Jedi: Extracting and Synthesizing Information from the Web

Gerald Huck, Peter Fankhauser, Karl Aberer, Erich Neuhold
GMD - German National Research Center for Information Technology
Integrated Publication and Information Systems Institute IPSI
Dolivostr. 15, 64293 Darmstadt, Germany
{huck, fankhaus, aberer, neuhold}@darmstadt.gmd.de

Abstract

Jedi (Java based Extraction and Dissemination of Information) is a lightweight tool for the creation of wrappers and mediators to extract, combine, and reconcile information from several independent information sources. For wrappers it uses attributed grammars, which are evaluated with a fault-tolerant parsing strategy to cope with ambiguous grammars and irregular sources. For mediation it uses a simple generic object-model that can be extended with Java-libraries for specific models such as HTML, XML or the relational model. This paper describes the architecture of Jedi, and then focuses on Jedi's wrapper generator.

1. Introduction

The World Wide Web has evolved into a general purpose information space that consists not only of hyperlinked documents, but also of queryable information sources delivering semi-structured data. Product catalogues, ticker services, weather reports, annotated bibliographies, software directories, conference-announcements are only a few examples, where structured information is forced into the browse&display oriented paradigm of the Web.

Such sources often contain overlapping or complementary information. However, they express this information with different syntax and semantics. To satisfy integrated information needs, such as finding the cheapest hand-held computer from several online-merchants, or compiling a table of weather forecasts per city from a number of regional providers, one needs to extract and synthesize information from several sources.

This paper describes Jedi (Java based Extraction and Dissemination of Information), a light weight wrapping and mediation tool to reuse, combine, and reconcile information from several independent information sources. Jedi provides a fault-tolerant parser to extract data from external sources, an extensible object model to describe structure and semantics of heterogeneous sources uniformly, and a flexible query and manipulation language to realize integrated views on multiple sources.

The remainder of this paper is organized as follows: Section 2 presents the overall architecture. Section 3 introduces Jedi's extraction language. Section 4 describes its fault-tolerant parsing strategy. Section 5 illustrates the use of Jedi for wrapping an online-product catalogue. Section 6 discusses related work, and Section 7 concludes the paper.

2. Architecture

This section gives a brief overview over Jedi's flexible integrated wrapping and mediation architecture, depicted in Figure 1.

The wrapping layer transforms heterogeneously modeled sources into a uniform object model. We distinguish between generic wrappers and specific wrappers. Generic wrappers can be defined independently of the schema of sources, such as relational DBMS, CORBA systems or also document sources with a well defined exchange format such as XML and HTML. Such wrappers can be realized by extending Jedi's object model with external libraries.

Specific wrappers for sources with irregular and proprietary format such as online product catalogues, or bibliographies need to take into account the schema of a source. For the construction of such wrappers, Jedi supports an extraction specification language based on attributed gram-

mars, evaluated by a fault tolerant parser.

The mediation layer can be used to define and query integrated views on multiple sources. Such views can relate data from multiple sources, and homogenize the structure and semantics such that the user can interact with an integrated view like with a single source. This layer offers also functionality to graphically interact with integrated views and to flexibly present results.

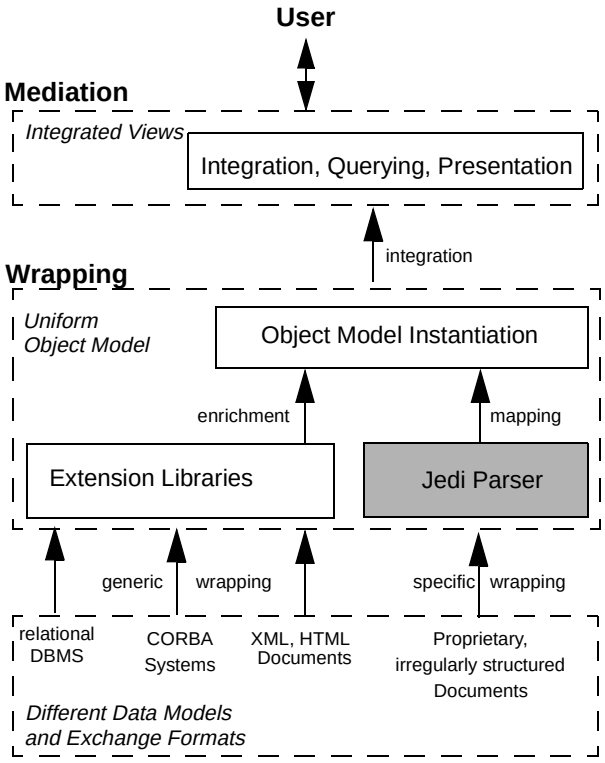

Figure 1: Jedi Architecture

In the remainder of this paper we will describe in detail Jedi's approach to generating specific wrappers for irregular sources.

