From Web Pages to Web Services with `wal`

Markus L. Noga[1] and Max Völkel[2]

[1] IPD Goos, Universität Karlsruhe
76128 Karlsruhe, Germany
markus@noga.de
[2] ZeMM, Universität Karlsruhe
76128 Karlsruhe, Germany
max@xam.de

Abstract. Today's web sites closely intertwine services with visual markup. Web page operators are unlikely to change this, as they depend on advertising revenues. `wal` enables average users to build web service wrappers for existing pages. By spawning many basic web services, this prepares the ground for the next wave of compound web services.

1 Introduction

The current generation web is designed for direct human usage. This technology has already realized huge savings across a broad swathe of industries, but deep penetration into process chains has so far remained elusive. Visual web pages do not lend themselves readily to automation. However, building layers of new services on top of existing ones is the most likely candidate for further savings from automation [1].

The semantic web initiative [2] aims to support automation by embedding not only visual, but content markup in web pages. This requires additional investment by web page operators. However, until a broad shift from advertising-based revenue models to subscription-based ones occurs, automatable web pages will actually hurt the economic self-interests of many web page operators [3]. They are therefore unlikely to make any such investment. In contrast, users should be willing to invest, as automation directly reduces their operating costs.

In 1999, some 80% of all web pages were generated from databases using multi-tier architectures [4]. This trend continues, indicated by the sustained growth of the LAMP platform [5] and its constituents Apache and PHP [6].

Generated pages fall into three broad categories: *text pages* contain text which is not formally structured, e.g., the top stories on a news site. *data pages* contain one or more database records, e.g., telephone numbers, stock quotes or weather forecasts. *logic pages* provide access to specific pieces of business logic, e.g., web mail services, online shopping or electronic banking.

Because users do not have direct access to the business logic or database layers of the page operator, automation by users amounts to reverse engineering the presentation layer to provide automation-friendly wrappers.

Writing these wrappers by hand requires trained personnel; it is costly and error-prone [7]. To provide an appropriate level of abstraction, many domain-specific languages for web data extraction have been proposed. However, every language has a learning curve, and languages cannot bridge the gap between an HTML file and its visual representation completely. In `wal`, we introduce a tool that operates on the visual representation only.

This paper is organized as follows. In the next section, 2, we introduce basic ideas about web pages, browsers and wrappers. Section 3 summarizes the state of the art in wrappers. In section 4, we introduce the design behind `wal` and its authoring and production subsystems. The following section, 5 discusses some implementation decisions. We evaluate the system in section 6. Section 7 summarizes the paper and outlines directions for future work.

2 Basics

It is a common misconception that texts have syntax and semantics. Consider the state of Egyptology before 1822. Hieroglyphs were but meaningless scribbles before Champollion successfully exploited the external semantic context given by the Rosetta stone. Only because of this context could he assemble sequences of hieroglyphs to *abstract syntax trees* (*ASTs*) of an intelligible language. A text without context, like the Phaistos disc, is devoid of syntax and semantics.

The web pages rendered in a browser are texts interspersed with markup. Where does their context come from? It is another common misconception that these texts conform to a standard called *Hypertext Markup Language* (*HTML*) [8] created by the Worldwide Web Consortium (W3C). In a random sample of 5.000 pages we tested, only 0.24% did so. For practical purposes, HTML documents are considered 'valid' if they can be parsed by at least one of the most widely used web browsers: Internet Explorer, Netscape/Mozilla and Opera [9]. We call this variant PH, or *practical HTML*.

Apart from syntax, web browsers also define a form of semantics for PH. After parsing a character sequence and building a corresponding AST, browsers render the AST as a two-dimensional bitmap graphic[3]. Thus, browsers implicitly define a map from ASTs to a formal model of bitmaps. Displayed on a computer screen, these bitmaps are in turn interpreted by a human being (see fig. 1).

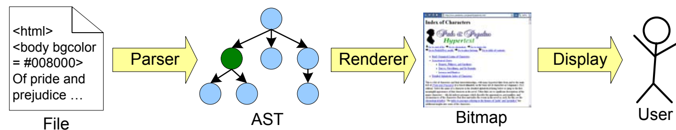

Fig. 1. Interpreting HTML files

Parsing PH and building an AST is hard, but unparsing this AST to an unambiguous format is easy. There are several common notations for trees, including the *Standardized General Markup Language* (*SGML*) [10], and its modern

[3] For simplicity, we disregard scripting, embedded media and the `<blink>` tag.

subset *eXtensible Markup Language* (*XML*) [11]. We call the XML documents obtained by encoding PH ASTs as XML documents XH.

