *In ICDE*, 611–621, 2000.

[19] CRESCENZI, Valter, GIANSALVATORE, Mecca, MERIALDO, Paolo. RoadRunner: Towards Automatic Data Extraction from Large Web Sites. 2001.

[20] GOLD, E. Mark. Language identification in the limit. *Information and Control*, 10(5), 1967.

[21] SLEIMAN, Hassan A., CORCHUELO, Rafael . TEX: An Efficient and Effective Unsupervised Web Information Extractor. *Knowledge-Based Systems 39* (2013): 109–123.

[22] SLEIMAN, Hassan A., CORCHUELO, Rafael . Towards a Method for Unsupervised Web Information Extraction. *In Web Engineering*, 427–430. Springer, 2012.

[23] KNUTH, Donald E., MORRIS, James H., PRATT , Vaughan R. Fast Pattern Matching in Strings. *SIAM Journal on Computing 6*, no. 2 (1977): 323–350.

[24] RILOFF, Ellen. Automatically Constructing a Dictionary for Information Extraction Tasks. *In Proceedings of the National Conference on Artificial Intelligence*, 811–811, 1993.

[25] LEHNERT, Wendy, CARDIE, Claire, FISHER, David, RILOFF, Ellen, WILLIAMS, Robert. University of Massachusetts: Description of the CIRCUS System as Used for MUC-3. *In Proceedings of the 3rd Conference on Message Understanding*, 223–233, 1991.

[26] HUFFMAN, Scott B. Learning Information Extraction Patterns from Examples. *In Connectionist, Statistical and Symbolic Approaches to Learning for Natural Language Processing*, 246–260. Springer, 1996.

[27] BRILL, Eric. A Simple Rule-based Part of Speech Tagger. *In Proceedings of the Workshop on Speech and Natural Language*, 112–116. HLT '91. Stroudsburg, PA, USA: Association for Computational Linguistics, 1992.

[28] KIM, Jun-Tae, MOLDOVAN, Dan I. Acquisition of Linguistic Patterns for Knowledge-based Information Extraction. *Knowledge and Data Engineering, IEEE Transactions On 7*, no. 5 (1995): 713–724.

[29] Soderland, Stephen, Fisher, David, Aseltine, Jonathan, Lehnert, Wendy. CRYSTAL: Inducing a Conceptual Dictionary. *arXiv Preprint Cmp-lg*/9505020 (1995).

[30] Soderland, Stephen. Learning to Extract Text-based Information from the World Wide Web. *In Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*, 251–254, 1997.

[31] Soderland, Stephen, Aronow, David, Fisher, David, Aseltine, Jonathan, Lehnert, Wendy. Machine Learning of Text Analysis Rules for Clinical Records. *TE-39: University of Massachusetts, Center for Intelligent Information Retrieval Technical Report* (1995).

[32] Soderland, Stephen, Cardie, Claire, Mooney, Raymond. Learning Information Extraction Rules for Semi-structured and Free Text. *In Machine Learning*, 233–272, 1999.

[33] Muslea, Ion, Minton, Steven, Knoblock, Craig A. Hierarchical Wrapper Induction for Semistructured Information Sources. *Autonomous Agents and Multi-Agent Systems 4, no. 1–2* (2001): 93–114.

[34] Muslea, Ion, Minton, Steven, Knoblock, Craig A. A Hierarchical Approach to Wrapper Induction. *In Proceedings of the Third Annual Conference on Autonomous Agents*, 190–197, 1999.

[35] Kushmerick, Nicholas. Wrapper Induction: Efficiency and Expressiveness. *Artificial Intelligence 118*, no. 1–2 (2000): 15–68.

[36] Hsu, Chun-nan, Dung, Ming-Tzung. Generating Finite-State Transducers For Semi-Structured Data Extraction From The Web, 1998.

[37] Adelberg, Brad. NoDoSE - A Tool for Semi-Automatically Extracting Structured and Semistructured Data from Text Documents. *In SIGMOD Record*, 283–294, 1998.

[38] Laender, Alberto H. F., Ribeiro-Neto, Berthier A., Da Silva, Altigran S. DEByE – Data Extraction By Example. *Data & Knowledge Engineering 40, no. 2* (February 2002): 121–154.