3. Extraction Language

The two most widely used approaches to extract information from documents are pattern matching, e.g. [9], and parsing with grammars, e.g. [12]. Often these two approaches are combined, with patterns used to map the concrete syntax of a source to an abstract syntax - tokenization - and grammars used to generate a structured representation from the sequence of tokens.

Patterns are usually expressed by means of regular expressions. When applied to a document, they typically serve as filters, such that document portions that do not match any pattern are discarded automatically. Thus one can focus on the interesting portions of a document, and need not be concerned with irregularities of the irrelevant portions.

However, the expressive power of patterns is limited. Essentially, a set of patterns can only describe the structure of a document as a flat set of objects. When the interpretation of patterns depends on their actual sequence or on their nesting structure, patterns alone do not suffice.

As a very simple example consider the following document from which a list of articles, their price and available quantity should be extracted:

```
The following articles are on stock (4/7/98):
Article         Price/$          Quantity
Psion Computer  560              23
8MB memory      100 bargain!     15
```

To distinguish between numbers as a price or a quantity, we need to explicitly specify that the first number pattern in each line matches the price and the second number pattern matches the quantity.

For this purpose we need grammars that allow to use sequence and nesting structure of documents to interpret patterns according to their syntactic context. With a simplified syntax the above example thus could be matched with the following expression:

```
<string><blanks><number><blanks><number>
```

Note however, that this expression only matches the first line. The string "bargain!" between price and quantity in the second line is not anticipated by the above specification. Thus with the conventional brittle interpretation of grammars, the second line would not match. One could explicitly trap this irregularity by extending the expression to:

```
<string><blanks><number><blanks>
[<string><blanks>]<number>
```

But anticipating all such irregularities soon results in very complex grammars. To avoid this we need to combine the flexibility of pattern matching with the higher expressive power of grammars. Like patterns, grammars should not normatively describe the structure of a document, but serve as filters that automatically discard irrelevant portions. This can be accomplished by the parsing strategy introduced in Section 4.2.

In the following, we introduce briefly Jedi's language to specify the syntactic structure of sources, to associate semantic predicates for further disambiguation, and to map source structure to a goal structure.

3.1. Syntactic Source Structure

Jedi uses context free grammars to describe the syntactical structure of textual sources. The grammars consist of rules having a name and a body which contains the production expression for the rule.

Composition operators can be used to construct more

complex productions from simpler ones. As usual, these comprise sequences, optional production, optional repetition, repetition, alternatives and grouping:

```
E1 E2 E3  // sequence of 'E1',' E2' and 'E3'
E?        // optional 'E'
E*        // optional repetition of 'E'
E+        // repetition of 'E'
E1 | E2   // 'E1' or 'E2'
(E)       // grouping of 'E'
```

Terminal productions match characters of the source. These comprise single character productions that match any character or character sets in a regular expression like syntax and productions for character sequences:

```
.     // any character
[A-Z]// uppercase letters A to Z
[^A-Z]// any character but A to Z
'text'// the character sequence 'text'
```

Non printable characters can be defined by using backslash notation, e.g. '\t','\n','\r'.

Finally, rule productions may refer recursively to themselves or to other rule definitions. The following gives a small grammar that detect dates in a source:

```
rule day   is [0-9]+ end
rule month is [0-9]+ end
rule year  is [0-9]+ end

rule date is
   month() '/' day() '/' year()
end

rule dates is
(
   .+ // skip anything which is not a date
   | date()
)+
end
```

In contrast to conventional grammar specification tools, e.g. lex and yacc, Jedi does not require separate specifications for a scanner to tokenize the source and a parser that operates on the tokenized source. When using grammars for extracting information from irregular electronic documents, the interpretation of tokens, such as delimiters, tags, or keywords, depends on their syntactic context.

Another noteworthy feature of Jedi's use of grammars is its robustness with respect to ambiguity. Whereas tools for parsing formal languages, such as yacc, require to carefully avoid or limit ambiguities, Jedi tolerates highly ambiguous grammars with arbitrarily overlapping rules. For example, the production ".+" in the sample grammar above accepts automatically only non numeric characters that can not be accepted by the rule for date.

3.2. Semantic Predicates

For performing semantic checks on the syntactically matched data, Jedi supports predicates that can, for example, check for number ranges or lookup a thesaurus.

Predicates can be defined by embedding arbitrary rule productions between 'accept' and 'if' keywords. Whenever the embedded production matches the source, the predicate expression[1] following the 'if' is checked against the matched data passed as a special parameter '$$'. If the predicate expression evaluates to true, the production match is accepted, otherwise it is rejected.

