Handling Irregularities in ROADRUNNER

Valter Crescenzi
Università Roma Tre – Italy
crescenz@dia.uniroma3.it

Giansalvatore Mecca
Università della Basilicata – Italy
mecca@unibas.it

Paolo Merialdo
Università Roma Tre – Italy
merialdo@dia.uniroma3.it

Abstract

We report on some recent advancements on the development of the ROADRUNNER system, which is able to automatically infer a wrapper for HTML pages. One of the major drawbacks of the ROADRUNNER approach was its limited ability in handling irregularities in the source pages. To overcome this issue, we have developed a technique to deal with chunks of unstructured HTML code. Several experiments have been conducted to evaluate the effectiveness of the approach, producing encouraging results.

Introduction

Motivated by the observation that many web sites contain now large collection of pages that share the same structure, several approaches have been recently proposed for automatically inferring a wrapper for structurally similar pages (Chang & Shao-Chen 2001; Crescenzi, Mecca, & Merialdo 2001; Arasu & Garcia-Molina 2003; Wang & Lochovsky 2002).

In (Crescenzi & Mecca 2004) we have developed a theoretical framework of our approach, explicating its connections with the more traditional field of grammar inference. Based on that theoretical study we have framed the problem of inferring wrappers into a search problem in a space of states. This setting has been used to implement the ROADRUNNER system, originally described in (Crescenzi, Mecca, & Merialdo 2001), whose source code has been recently released under GPL.[1]

This paper presents some recent advances that allow the original approach to handle pages containing local irregularities. In particular, we have introduced techniques, which have been included in the prototype, to improve the expressiveness of the inferred wrapper without compromising performances.

The paper is organized as follows. First, we revisit the core technique underlying the ROADRUNNER approach. Then, we show how the original framework has been extended in order to handle irregularities, and we report on some experimental results that show how the new features have improved the effectiveness of the system. Finally, we briefly mention some evolutions we are working on.

[1] `http://www.dia.uniroma3.it/db/roadRunner`

The Matching Technique

In ROADRUNNER wrappers are represented by *union–free regular expressions* (UFRE). To generate a wrapper we use a small set of sample pages to progressively infer a common grammar.

Let us briefly introduce our notation for describing UFREs. Given a special symbol `#PCDATA`, and an alphabet of symbols T not containing `#PCDATA`, a *union-free regular expression* over T is a string over alphabet $T \cup \{\texttt{\#PCDATA}, \cdot, +, ?, (,)\}$ defined as follows. First, the empty string, ϵ and all elements of $T \cup \{\texttt{\#PCDATA}\}$ are union-free regular expressions. If a and b are UFRE, then $a \cdot b$, $(a)^+$, and $(a)?$ are UFRE. The semantics of these expressions is defined as usual, $+$ being an iterator and $(a)?$ being a shortcut for $(a|\epsilon)$ (denotes optional patterns). `#PCDATA` is a special symbol we use to denote attributes to extract.

Figure 1 shows a regular expression, which can be used as a wrapper for a hypothetical web page.

The core of the ROADRUNNER approach for inferring wrappers expressed as UFRE is the matching technique (Crescenzi, Mecca, & Merialdo 2001).

MATCH treats HTML sources as lists of tokens, each token being either an HTML tag or a string, and works on two objects at a time: (i) a *sample*, i.e., a list of tokens corresponding to one of the sample pages, and (ii) a *wrapper*, i.e., a regular expression. The idea is to parse the sample with the wrapper: whenever the wrapper fails MATCH tries to generalize it. To start, one of the sample pages is taken as the initial version of the wrapper.

Figure 2 shows a simple example in which two HTML sources have been transformed into lists of 30 and 23 tokens, respectively.

The algorithm consists in *parsing* the sample by using the wrapper. Parsing can fail for mismatches between the wrapper and the sample: a *mismatch* happens when some token in the sample does not comply with the grammar specified by the wrapper. Whenever one mismatch is found, the algorithm tries to solve it by generalizing the wrapper. This is done by applying suitable *generalization operators*. The algorithm succeeds if a common wrapper can be generated by solving all mismatches encountered during the parsing.