[39] Embley, David W., Campbell, D. M., Jiang, Y. S., Liddle, S. W., Lonsdale, D. W., Ng, Y.-k, Liddle, S. W., Smith, R. D. Conceptual-Model-Based Data Extraction from Multiple-Record Web Pages. *Data & Knowledge Engineering 31* (1999): 227–251.

[40] Data Extraction Research Group web page [online], [cit. 2013-04-10]. http://www.deg.byu.edu/.

[41] Data Extraction Research Group Demos web page [online], [cit. 2013-07-21]. http://dithers.cs.byu.edu/deg/demos/ontosmx/simple-demo.php.

[42] Gruber, Thomas R. A translation approach to portable ontology specifications. *Knowledge Acquisition 5 (June 1993)*: 199–220.

[43] Luger, George F. Artificial Intelligence: Structures and Strategies for Complex Problem Solving. *Addison Wesley Longman*, 2005.

[44] Embley, David W., Park, Joseph S. Extracting and Organizing Facts of Interest from OCRed Historical Documents.

[45] Apache Jena reasoner project web page [online], [cit. 2013-04-10]. http://jena.apache.org/.

[46] Garshol, Lars Marius. Duke deduplication engine web page [online], [cit. 2013-04-10]. https://code.google.com/p/duke/.

[47] Betík, Roman, Helešic, Tomáš, Hoferek, Ondrej, Kóša, Peter, Lukšová, Ivana. SemJob, software project, Faculty of Mathematics and Physics, Charles University in Prague, 2012.

[48] TreeX - Highly Modular NLP Framework [online], 2012 [cit. 2013-07-17]. http://ufal.mff.cuni.cz/treex/.

[49] Protégé web page [online], [cit. 2013-07-21]. http://protege.stanford.edu/.

# List of Abbreviations

CN - Concept Node  
DOM - Document Object Model  
DEG - Data Extraction Group  
ECT - Embedded Catalog Tree  
FP - Frame-Phrasal pattern  
FST - Finite-State Transducer  
GUI - Graphical User Interface  
HTML - HyperText Markup Language  
IE - Information Extraction  
JD - Job Description  
NLP - Natural Language Processing  
NT - Non-terminal symbol  
OEM - Object Exchange Model  
OEP - Object Extraction Pattern  
POS - Part Of Speech  
RDF - Resource Description Framework  
XML - eXtensible Markup Language

# List of Tables

# List of Figures

# A. Comparison tables

| Tool features | | Tools | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | Minerva | TSIMMIS | WebOQL | W4F | XWRAP | RoadRunner | TEX |
| Pre-processing | NLP | no | no | no | no | no | no | no |
| | additional tags | no | no | no | no | no | no | no |
| | HTML validity | no | no | no | yes | yes | yes | no |
| Input | HTML | yes | yes | yes | yes | yes | yes | yes |
| | XML | yes | yes | no | no | no | no | yes |
| | plain text | no | no | no | no | no | no | no |
| | semistructured text | no | no | no | no | no | no | no |
| Extraction rule creation | user coded | yes | yes | yes | yes | no | no | no |
| | user specified (GUI) | no | yes | no | yes | yes | no | no |
| | induced | no | no | no | no | no | yes | yes |
| Training examples | need | no | no | no | yes | yes | yes | no |
| | ease of creation | N/A | N/A | N/A | ● ● ○ | ● ● ○ | ● ● ● | N/A |
| Complex objects | varations | yes | yes | yes | yes | no | yes | yes |
| | nested | yes | yes | yes | yes | yes | yes | yes |
| Degree to automation | | ○ ○ ○ | ○ ○ ○ | ● ○ ○ | ● ● ○ | ● ● ● | ● ● ● | ● ● ● |
| Resilience to structure change | | no | no | yes | no | no | yes | yes |
| Provides GUI | | no | yes | no | yes | yes | no | no |
| XML Output | | yes | no | no | yes | yes | no | yes |
| Overall ease of use | | ● ○ ○ | ● ○ ○ | ● ○ ○ | ● ● ○ | ● ● ● | ● ● ● | ● ● ● |
| Implementation language | | Minerva, Editor | TSIMMIS | WebOQL | Java, HEL | Java | Java | CEDAR |
| Published | | 1998 | 1997 | 1998 | 2000 | 2000 | 2001 | 2012 |