The following example illustrates the use of predicates to disambiguate between street numbers and zipcodes of addresses. Because both are defined by equivalent structural descriptions (`[0-9]+`), and both may be missing, they can neither be disambiguated by the local pattern nor by the syntactic sequence. Thus we need predicates that constrain street numbers to a length smaller then 5 and (German) zipcodes to be of length 5:

```
rule address is
   // ... - some rules omitted for simplicity
   streetname()
   (
      accept
         [0-9]+ // pattern for street numbers
      if $$.length<5;
   )?
   .* // skip non relevant characters
   (
      accept
         [0-9]+ // same pattern for zipcode
      if $$.length == 5;
   )?
   .* // skip non relevant characters
   cityname()
end
```

3.3. Filling Goal Structure

Rules and predicates allow to determine the source structure of a document. In the following, we introduce the extraction language concepts to map source structure to a goal representation. Such mappings can discard arbitrary portions of the source, transform sequence and nesting level of the source, add content such as XML tags, or also generate in-memory objects.

Some of these transformations can be simply achieved by mapping the segmented source to a fixed format, such as XML. This approach we have used in Dream [8], the predecessor of Jedi. But most of the transformations require more

1. Predicate expressions are given in Jedi's scripting language described in the next section.

flexible mechanism. Thus, Jedi adopts a simple grammar attribution approach. Source data matched by productions can be assigned to variables which can be further processed by code blocks.

Jedi offers two different assignment operators. The simple assignment operator ('=') assigns source data matched by its right hand side production to a variable. The accumulative assignment operator ('+=') collects all matched data into a sequence and assigns it to the variable. This operator is mostly used in repetition productions.

Variables can be further processed in code blocks which contain statements of Jedi's scripting language. The features of this scripting language needed in the context of this paper are object creation, calls to external functions and predicates. In addition to these features, the scripting language supports standard control statements, multithreading, exception handling and offers an embedded OQL like query language. The underlying object model is self descriptive and supports querying of structure and type of objects. The built-in data modeling facilities can be extended by Java classes to add new data types or to integrate data models generically.

Rules can optionally declare a result variable to return the object created in the code block. Without a return variable, rules return the source string matched by their production.

The following example extends the date grammar introduced with assignments and code blocks to create a fully functional wrapper. This wrapper extracts dates from sources, maps them onto XML element objects and finally generates well-formed XML output from the created objects. The XML element type used is part of an extension library implemented in Java which offers data types to represent XML documents in the document object model proposed by the World Wide Web Consortium (W3C) [16].

```
rule day   is [0-9]+ end
rule month is [0-9]+ end
rule year  is [0-9]+ end

rule date : res
   // declares res as result variable
is
   m=month() '/' d=day() '/' y=year()
      // assign matched data portions
      // to variables m,d,y
   do
      // definition of result in code block
      res = xml_element("date");
      res.addChild(xml_element("month",m));
      res.addChild(xml_element("year",y));
      res.addChild(xml_element("day",d));
   end
end
```

```
rule dates is
   (
      .+ // skip anything which is not a date
      |
         list += date()
         // rule variable list holds
         // a sequence of XML 'date' elements
         // returned by rule date.
   )+
   do
      res = xml_element("dates");
      res.addChildren(list);
      res.prettyPrint();
   end
end
```

4. Parsing Strategy

Grammars describe source structure declaratively, but they do not describe their interpretation. As we will demonstrate in this section, often a grammar can have many different interpretation, i.e. they are ambiguous. Existing grammar based parsing approaches avoid or limit ambiguity by imposing constraints on the grammar specifications, e.g. LL grammars (PCCTS) or LALR grammars (yacc).

Such constrained grammars work well for programming languages but lack the flexibility do deal with irregular documents.

Ambiguity can be a very powerful tool to specify grammars that can deal with unanticipated structural deviations in documents and to focus only on the relevant portions of a document for extraction. As an example, the following grammar can be used to extract numbers from a source.

```
(num=[0-9]+ | .+)*
```

A conventional parser would not allow for such a specification because ".+" subsumes "`[0-9]+`", requiring e.g. to exclude explicitly digits from the pattern ".+".

To resolve such ambiguities we need to be able to explore all possible solutions, and a mechanism to choose the 'best' possible one. In our example, such a 'best' solution should match all digits with the number pattern and all other characters with the ".+".

In the following we describe first the solution space for ambiguous grammars, propose a specificity criterion used to rank possible grammar interpretations, and show how the introduced principles can be used to achieve fault tolerant parsing. Then, we introduce our parser model and describe the parsing algorithm which realizes a novel breadth first search based parsing strategy. Finally, we discuss some implementation aspects and the complexity of the algorithm presented.