<html>...(...Model:#PCDATA<hr>... Price:
#PCDATA ...)+...(#PCDATA)?</html>

Figure 1: A UFRE as a wrapper

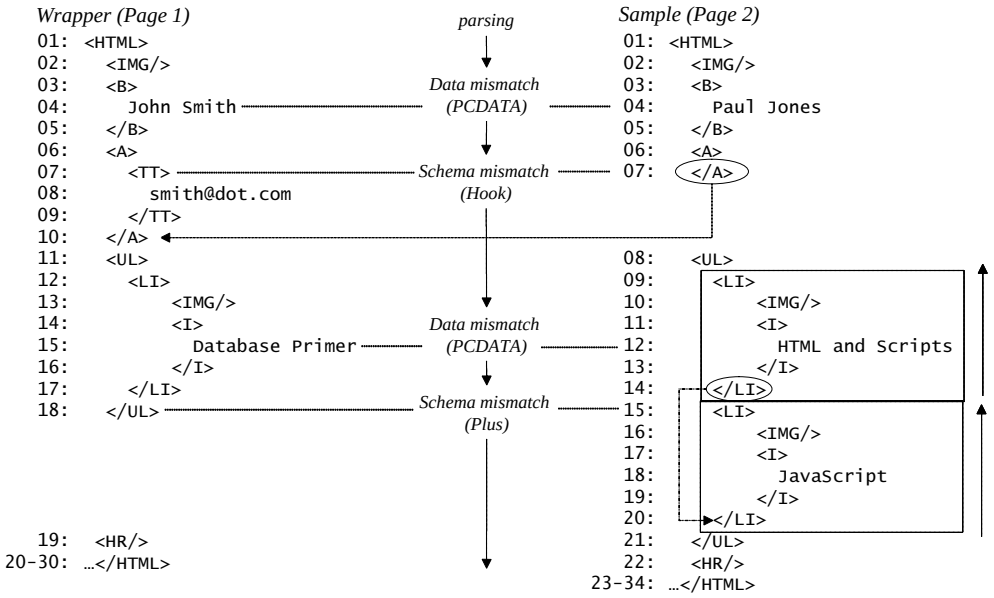

Figure 2: One Simple Matching

After a first wrapper is produced by matching two samples, it can be refined by iteratively applying MATCH against the other samples.

The Generalization Operators

There are essentially two kinds of mismatches that can be generated during the matching. The simplest case is that of *data mismatches*, i.e., mismatches that happen when two different strings occur in corresponding positions of the wrapper and of the sample. Their presence may be due only to different values of the same attribute. This case is solved by applying the operator *addPCDATA*.

More complex mismatches that involve either two different tags, or one tag and one string on the wrapper and on the sample are called *schema mismatches*. These mismatches, can be due to the presence of iterators (i.e., lists) and optional patterns. Whenever it is possible, we generalize zero or one repetition by introducing an optional, one or more repetitions by introducing a list. Schema mismatches are solved by applying the operators *addHook* and *addPlus.* These ideas are clarified in the following with the help of an example, shown in Figure 2.

Applying Operator *addPCDATA***: Discovering Attributes** Figure 2 shows several examples of data mismatches during the first steps of the parsing. Consider, for example, strings `'John Smith'` and `'Paul Jones'` at token 4. To solve this data mismatch, we apply operator *addPCDATA*, i.e., we generalize the wrapper by replacing string `'John Smith'` by `#PCDATA`. The same happens a few steps after for `'Database Primer'` and `'HTML and Scripts'`.

Applying Operator *addHook***: Discovering Optionals** Schema mismatches are used to discover both lists and optionals. This means that whenever one of these mismatches is found, the algorithm needs to choose which operator to apply. Let us for now ignore the details of this choice, and concentrate first on the discovery of optionals, i.e., the application of operator *addHook*. Lists will be discussed in the following.

Consider again Figure 2. The first schema mismatch happens at token 7 due to the presence of the email in the wrapper and not in the sample, i.e., the mismatch is due to an optional which has been instantiated in different ways. To apply operator *addHook* and generalize the wrapper, we need to carry out the following steps:

1. Optional Pattern Location by Cross–Search With respect to the running example, given the mismatching tags at token 7 – `<TT>` and `</A>`, a simple cross-search of the mismatching tags leads to the conclusion that the optional pattern is located on the wrapper.