By definition, every XH document is an XML document. In contrast, the XML subset XHTML [12] is a subset of XH, because ASTs for PH do not have to conform to the grammatical restrictions of XH. Similarly, PH is a superset of SGML due to the extensive error correction in browsers.

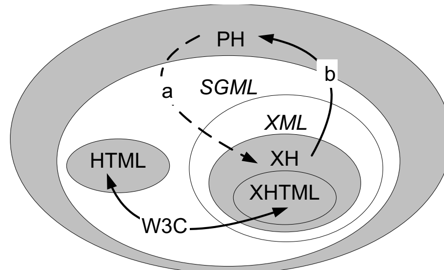

Fig. 2. Language inclusions

Figure 2 shows the relations between PH, XH and the standards. The a arrow depicts the hard act of parsing PH, whereas the b arrow depicts the easy act of writing XH as PH. The W3C arrow shows the correspondence between HTML and XHTML.

The real-world meaning of an HTML document, its pragmatics, are established by a human human user regarding the bitmap. Unfortunately, the bitmap level is rather less tractable to machines than the AST level, as it decomposes phrases, words and characters into pixels. Thus, content extraction should happen on the AST level.

What is the impact on the AST if some 80% of all web pages are generated from templates using two- or three-tier architectures [4]? Pages generated from the same template form a *page family* — family members share *structural* AST nodes, but differ in *content* nodes (see fig. 3). The remaining 20% of web pages could each be perceived as members of a singleton family, but there is little point in automating extractions for a singleton.

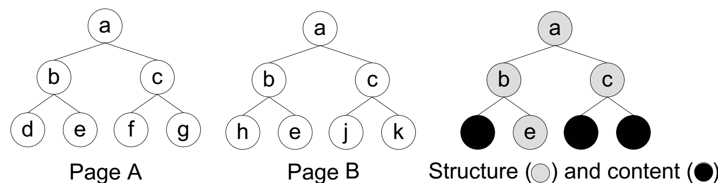

Fig. 3. Structual and content nodes in a page family

Now, we can formally define wrappers for our context. An *atomic* wrapper encodes the AST of a given PH document as XH, then extracts all or part of its content nodes and provides an external interface to them. *Compound* wrappers aggregate several atomic wrappers to provide services that require navigation across or integration of multiple pages. This paper focuses on atomic wrappers.

3 State of the Art

In this section, we examine existing production and authoring tools for atomic wrappers and discuss robust HTML parsers as vital constituents of a new tool.

3.1 Wrapper tools

To republish web pages as web services with atomic wrappers, we are interested in tools that operate on the AST level and deliver XML output. Because pragmatics are established during visualization, their authoring environment should work on that level. Tools must be freely available at least for noncommercial users to encourage widespread use.

Two good surveys of web data extraction tools are [3, 13]. The regularly updated companion website [14] to [3] lists 35 tools at the time of this writing. Fig. 4 shows a timeline of the seven freely available, XML-capable tools alongside `wal`. The black bar denotes the first and last known releases, a gray continuation indicates ongoing maintenance and development. As a lack of maintenance is a strong indicator for disuse, we disregard the four legacy competitors [15–18] and focus on the remaining three.

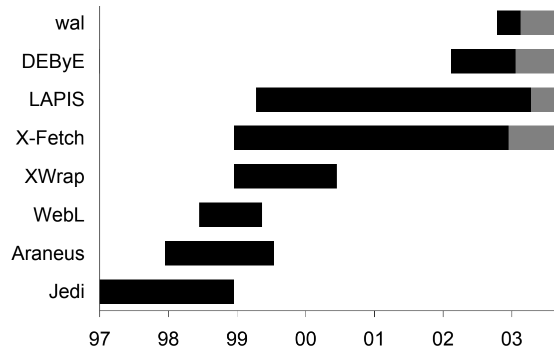

Fig. 4. Timeline of XML-capable wrapper tools

Data Extraction by Example (*DEByE*) [19] proceeds from sample data given by the user to generate regular expressions that operate on the file level. Thus, this system does not work on the AST proper.

LAPIS [20] is a text editing tool adapted to generate wrappers. Extractions are specified in a custom *text constraints* language, which again works on the file level, not the AST proper. There is no support for authoring during visualization.