4.1. Ambiguous Grammars

Solution Space

The solution space for ambiguous grammars comprises all possible interpretations of a grammar for a given source.

Each of these solutions can be described by a unique path through the grammar containing the terminal productions that match a character at a certain source position. The following example gives a source, grammar and all possible grammar interpretations:

Source: “abb”
Grammar: ’a’ (’b’ | .)+

Solution paths:

’a’ . .
’a’ . ’b’
’a’ ’b’ ’b’
’a’ ’b’ .

Selecting an Interpretation

Each possible grammar interpretation can lead to a different result, i.e. the data portions extracted and code blocks executed by each solution can vary according to the path described by each solution. To choose among these solutions we need a ranking: Each solution path consists of a sequence of accepting transitions. Each transition accepts a certain character set. Single transitions are ordered ascending according to the cardinality of the accepted character set. Paths are ordered by comparing their transitions lexicographically according to this ranking.

For the example above, the terminals ’a’ and ’b’ have specificity 1 whereas the dot has specificity 256 (all Latin-1 characters) and thus the resulting ranking of solutions is:

’a’ ’b’ ’b’ => this is the ’best’ solution
’a’ ’b’ .
’a’ . ’b’
’a’ . .

The ranking is strictly monotonous. Thus, it allows for early disambiguation, i.e., a partial solution which is more specific than another partial solution at any position in the document will never get less specific by regarding the rest of the document.

This ranking has been chosen as it selects a grammar interpretation which is close to user expectations. Regard the following small example grammar “([0-9]+ | .+)*”. Apparently, there exists no objective criterion to prefer the number pattern over the “.+”. However it appears natural to interpret this grammar in such a way that it describes sources which consist repeatedly of numbers or ’anything else’, because a number is more specific than ’anything’.

We have also tested two other specificity rankings. The first one also uses lexicographic comparison of solutions but ranks those solutions higher whose transitions are more specific towards the end of the solution path. The second ranking orders paths according to the sum of their terminal specificities. Both rankings however often lead to ’surprising’ grammar interpretations and thus have been rejected.

Fallbacks

One particularly important use of “.+” is to serve as a fallback rule which matches portions of a document for which no more specific rule exists.

To free the user from explicitly considering such fallbacks, Jedi automatically includes them with every rule[2]. To distinguish such fallbacks from explicitly given “.*” productions, they get an artificial lowest specificity. This guarantees that a fallback can only show up in a final solution if no other user defined production can match the source at a given point, and consumes only as much characters as needed until a production in the direct grammar context of the fallback can match again.

4.2. Parsing Ambiguous Grammars

Grammar Representation

Context free grammars can be represented as non deterministic finite stack automatons (NDSA). Jedi uses an optimized NDSA which is generated on the fly from grammar rule specifications whenever matching of a rule against a source is requested. This allows any rule to serve as an entry point for a grammar as opposed to yacc for example where only one dedicated start rule is allowed.

In a first step, an initial automaton is created which consists of state nodes combined by transition edges. Transitions can either accept characters or are ε-transitions. Various kinds of ε-transitions model assignments, code blocks and predicate evaluation.

In a second step, the automaton is transformed further. Starting from the start state, all outgoing ε-transitions are followed recursively until an accepting transition is reached. Then, the whole transition path followed so far is collapsed into a new transition accepting the same characters as the accepting transition reached. The attributes of the original path are also included. This proceeds recursively until every state has been visited. Finally, all initial transitions and unneeded intermediate states are removed.

The final automaton then consists only of accepting transitions which optimizes the parsing algorithm as it need not repeatedly traverse ε-transition edges to find accepting transitions.

Figure 2 gives the initial automaton and transformed automaton for the grammar “’a’* .* ’b’+”:

2. This behaviour may be overridden in Jedi by preceding rule definitions with the keyword “strict”.

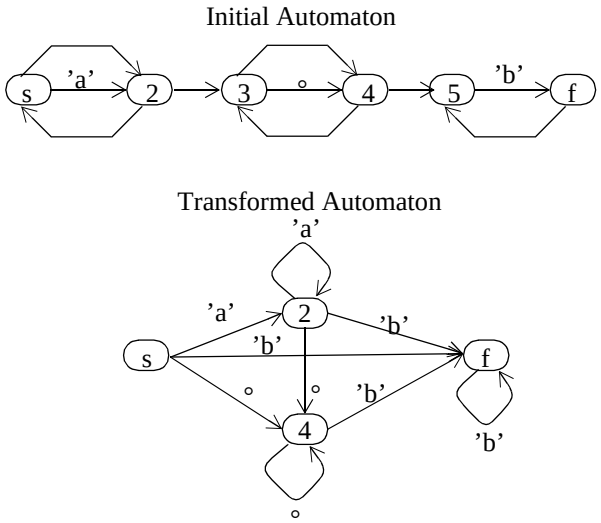

Figure 2: Automaton Transformation

Parsing Strategy

Parsing with ambiguous grammars needs to efficiently explore and rank huge solution spaces. The solution space can be explored either by using a depth first search or a breadth first search based algorithm.