2. Wrapper Generalization the optional is introduced in the wrapper. In this case, the wrapper is generalized by introducing one pattern of the form `(<TT>smith@dot.com</TT>)?`, and the parsing is resumed by comparing tokens `<UL>` (11 and 8 respectively).

Applying Operator *addPlus***: Discovering Iterators** Consider again Figure 2; it can be seen that the two HTML sources contain, for each author, one list of book titles. During the parsing, a schema mismatch between tokens 18 and 15 is encountered; it is easy to see that the mismatch comes from different cardinalities in the book lists (one book on the wrapper, two books on the sample). To solve the mismatch, we need to identify these repeated patterns, called *squares*, by applying the operator *addPlus*. In this case three main steps are performed:

1. Square Location by Delimiter Search A schema mismatch due to presence of a repeated pattern gives key hints about the involved square: we can identify the last token of the square by looking immediately before the mismatch position, i.e. `</LI>`, and the first token of the square by looking at mismatching token, i.e. `</UL>` on the wrapper and `<LI>` on the sample. However, this originates two possibilities that we resolve by searching first the wrapper and then the sample for occurrences of the last token of square `</LI>`; in our example, this lead to conclude that the sample contains one candidate square occurrence at tokens 15 to 20.

2. Candidate Square Matching To check whether this candidate occurrence really identifies a square, we try to match the candidate square occurrence (tokens 15–20) against some upward portion of the sample. This is done backwards, i.e., it starts by matching tokens 20 and 14, then moves to 19 and 13 and so on. The search succeeds if we manage to find a match for the whole square, as it happens in Figure 2.

3. Wrapper Generalization It is now possible to generalize the wrapper; if we denote the newly found square by s, we do that by searching the wrapper for contiguous repeated occurrences of s around the mismatch point, and by replacing them by $(s)^+$.

Recursion

In general, the number of mismatches to solve may be high, mainly because the mismatch solving algorithm is inherently recursive: when trying to solve one mismatch by finding an iterator, during the candidate square matching step more mismatches can be generated and have to be solved.

To see this, consider Figure 3, which shows the process of matching two pages inspired from our previous example with the list of editions nested inside the list of books. The wrapper (page 1) is matched against the sample (page 2). After solving a couple of data mismatches, the parsing stops at token 25, where a schema mismatch is found. It can be solved by looking for a possible iterator, following the usual three steps: (i) the candidate square occurrence on the wrapper is located (tokens 25–42) by looking for an occurrence of the possible end delimiter (`</LI>` at token 24); then (ii) the candidate is evaluated by matching it against the upward portion of the wrapper (tokens 25–42 against the portion preceding token 25); and finally, (iii) the wrapper is generalized. Let us concentrate on the second step: remember that the candidate is evaluated by matching it backwards, i.e., starting from comparing the two occurrences of the end delimiter (tokens 42 and 24), then move to tokens 41 and 23 and so on.

This comparison has been emphasized in Figure 3 by duplicating the wrapper portions that have to be matched. Since they are matched backwards, tokens are listed in reverse order. Differently from the previous example – in which the square had been matched by a simple alignment – it can be seen that, in this case, new mismatches are generated when trying to match the two fragments. These mismatches are called *internal mismatches*. The first internal mismatch in our example involves tokens 35 and 17: it depends on the nested structure of the page, and will lead to the discovery of the list of editions inside the list of books.

These internal mismatches have to be processed exactly in the same way as the external ones. This means that the matching algorithm needs to be recursive, since, when trying to solve some external mismatch, new internal mismatches may be raised, and each of these requires to start a new matching procedure, based on the same ideas discussed above. The only difference is that these recursive matchings do not work by comparing one wrapper and one sample, but rather two different portions of the same object, i.e. either wrapper or sample.[2]

MATCH as a state space search problem

The matching of a wrapper and a sample can be considered as a search problem in a particular state space. States in this space correspond to different versions of the wrapper, i.e. regular expressions. The algorithm moves from one state to another by applying instantions of operators *addPCDATA*, *addPlus*, *addHook*. A final state is reached whenever the current version of the wrapper can be used to correctly parse the given sample.