X-Fetch Wrapper [21] is a general text-to-XML converter. Its *data extraction language* operates on files using regular expressions, not on the AST proper.

Thus, none of the tools available for evaluation satisfy all of our criteria.

3.2 Error-correcting HTML parsers

By definition, an atomic wrapper must obtain an AST prior to extracting content nodes from it. Several freely available libraries focus on the task of parsing practical HTML, or PH. We are interested in well-maintained libraries that can deal with real-world input and deliver XH output. Fig. 5 shows a timeline of such tools. We again disregard the two unmaintained tools [22, 23] and focus on the remaining four.

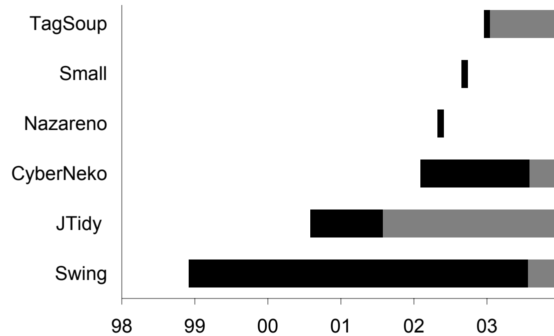

Fig. 5. Timeline of error-correcting HTML parsers

TagSoup [24] is an open source project. It generates SAX events corresponding to XH. TagSoup attempts to generate output for all input files, imitating Internet Explorer and Netscape behavior closely. Embedded scripts still cause some problems for TagSoup.

CyberNeko [25] is a part of the Apache Xerces 2 XML parser. It generates a stream of SAX events encoding XH. CyberNeko properly handles unknown elements, escaped and unescaped scripts as well as entities. Development and maintenance continue as an open source project.

JTidy [26] is the Java port of *HTMLTidy* [27], the first free PH parser. It generates XH and XHTML output in the form of DOM trees or files. This parser treats unknown elements as errors and breaks unescaped scripts. Development and maintenance continue as an open source project.

The *Swing Html Parser* [28] is included with the Java SDK since version 1.2. It generates XH output as a DOM tree. Although adequately maintained, this library implements the deprecated HTML 3.0 standard, excluding all HTML 4.0 tags used in PH.

Therefore, only CyberNeko, JTidy and TagSoup fullfill both of our criteria in being well-maintained and capable of handling real-world input.

4 Design

In this section, we first outline basic design decisions for the production system to execute wrappers, then for the authoring system to generate them. Afterwards, we describe the architecture in more detail.

By the definition of an atomic wrapper, the production system retrieves web pages and returns extraction results in XML. Components to retrieve web pages are readily available. They deliver PH files. Any of the three remaining candidates from Section 3.2 can convert from PH to XH. Because XH documents are XML documents, a host of standard tools apply to them. For example, we can formulate extractions on XH using *eXtensible Stylesheet Language Transformations* (*XSLT*) [29], which deliver the desired XML output. Thus, the production system is a classic case of design by reuse.

The task of the authoring environment therefore boils down to generating suitable XSLT transformations by user interaction on a visualization of the source document. As building a custom visualizer compatible with major browsers would be prohibitively expensive, visualization and interaction must occur inside a regular browser. For in-browser interaction, the authoring system must instrument the source document with custom user interface code. In the language of design patterns [30], the authoring system must act as a proxy to the source document. As a direct manifestation of this concept, the authoring system operates as a custom HTTP Proxy.

4.1 System Architecture

The production system consists of three components called surf, clean and extract. Fig. 6 shows the data flow between the components as a sequence diagram.

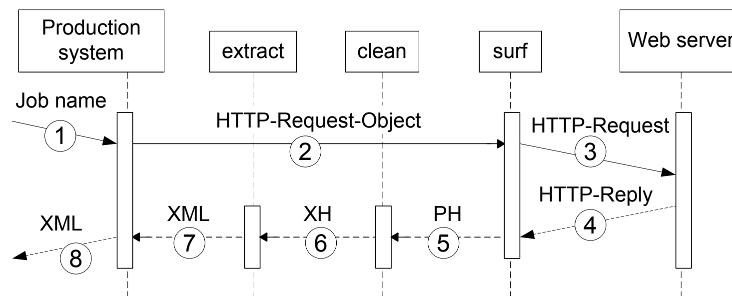

Fig. 6. Production system data flow

Requests for jobs arrive at the system boundary (1), where the web page and transformation are determined. Passing unchanged through the extraction and cleaning modules (2), the surf model requests the page from a remote server (3). Reversing the invocation chain, the server response (4) is stripped of its headers to reveal PH (5). This is converted XH (6) by the cleaner, then the extractor retrieves the desired fragments by interpreting an XSLT transformation (7). Crossing the outer system boundary again, the XML result is delivered to the user (8).