A depth first search based algorithm requires exponential time in the worst case. Such an algorithm iterates over the source from the first to the last character. At each position, it follows the most specific matching transition through the automaton and goes to the next source position. If no matching transition exists, the algorithm uses backtracking and returns to a previous document position where it follows the next, less specific matching transition.

The backtracking requires exponential runtime in the worst case. Consider the following example. The source "aaaaaaaaaaaaaaaaaa$" and grammar "('a' | [a-z] | [a-z0-9])+ | .*" requires 3^{21} backtracking tests by this algorithm before it enters the solution path described by ".*".

Furthermore, several matching transitions can have equal specificity at any document position. The algorithm however cannot decide in advance which will get more specific in the future. Thus, it has in any case to backtrack to this position and has to evaluate all paths originating from equally specific transitions to determine the most specific solution. Again, this leads to exponential runtime in the worst case.

Therefore, we use a breadth first search based algorithm to explore the solution space and use the dynamic programming principle [5] to prune partial solutions that can never make the best solution.

The breadth first search initializes a first generation of solutions comprising all outgoing transitions of the start state which accept the first character of the source and ranks them according to the specificity of the accepting transitions.

Then, the breadth first search determines at each character in the document the next accepting transitions. This is done for each existing solution, starting traversal from the solution's last target state. For each accepting transition of a solution, a new solution is created by appending the transition to the already existing solution path. This is done in the order of increasing character positions and for each position in the order of decreasing lexicographic rank of existing solutions.

In order to preserve the lexicographic ranking, the new solutions have to be sorted. The breadth first search enforces already that new solutions which derive from equally ranked predecessor solutions are presorted and thus need only be sorted within the groups according to the actual specificity of the accepting terminal rule.

The dynamic programming principle is used to discard those new solutions which are candidates, but do not have a chance in giving a best solution. The criterion used here is that a solution which reaches the same automaton state and is in the same stack state (recursion!) already reached by another, more specific solution can be safely discarded as the future of both solutions is the same[3].

Given the source "abb" and the automaton from the previous section, Figure 3 illustrates our algorithm:

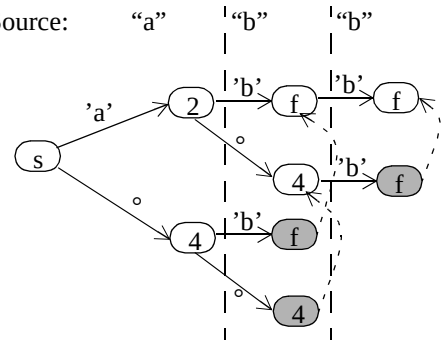

Figure 3: Solution Paths and Pruning

The vertical order of solution paths corresponds to the lexicographic ranking, the grayed states denote the solutions that can safely be discarded because of the more specific solution pointed to by the dotted arrows.

Predicates are modeled as follows. Special transition flags mark the entry and exit of predicate embedded automaton sections. Whenever such an enter transition accepts a character, the character's position is put on the current solution's stack. Whenever an exit transition accepts a character, the position is popped from the current solution's stack and the string embedded between current position and popped position is given to the predicate expression for

3. This dynamic lexicographic approach has been applied first to mating of sows [7].

evaluation. Only if the predicate returns true, a new solution is created.

There exist still some possibilities to optimize the presented algorithm as it follows a rather pessimistic strategy. It often keeps and creates candidate solutions that originate again and again from unspecific ambiguous rules in the grammar even if a good specific solution has already reached a more meaningful point in the grammar. Therefore, we added a prune operator. It allows users to specify a grammar location at which the system simply prunes all candidate solutions which are less specific than the one that passed this point. However, optimizations like these should be used cautiously as they require a deep understanding of the parsing algorithm in order to obtain the same results as not using them.

4.3. Implementation Aspects

The Solution Stack

As described above, we represent context free grammars by NDSAs. Normally, modeling the stack for such an NDSA is trivial but our breadth first search approach requires a separate stack for each partial solution.

Copying stacks for each solution on demand would require too much overhead. New solutions derived from the same predecessor solution can however share the predecessor stack. This can be modeled efficiently by a multi stack data structure based on a reversed tree representation. This tree does not refer from the root to its children but the children refer to the root. So, each solution sees the stack it requires and is free to manipulate it without affecting other solutions.