Table A.1: Comparative table of individual tools' features

| Tool features | | Tools | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | AutoSlog | LIEP | PALKA | CRYSTAL | WHISK | WIEN | SoftMealy |
| Pre-processing | NLP | yes | no | yes | yes | yes | no | no |
| | additional tags | no | no | no | no | yes | no | no |
| | HTML validity | no | no | no | no | no | no | no |
| Input | HTML | no | no | no | yes | yes | yes | yes |
| | XML | no | no | no | no | yes | yes | yes |
| | plain text | yes | yes | yes | yes | yes | no | no |
| | semistructured text | yes | yes | yes | yes | yes | no | yes |
| Extraction rule creation | user coded | no | no | no | no | no | no | no |
| | user specified (GUI) | no | no | no | no | no | no | no |
| | induced | yes | yes | yes | yes | yes | yes | yes |
| Training examples | need | yes | yes | yes | yes | yes | yes | yes |
| | ease of creation | ● ○ ○ | ● ○ ○ | ● ○ ○ | ● ○ ○ | ● ○ ○ | ● ○ ○ | ● ● ○ |
| Complex objects | variations | yes | yes | yes | yes | no | no | yes |
| | nested | no | no | no | no | no | no | yes |
| Degree to automation | | ● ● ○ | ● ● ○ | ● ● ○ | ● ● ○ | ● ● ○ | ● ● ○ | ● ● ○ |
| Resilience to structure change | | yes | yes | yes | yes | no | no | no |
| Provides GUI | | no | yes | no | no | no | no | yes |
| XML Output | | no | no | no | no | no | no | no |
| Overall ease of use | | ● ○ ○ | ● ● ○ | ● ○ ○ | ● ○ ○ | ● ○ ○ | ● ○ ○ | ● ● ○ |
| Implementation language | | N/A | N/A | N/A | N/A | Perl | N/A | Java |
| Published | | 1993 | 1996 | 1995 | 1995 | 1999 | 2000 | 1998 |

Table A.2: Comparative table of individual tools' features

| Tool features | | Tools | | | | | |
|---|---|---|---|---|---|---|---|
| | | STALKER | NoDoSE | DEByE | OntoES | FROntIER | SemJob |
| Pre-processing | NLP | no | no | no | no | no | no |
| | additional tags | yes | no | no | yes | yes | yes |
| | HTML validity | no | no | no | no | no | no |
| Input | HTML | no | yes | yes | yes | yes | yes |
| | XML | no | yes | yes | yes | yes | yes |
| | plain text | yes | no | no | yes | yes | yes |
| | semistructured text | yes | yes | yes | yes | yes | yes |
| Extraction rule creation | user coded | no | no | no | yes | yes | yes |
| | user specified (GUI) | yes | yes | yes | no | no | no |
| | induced | yes | yes | yes | yes | yes | no |
| Training examples | need | yes | yes | yes | no | no | no |
| | ease of creation | ● ○ ○ | ● ● ● | ● ● ● | N/A | N/A | N/A |
| Complex objects | variations | yes | yes | yes | yes | yes | yes |
| | nested | yes | yes | yes | yes | yes | yes |
| Degree to automation | | ● ● ○ | ● ● ○ | ● ● ○ | ● ○ ○ | ● ○ ○ | ● ○ ○ |
| Resilience to structure change | | no | no | no | yes | yes | yes |
| Provides GUI | | no | yes | yes | no | no | no |
| XML Output | | no | yes | yes | no | yes | yes |
| Overall ease of use | | ● ● ○ | ● ● ○ | ● ● ○ | ● ○ ○ | ● ○ ○ | ● ○ ○ |
| Implementation language | | N/A | Java | Java | Perl, Java, C++ | N/A | Java |
| Published | | 1999 | 1998 | 2002 | 1999 | 2013 | 2012 |

Table A.3: Comparative table of individual tools' features

# B. Content of the attached CD

The attached CD contains following materials:

- `data/` - all the data used for the testing (various formats).
- `semjob/` - original and modified source code for SemJob. Also contains multiple documentation documents.
- `whisk/` - original source code for WHISK.
- `ontos/` - original and modified source code for Ontos.
- `results/` - results of the practical comparison.
- `Peter_Kosa_master_thesis.pdf` - the text of this thesis.
- `README.txt` - a short description of the CD contents.
- `contact.txt` - author's credentials.