The authoring system builds on the production system. It contains three additional components called proxy, ui and learn and acts as an HTTP proxy towards the user's browser. Fig. 6 shows the data flow between the components for a typical authoring session.

The authoring system forwards the user's page requests to the surf module as the user surfs to the desired location (1). By sending a *record* command (2), the user initiates actual authoring.

His next HTTP request (3) is intercepted, the page is cleaned (4) and instrumented with a selection user interface. The browser displays the page (5), which still looks normal. But clicks on the document build the selection on the client side. The *select* command transmits the selected set to the server, which returns a visual confirmation of the selection (6).

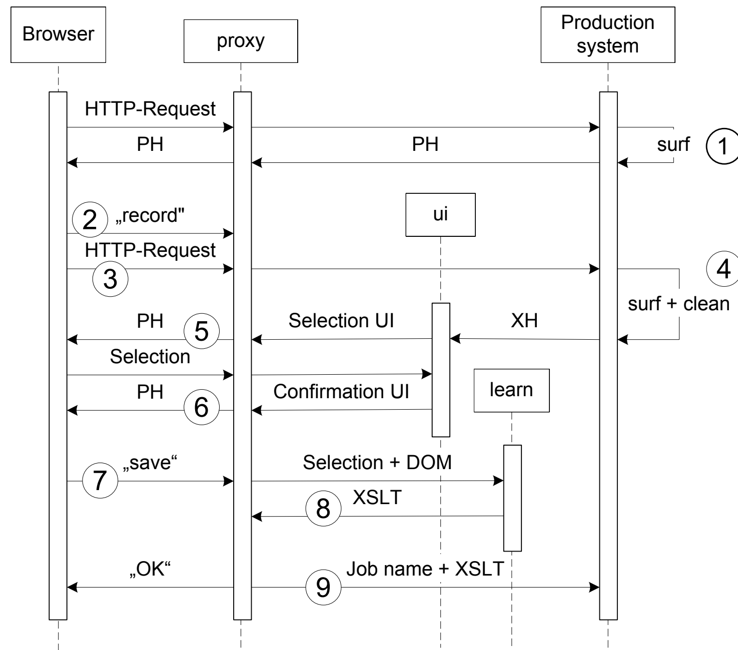

Fig. 7. Sequence diagram of the authoring system

Upon receiving the user's *save* command (7), the learning component generalizes the user selection to an XSLT extraction (8) that is stored under the desired job name (9) in the production system.

5 Implementation

The architecture left open several issues: How to instrument pages and add user commands without disrupting the visualization? How to learn transformations from user selections? And which off-the-shelf components should be used?

5.1 Instrumenting pages

To instrument pages for selection, first assign each element an an unique ID in a previously unused attribute. A script to create a hidden selection form is added, and all elements are supplied with mouse event handlers that add their IDs to the selection form respectively enlarge the selection. The *select* bookmarklet submits that form to the proxy server. To prevent interaction with existing links, all link elements are replaced with text spans that simulate the appearance of a link. See fig. 8 for an instrumented sample.

5.2 User commands

Which are the implementation alternatives for the commands the user has to send from within the browser? Command buttons in a separate window cannot

Inserted JavaScript
d2c_action realizes selections,
d2c_callServer sends them to proxy

```
<HTML d2c_uid="0">   (2)
  <HEAD d2c_uid="1">

(1)
    <SCRIPT language="JavaScript" type="text/javascript">}
      <!--// hide from old browsers}
      ... static JavaScript with the functions:
      function d2c_action( element ) { ... }
      function d2c_callServer() { ... }
      ...
      // end hiding-->
    </SCRIPT>

    <TITLE d2c_uid="3">
       d2c-UserInterface running... Old title</TITLE>
    ... Existing JavaScript-Elements...
    ... META-Elements
  </HEAD>         (3)
  <BODY d2c_uid="2" onclick="d2c_action(this);">

(5)
    <FORM name="d2c_selection"
        action="http://uipost.action.d2c" method="post">
      <INPUT type="hidden" name="d2c_selection" value="-">
    </FORM>

    <TABLE border="0" d2c_uid="9" onclick="d2c_action(this);">
      <TR d2c_uid="10" onclick="d2c_action(this);">
        <TD class="menu" d2c_uid="11" onclick="d2c_action(this);">
          <H3 d2c_uid="13" onclick="d2c_action(this);">
(4)         <SPAN d2c_uid="55" onclick="d2c_action(this);"
              onmouseover="d2c_showlink('relative Link');"
              onmouseout="d2c_showlink(' ');"
              style="color:#000099; ">Heading with Link</SPAN>
          </H3>
        </TD>
      </TR>
    </TABLE>

  </BODY>
</HTML>
```