Assignment and Code Block Execution

The second aspect of our implementation concerns execution of assignments and code blocks associated to grammar rules. Executing these directly for each partial solution is neither desirable nor advisable due to possible side effects between solutions and runtime overhead to execute code for intermediate solutions that never give the final solution. Therefore, we postpone execution until the final solution has been found by the algorithm. This comes at the cost that we have to keep in the worst case the entire transition paths for each partial solution in order to be able to reconstruct assignments and code execution.

As a consequence, our tool cannot be used for continuous parsing problems, e.g. needed to analyse log files that are written in parallel by another program.

Compilation

Conventional parser tools, e.g. yacc or PCCTS, generate parser code which has to be compiled before the parser can be used. Our strategy does not gain much from compilation as it is dominated by the always required breadth first search iteration loop and subsequent sorting and pruning tests.

Furthermore, interpreting grammars directly offers the advantage of better maintenance and incremental development of grammars. This is important for Web sources which frequently change structural aspects of their information and thus require changes to the extraction specifications.

Debugging

The complete knowledge about the parsing process is encapsulated in the final solution's transition path and can be used for debugging purposes. Jedi offers a debugging mode which uses this knowledge to generate an HTML document from a parsed source which contains different markup for those source portions which have been matched by rules, fallbacks or have been assigned to variables.

4.4. Complexity

The worst case runtime complexity of our algorithm depends on the length l of the document and number n of transitions in the transformed automaton. Using breadth first search ensures that each character is only regarded once, and thus the algorithm is linear with respect to the length of the document. In each breadth first iteration step, a maximum number of n solutions (for each transition one) can survive and each of these requires in the worst case n pruning tests (all transitions can be reached from each solution and match). This leads to an overall worst case complexity of $O(n^2 l)$.

The average complexity is roughly $O(nl)$ as a solution normally does not have more than 2 or 3 possible successor solutions which reduces the n^2 pruning tests down to $3n$. Some experiments based on realistic grammars/documents confirm this assumption. The number of surviving solutions normally ranges between 10 and 25 and in each step the amount of existing solutions is doubled and needs to be reagrded for pruning. These numbers correspond to 10000 - 25000 characters/sec that can be parsed on a SPARC Ultra.

As described in the previous section, the worst case space requirements depend linearly on the length l of a source and surviving solutions n, if for each solution the whole transition path must be kept. Concrete memory requirements varied between 25 - 200 byte/character, depending on grammar and source.

However, these complexity considerations do only hold if we assume grammars without predicates and recursion. The following two grammars show that the algorithm runtime can explode by using predicates and recursion. The first example grammar calls the predicate for each partition of substrings in a given source:

```
.*
accept .* if predicate($$);
.*
```

The second example uses recursion. For each prefix substring matched by the ".+" in the example, the rule must be evaluated recursively and initiates the same segmentation process for the rest of the document again. If the source length is *l*, (*l*-1)! recursive evaluations of the rule are performed:

```
rule call_me_often is
   .+ call_me_often()
end
```

However, such pathological examples have never occurred in the experiments and applications that we have performed with Jedi so far.

5. Example

In the following we illustrate Jedi's parsing strategy along a realistic example taken from a demo located at "http://www.darmstadt.gmd.de/oasys/projects/jedi/index.html".

The online demo shows Jedi's facilities to extract, model and integrate PSION palmtop computer related product data from multiple Web sources, and to query and visualize the extracted data.

Figure 4 presents a screenshot of one source[4] which is highly irregular, mixing images, natural language text, forms and the relevant product data arbitrarily.

The code fragment depicted below the figure is the complete Jedi specification needed to define a grammar to extract article codes, article descriptions and their price from this source. Other than in the online demo, the extracted data is not mapped onto an object model, but directly rewritten as tagged XML source.

The first rule 'Article' specifies the source structure of one article 'record' and assigns appropriate data portions to the variables 'code', 'description' and 'price'. These are reused in the code block to write tagged XML code to stdout.

The first two assignment productions can safely contain a trailing ".*" which is automatically left whenever the more specific productions '</B>' or '<B>' match.

The third assignment production either requires a specific pattern which identifies exactly the price or it requires a more specific end tag to indicate where the price ends, e.g.:

```
'<B>'price = ('$' .*) '</B>'
```

The second rule 'ArticleList' describes that the source structure comprises a sequence of 'Article' records as described by the first rule.

As can be seen from the screenshot, this rule does not describe exactly the source as it contains a lot of additional, irrelevant information that must be filtered out.