It can be seen that this recursive nature of the problem makes the algorithm quite involved. During the search in the state space, in order to be able to apply *addPlus* operators it is necessary to trigger a new search problem, which corresponds to matching candidate squares. In this respect, the state space of this new problem may be considered at a different level: its initial state coincides with the candidate square of the operator while the final state, if any, is the square which will be used to generalize the wrapper in the upper level. The search in this new space may in turn trigger other instances of the same search problem. These ideas are summarized in Figure 4, which shows how the search is really performed by working on several state spaces, at different levels.

As a search space, the algorithm sketched in this section might be subjected to backtracking. This is due to the fact that, in general, to solve a mismatch the algorithm needs to choose among several alternatives, which are not guaranteed to lead to a correct solution. When, going ahead in the matching, these choices prove to be wrong, it is necessary to backtrack and resume the parsing from the next alternative until the wrapper successfully parses the sample.

[2]Internal mismatches may lead to matchings between portions of the wrapper; since the wrapper is in general one regular expression, this would require matching two regular expressions, instead of one expression and one sample. The solution of this problem, which is out of the scope of this paper, can be found in (Crescenzi & Mecca 2004).

Figure 3: A More Complex Matching

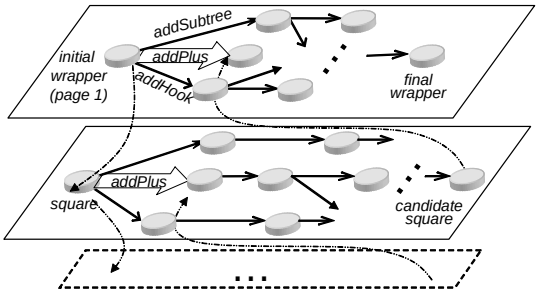

Figure 4: Matching as a search problem in a state space

In the following we exploit the framing of the matching technique into a search problem to augment the expressiveness of wrappers.

Handling irregularities

A key point of our approach is that a wrapper can be inferred only for pages that comply to a union-free regular grammar. Unfortunately, several web pages do not fall in this class. To give an example, consider the customer reviews that are usually published with the products sold by e-commerce web sites. Reviews are pieces of free text including HTML tags that may occur in different ways in every reviews just for presentational purposes. These tags would lead our approach to a failure, since none of our operators could suitably generalize the wrapper to continue the parsing. It is worth noting that the approach fails, even if the structures of the input pages strongly overlap in every section but the reviews. Another typical example that arise the same issue is related to the presence of banners, advertising, and personalized headers: we have pages whose structures largely overlap, but differ in few details.

With respect to our framework, we may say that it is not always possible to solve the differences of the pages with optional and iterative patterns. To address this issue we have extended ROADRUNNER with a further operator, called *addSubtree*. By introducing a *subtree* in the generalization of a wrapper, we somehow desist to model in detail the differences of specific regions, but we let the matching continue on the remainder.

The idea is to avoid a mismatch by skipping a region containing it. In order to identify such a region, it is convenient to rely on the DOM tree representation of the page: we choose the smallest DOM subtree containing the mismatch.

Applying Operator *addSubtree*: Handling Irregularities

Consider Figure 5: it shows the bottom parts (lines 20-29 and 23-33, respectively) for the two pages of our first example (Figure 2). It is likely that these pieces of text, which represent the authors' biographies, come from a database in which they are stored directly as fragments of HTML code. Note that tags are inserted in these fragments uniquely for presentation purposes, and they do not follow any predefined order. Then, the schema mismatch involving tokens 22 and

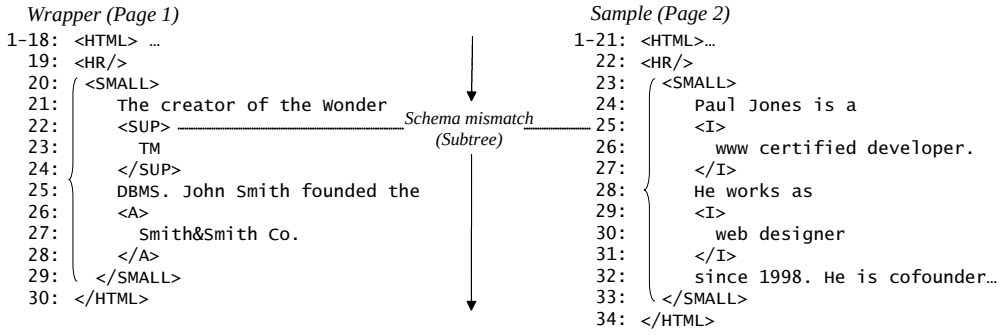

Figure 5: Solving mismatches by applying subtree

25 cannot be solved by the operators explained so far.