Dynamically generated
HTML form to submit
the selection

Simulate link with SPAN Element and JavaScript
to make text selectable and disable
other browser actions.

Legende:
Inserted element numbers (d2c_uid) and JavaScript calls (d2c_action)
are shown in gray.

Fig. 8. An instrumented *XH* page

access the form in the selection window due to security constraints. As the page is being instrumented in any case, one could insert the extra buttons there. However, this disrupts the page layout and may break scripts in that page.

The solution comes in the shape of bookmarklets [31]. Browsers interpret scripts in bookmark URLs in the context of the current page. This allows us to add proxy commands directly to the browser toolbar (see fig. 9).

Fig. 9. Bookmarklets can execute command scripts in the current page

5.3 Learning transformations

`wal` currently does not use machine learning techniques, but translates selection identifiers directly into XPath expressions. This places the burden of correct generalization squarely on the user. Due to the modular construction of the system, more sophisticated approaches can be easily introduced in the future.

5.4 Off-the-shelf components

Using a custom benchmark for HTML parsers whose description would exceed the scope of this article [32], we determined TagSoup to approximate existing browsers the most closely. The *clean* component uses this parser.

To implement the proxy server, we chose Jetty 4.1.4. Other alternatives included the Tiny Java Web Server (awkward configuration) and Tomcat 4.1.24 (too heavyweight).

For the *surf* module, we chose the Jakarta HttpClient 2.0 Alpha 2. Other alternatives included the JDK URLConnection (HTTP Response Codes $>$ 400 unsupported), the Innovation HttpClient 0.3-3 (not maintained), httpUnit (automatic frameset processing) and htmlUnit (no access to HTTP messages).

For the instrumentation of XH files via DOM Operations and the XSLT transformation in the *extract* component, we chose DOM4j 1.3. Other alternatives included Java DOM (slow, lack of features), JDOM Beta 8 (lack of XSLT) and XOM (one-man project).

6 Evaluation

To evaluate `wal`, we generated some 20 atomic wrappers for extractions from text and record pages. Logic pages usually require compound wrappers. We show the extraction of the top story on CNN in detail.

After surfing to `http://www.cnn.com`, the user initiates the authoring process by clicking on the *record* bookmarklet. By clicking in the document, the

(A) initial state

(B) click selects headline element

(C,D) successive clicks enlarge the selection

(E) final selection

(F) confirmation from server

Fig. 10. Sample selection process

(A) Naming the wrapper

(B) Generation is complete.

Fig. 11. Save dialog

user builds a selection on the client side (A-E in fig. 10). The *select* bookmarklet transmits the selection to the server and triggers a visual confirmation (F).

The *save* bookmarklet stores the HTTP request and the extraction as a named wrapper (see fig. 11). It can subsequently be invoked as a web service.

7 Conclusions

With `wal`, we demonstrate that authoring on the visualization level offers great usability. Compared to writing custom extractors in XSLT, regular expressions or proprietary languages, the visual approach makes wrapper developer's lives a lot easier. Download `wal` to find out for yourself [33].

That a direct correspondence between selections and path expressions is sufficient to perform many extraction tasks underlines our initial arguments for describing extractions on the AST level.

Certainly, `wal` leaves much to be desired. From a theoretical viewpoint, it would be interesting to apply more sophisticated machine learning approaches to authoring. From the application viewpoint, an extension to compound wrappers is sorely needed as multi-page navigation sequences are required to operate logic pages like web mailers or auction sites. From a technical viewpoint, `wal` needs support for SSL connections and cookie-based session management.

But the existing system is a good vantage point for such extensions. Due to reuse and modularity, `wal` profits immediately from improvements in the off-the-shelf components used, e.g., HTML parsers, web clients and web servers.

As automation permeates the web, wrapper generation in the large may become an interesting subject for future research. this includes long-term studies of wrapper failure rates and the development of wrapper recovery strategies.