4. located at http://www.mplanet.com/cgi/Web_store/web_store.cgi?page=psion.html&cart_id=2726135.4533

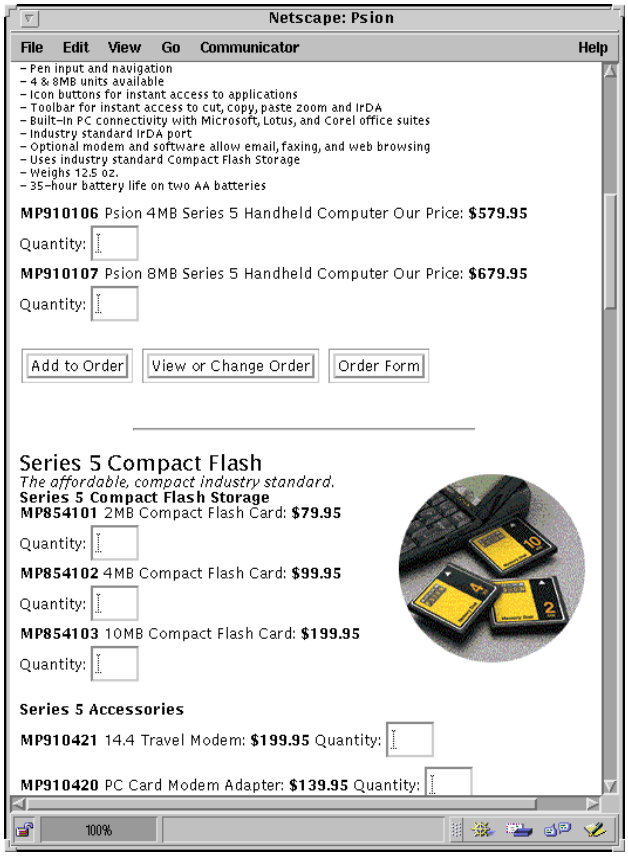

Figure 4: Snapshot of Source

```
rule Article is
   '<B>'   code = ('MP' .*)
   '</B>'  description = .*
   '<B>'   price = ('$'[0-9.]+)
   do
      println(
         "<Article>",
         "<Code>", code, "</Code>",
         "<Price>", price, "</Price>",
         "<Description>", description,
         "</Description></Article>"
      );
   end
end

rule ArticleList is
   do
      println("<ArticleList>");
   end
   (list += Article())+
   do
      println("</ArticleList>");
   end
end
```

Strict parsing approaches will fail when given such a grammar. Jedi however is able to proceed meaningfully. Its fault tolerant interpretation of the 'Article' rule allows to skip irrelevant portions of the source by the fallback production associated to the rule.

Finally, the embedded code blocks are evaluated according to the grammar interpretation described by the most specific solution path. Code execution will start with the first 'println' statement of rule 'ArticleList', it proceeds with the assignments and code defined in rule 'Article' as often as this rule has matched and ends with executing the second 'println' statement in rule 'ArticleList'. The portions accepted by fallbacks do not have any side-effects and thus do not cause any output to be written.

6. Related Work

Jedi's approach to extraction is based on the work described in [8]. Jedi extends this approach with the support for context free grammars, disambiguation predicates, flexible grammar attribution, and the generic fault tolerance mechanism presented.

The Editor system described in [4] offers an extraction language that provides search, copy and replace operators for rewrite and restructuring purposes. InfoExtractor [15] utilizes concept definition frames, based on a mixture of simple regular expressions and weighted keywords to identify and interpret relevant source parts. Other approaches [2,9] define wrapper model specific extraction languages. The first approach is based on simple pattern matching and utilizes heuristic rules to determine the hierarchical structure of documents whereas the second one uses yacc to generate source specific extractors. NoDoSe[1] offers an interactive environment for semi-automatic extraction of data based on outlining of interesting regions in documents, semantic descriptions and mining tools which rely on the existence of simple prefix/postfix patterns to identify relevant data.

[11,12] both use non-deterministic automatons (NDA) where state transitions are determined by a pattern match. Though NDAs and Jedi's internal modeling of grammars by non deterministic stack automata (NDSA) is similar, NDSAs are more powerful as they can model context free grammars whereas NDAs can only model regular grammars.

Most approaches do not support fault tolerant interpretation of extraction specifications at all. [9, 11, 12] support a very limited kind of fault tolerance for unanticipated cases by providing explicitly defined alternative rules or transitions in case a rule does not match. Jedi's fallback mechanism is much more flexible and allows for generic, fault tolerant interpretation of grammar rules.

None of the above grammar based approaches can deal with ambiguous grammar specifications. As far as we know, no other parsing strategy uses a breadth first search based algorithm combined with lexicographic ranking and pruning to interpret ambiguous grammars.

7. Conclusion and Further Research

In order to overcome the brittleness of grammar based parsing approaches for extracting and filtering data from irregularly structured sources, new parsing technology is needed. This technology must allow for fault tolerant interpretation of ambiguous grammars.