In this case the mismatch can be solved by isolating the whole fragment, as follows:

1. Subtree Location It is chosen the shortest well-formed region on the wrapper containing the mismatch, in this case tokens 20 to 29 which cover the subtree under the `<SMALL>`.

2. Wrapper Generalization Once the subtree has been identified, we may generalize the wrapper accordingly and then resume the parsing. In this case, the wrapper is generalized by introducing one pattern of the form `<SMALL>`$\triangle$`</SMALL>`, and the parsing is resumed by comparing tokens `</HTML>` (30 and 33 respectively).

$\triangle$ is a special symbol we use to represent a subtree in a regular expression, it is always placed between an open tag and the corresponding closing tag and it matches with a list of tokens in a region likewise placed within the same open and close tag.[3] Figure 6 shows the final wrapper produced.

$$
\begin{array}{l}
< HTML >< IMG/ >< B > \#PCDATA < /B > \\
< A > (< TT > smith@dot.com < /TT >)? < /A > \\
< UL > \\
(\\
< LI >< IMG/ >< I > \#PCDATA < /I >< /LI > \\
)+ \\
< /UL > \\
< HR/ >< SMALL > \triangle < /SMALL > \\
< /HTML >
\end{array}
$$

Figure 6: Final Wrapper

Observe that whenever a mismatch occurs, a subtree can always solve it. Therefore, in the worst case, we could insert a subtree rooted at `HTML` – i.e., treat the input as 100% noise. To avoid such undesired solutions, we only add $\triangle$ as a last resort. As discussed in the following, ROADRUNNER applies the operators according to a priority system, and $\triangle$ is applied only if neither *addPlus* nor *addHook* can solve the mismatch.

[3]The semantics of $\triangle$ refers to the concept of open and close tag, and it makes sense only within expressions over alphabets which model this concept.

Expressiveness and Performances

The presence of backtracking makes the algorithm exponential.[4] However, from a practical point of view, even if exponential behaviors are possible, they rarely occur if a solution exists. Intuitively, this may be motivated by the fact that the matching is a process quite delicate, and whenever a wrong operator is chosen, it usually leads to expressions which quickly generate unsolvable mismatches. In other words, most of *dead-paths* in the search space are quickly recognized without too much backtracking.

Another relevant issue is that the augmented expressive power of inferrable expressions entails a larger number of possible solutions. We solve this issue by ordering operators according to a heuristic, which gives higher priority to operators that most likely lead towards more precise solutions. The management of priorities is rather complex and configurable; in general, we may say that *addPlus* are always preferred to *addHook*; *addHook* are always preferred to *addSubtree*.

Experiments

We have evaluated the impact of subtrees with several experiments, in order to evaluate their effectiveness. In this paper, to summarize the significance of subtrees, we report on the result obtained when running the prototype against the Wien test-bed (Kushmerick 2000).

We have used the same system configuration for all sites, selecting ten pages from every source. If after twenty seconds the prototype was still running, we stopped it and consider the test failed (we assume that the system is not able to produce a wrapper).

For each dataset, we have computed the number of values a wrapper should extract (column `#values`).[5]Then, we have generated the wrappers, and we have used them in order to extract data from the source pages. Finally, we have

[4]In (Crescenzi & Mecca 2004) it has been identified a class of languages, called *prefix mark-up languages*, for which the matching can be performed in polynomial time with respect to the length of input samples. Unfortunately, prefix mark-up languages are not sufficiently expressive for several real-life web pages.