References

1. Cruz, I.F., ed.: International Semantic Web Working Symposium (SWWS), Stanford University, Palo Alto, California, USA (2001)
2. Berners-Lee, T., Hendler, J., Lassila, O.: The semantic web. Scientific American (2001)
3. Kuhlins, S., Tredwell, R.: Toolkits for generating wrappers. In Aksit, M., Mezini, M., Unland, R., eds.: Proc. NetObjectDays 2002. Volume 2591 of LNCS., Erfurt, Germany, Springer (2003) 184–198
4. Sahuguet, A., Azavant, F.: Web ecology: Recycling HTML pages as XML documents using w4f. In: WebDB (Informal Proceedings). (1999) 31–36
5. O'Reilly: LAMP. `http://www.onlamp.net/` (2002)
6. E-Soft Inc.: Security space internet research reports. `http://www.securityspace.com/s_survey/data/index.html` (2003)
7. Myllymaki, J.: Effective web data extraction with standard XML technologies. In: World Wide Web. (2001) 689–696
8. Raggett, D., Hors, A.L., Jacobs, I., eds.: HTML 4.01 Specification. W3C, `http://www.w3.org/TR/html4/` (1999)

9. Fittkau, Maass: 15. WWW-Benutzer-Analyse W3B. http://www.w3b.org/trends/browserwatch.html (2003)
10. ISO: Information processing – text and office systems – standard generalized markup language (SGML). ISO 8879:1986(E) (1986)
11. Bray, T., Paoli, J., Sperberg-McQueen, C., eds.: Extensible Markup Language (XML) 1.0. W3C, http://www.w3.org/TR/1998/REC-xml-19980210 (1998)
12. Pemberton, et al.: XHTML 1.0 The Extensible HyperText Markup Language (Second Edition). W3C, http://www.w3.org/TR/2002/REC-xhtml1-20020801 (2002)
13. Laender, Ribeiro-Neto, Silva, Teixeira: A brief survey of web data extraction tools. In: SIGMOD Record. Volume 31. (2002)
14. Tredwell, R., Kuhlins, S.: Wrapper development tools. http://www.wifo.uni-mannheim.de/~kuhlins/wrappertools/ (2003)
15. Compaq: Web language. http://research.compaq.com/SRC/WebL/ (1999)
16. Crescenzi, V., Mecca, G.: Grammars have exceptions. Information Systems **23** (1998) 539–565
17. Huck, G., Fankhauser, P., Aberer, K., Neuhold, E.J.: Jedi: Extracting and synthesizing information from the web. In: CoopIS 1998. (1998) 32–43
18. Liu, L., Pu, C.: XWrap. http://www.cc.gatech.edu/projects/disl/XWRAPElite/ (2000)
19. Laender, Ribeiro-Neto: DEbyE. http://www.lbd.dcc.ufmg.br/~debye/ (2002)
20. Miller, Myers: LAPIS. http://www-2.cs.cmu.edu/~rcm/papers/chi02/chi02.html (2001)
21. Republica Corp.: X-Fetch Wrapper. http://www.x-fetch.com/wrapper.html (2003)
22. Font, F.: Small HTML parser. http://www.room4me.com/software/SmallHTMLParser.htm (2002)
23. Rodriguez, J.N.: Nazareno. http://www.geocities.com/jesus2nyc/index.html (2002)
24. Cowan, J.: Tagsoup. http://mercury.ccil.org/~cowan/XML/tagsoup (2002)
25. Clark, A.: CyberNeko tools for XML. http://www.apache.org/~andyc/neko/doc/index.html (2003)
26. Tripp, A., et al.: JTidy. http://www.sf.net/projects/jtidy/ (2001)
27. Raggett, D.: Html tidy. http://tidy.sf.net/ (2003)
28. SUN Microsystems: Package javax.swing.text.html.parser. http://java.sun.com/j2se/1.4.2/docs/api/javax/swing/text/html/parser/pa%ckage-summary.html (2003)
29. Clark, J., ed.: XSL Transformations (XSLT) Version 1.0. W3C, http://www.w3.org/TR/xslt (1999)
30. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Software Components. Addison-Wesley (1995)
31. Kanga, S.: Bookmarklets. http://www.bookmarklets.com (2003)
32. Völkel, M.: Extraktion von XML aus HTML-Seiten. Diplomarbeit, Universität Karlsruhe, IPD Goos (2003)
33. Völkel, M.: Web abstraction layer WAL. http://wal.sf.net (2003)