As ambiguous grammars allow for multiple possible interpretations of a source, a criterion is needed to select one 'good' interpretation. We propose a lexicographic ranking on possible interpretations which uses the specificity of terminal rules as lexicographic comparison criterion. We show that the highest ranked solution is near to user expectations and thus can be selected as final solution.

Fault tolerance is achieved by adding implicitly fallback productions to rules which consume a minimal number of characters until an explicitly given production in the direct context of the failing rule matches again.

We present a breadth first search based parsing strategy to explore the solution space and use the specificity ranking to prune partial solutions which can never make the best solution in the end. We show further that this algorithm requires almost linear runtime whereas the runtime of a depth first search based strategy can easily explode.

The strategy has been implemented as part of the Jedi tool. It offers the extraction language needed to specify context free grammars for irregularly structured sources which can be extended by semantic predicates to disambiguate rules further. Grammar attribution is used to extract relevant source portions. A fully fledged scripting language and built-in data modeling means can be used to create flexible wrappers which rewrite sources directly or instantiate rich conceptual models for further querying and processing.

Further research has to address the most difficult and time consuming task related to wrapper generation for textual sources: the creation and maintenance of extraction specifications. This task has to be assisted and simplified by appropriate tools and metaphors based on supervised learning by example approaches, e.g. like MarkItUp![6], or unsupervised statistical analysis methods which can detect source structures automatically.

The fault tolerant extraction of irregularly structured sources leads to irregularly structured and heterogeneously typed goal representations. Thus, research has also to investigate in fault tolerant processing and querying of these representations which allows for meaningful treatment of type mismatches or structural irregularities.

8. References

[1] Brad Adelberg: NoDoSE: A Tool for Semi-Automatically Extracting Semi-Structured Data from Text Documents. SIGMOD Conference 1998: 0-

[2] Serge Abiteboul, Sophie Cluet, Vassilis Christophides, Tova Milo, Guido Moerkotte, Jérôme Siméon: Querying Documents in Object Databases. Int. J. on Digital Libraries 1(1): 5-19 (1997)

[3] Naveen Ashish, Craig A. Knoblock: Wrapper Generation for Semi-structured Internet Sources. SIGMOD Record 26(4): 8-15 (1997)

[4] Paolo Atzeni, Giansalvatore Mecca: Cut & Paste. PODS 1997: 144-153

[5] Bellman, R. E.: Dynamic Programming. Princeton University Press, Princeton. (1957)

[6] Peter Fankhauser, Yi Xu: MarkItUp! An incremental approach to document structure recognition. Electronic Publishing, vol 6(4), 447-456, Dec. 1993. Summary.

[7] Greve, J., Thysen, I., Jrgensen, E., and Andersen, S. K. (1994). The dynamic lexicographic approach and its application to mating of sows. Working paper.

[8] Bertin Klein, Peter Fankhauser: Error tolerant document structure analysis: International Journal of Digital Libraries 1997: 344-357

[9] J. Hammer, H. Garcia-Molina, J.Cho, R.Aranha, A. Crespo: Extracting Semistructured Information from the Web: Proceedings of the Workshop on Management of SemiStructured Data in conjunction with the 1997 ACM

[10] Joachim Hammer, Hector Garcia-Molina, Svetlozar Nestorov, Ramana Yerneni, Markus M. Breunig, Vasilis Vassalos: Template-Based Wrappers in the TSIMMIS System. SIGMOD Conference 1997: 532-535

[11] C.-N. Hsu: Initial Results on Wrapping Semistructured Web pages with Finite-State Transducers and Contextual Rules. AAAI'98 Workshop 'AI and Information Integration', Madison, July 1998. Summary.

[12] I. Muslea, S. Minton, C. Knoblock: Learning Wrappers for Semi-structured, Web-based Information, AAAI'98 Workshop 'AI and Information Integration', Madison, July 1998. Summary

[13] Hector Garcia-Molina, Yannis Papakonstantinou, Dallan Quass, Anand Rajaraman, Yehoshua Sagiv, Jeffrey D. Ullman, Vasilis Vassalos, Jennifer Widom: The TSIMMIS Approach to Mediation: Data Models and Languages. JIIS 8(2): 117-132 (1997)

[14] Mary Tork Roth, Peter M. Schwarz: Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. VLDB 1997: 266-275

[15] Dan Smith, Mauricio Lopez: Information Extraction for semi-structured documents: Proceedings of the Workshop on Management of SemiStructured Data in conjunction with the 1997 ACM

[16] Document Object Model Specification, Version 1.0, W3C Working Draft 16 April, 1998, http://www.w3.org/TR/WD-DOM/