[5]When available, we have used the set of labels provided with the dataset.

counted the number of values the generated wrapper extracts. We distinguish *extracted* and *partially extracted* values. The former represent values that have been exactly extracted by the system; the latter are values that, though extracted, have been grouped together with other values.[6]

		without Subtrees		with Subtrees	
Sample	# values	Extracted	Partially Extracted	Extracted	Partially Extracted
1	1612	no wrapper		1612	
2	1010*	1010		1010	
3	1044	522	522	522	522
4	400	no wrapper			400
5	144	144		144	
6	100*	no wrapper		100	
7	1688	no wrapper			1688
8	654	654		654	
9	572	no wrapper		80	492
10	400	295	42	295	42
11	400*	no wrapper			400
12	888	862		862	
13	400	290	108	290	108
14	2910	2899	10	2899	
15	708	708		708	
16	100*	100		100	
17	1891	1891		1891	
18	2436	no wrapper		389	2047
19	1000	794	200	794	200
20	1962	1308		1308	
21		no wrapper		no wrapper	
22	3000	3000		3000	
23	1635	no wrapper		1635	
24	1550*	no wrapper		1550	
25	654	654		654	
26	386*	10	376	10	376
27	60	30	30	30	30
28		no wrapper		no wrapper	
29	425*	no wrapper			425
30	240	30	210	30	210

** not labeled in the wien dataset, estimated by the authors of the present paper*

Figure 7: Experimental results: the effects of subtrees on the Wien dataset

Figure 7 reports the results of our experiment. Disabling the *addSubtree* operator, ROADRUNNER generated 18 wrappers. Enabling the *addSubtree* operator, the situation improved, as the system correctly generated wrappers for 28 datasets, and only in 2 cases (sources 21, 28) it failed.

However, we observe that the solution generated for sources 4, 7, 9, 11, 26, 29 cannot be consider satisfactory, as the system generated wrappers that extract most of the expected values into a unique attribute (a subtree). On the contrary, the introduction of subtrees allowed the system to correctly generate "good" wrappers for sources 1, 6, 23, 24.

We discuss the failures starting from a general observation: the HTML code of pages from this test-bed are quite poor compared to modern web sites (the dataset is rather old). Poor HTML code may confuse ROADRUNNER, because the same tags are extensively used to mark completely different attributes, thus originating ambiguities.

As for the single sources, the ambiguity due to the poor HTML motivates the low quality of wrappers obtained on sources 7, 9 and 11; source 26 presents main contents as untagged text (text within `<PRE>` tags). Source 21, which reports acts of Shakespeare's operas, is poorly structured.

Finally, we report that by manually tuning the system configuration, we have improved the overall results as the system generated wrappers also for sources 4, 28, 29.

Future Work

We believe there are at least two different reasons why subtrees deserve more work. First, even if we have introduced them in order to improve the expressiveness of inferrable wrappers, they can be used to enhance the robustness of the wrappers (Davulcu *et al.* 2000), as well. If the structure of the HTML code of samples changes, even slightly, it is likely that a wrapper for that source stops working. Since ROADRUNNER models every single token of input samples, the inferred wrappers are sensitive to changes in any portions of samples. For example, they may fail even if changes involve the formatting of advertisements, or irrelevant attributes. In order to make wrappers more resilient to changes, during a post-generation phase we can relax its regular expression by replacing regions which do not involve attributes of interest with subtrees that model as loosely as possible that regions. If we are interested in a subset of the attributes associated with the inferred wrapper, this technique can be even more effective, since larger regions can be pushed under a subtree.

Second, we are studying how to enhance further expressiveness and precision of our formalism in order to introduce a restricted form of disjunction. This possibility is based on the observation that whenever a disjunction occurs, the system introduces a subtree.

[6] It is important to say that often this happens because of the alphabet of ROADRUNNER includes only HTML tags, while in the Wien dataset several values are separated by symbols like punctuation marks.

References

Arasu, A., and Garcia-Molina, H. 2003. Extracting structured data from web pages. In *ACM SIGMOD 2003*.

Chang, C.-H., and Shao-Chen, L. 2001. Iepad: Information extraction based on pattern discovery. In *WWW2001*.

Crescenzi, V., and Mecca, G. 2004. Automatic information extraction from large web sites. *Journal of the ACM*. Accepted for publication.

Crescenzi, V.; Mecca, G.; and Merialdo, P. 2001. ROADRUNNER: Towards automatic data extraction from large Web sites. In *VLDB2001*.

Davulcu, H.; Yang, G.; Kifer, M.; and Ramakrishnan, I. 2000. Computational aspects of resilient data extraction from semistructured sources. In *PODS2000*.

Kushmerick, N. 2000. Wrapper induction: Efficiency and expressiveness. *Artificial Intelligence* 118:15–68.

Wang, J., and Lochovsky, F. 2002. Data-rich section extraction from html pages. In *WISE 2002*.