

Implementing Logic Wrappers Using XSLT Stylesheets*

Amelia Bădică

University of Craiova, Business Information Systems Department
A.I.Cuza 13, Craiova, RO-200585, Romania
ameliabd@yahoo.com

Costin Bădică, Elvira Popescu

University of Craiova, Software Engineering Department
Bvd.Decebal 107, Craiova, 200440, Romania
{badica_costin, popescu_elvira}@software.ucv.ro

Abstract

The Web has become a major source of information that is easily accessible at low cost by individual and business consumers. Logic wrappers are a new technology that was proposed to help automatizing the task of data extraction from the Web. In this note we present an approach for efficiently implementing logic wrappers with the help of XSLT transformation language. The approach was successfully applied in various application areas: collecting product features from product information sheets and mining travel resources as found on Web sites of online transaction brokers.

1. Introduction

The Web was designed for human consumption rather than machine processing. Web pages are designed by humans and are targeted to human consumers that seek specialized information in various areas of interest. That information can be reused for different problem solving purposes; in particular it can be searched, filtered out, processed, analyzed, and reasoned about.

Web data sources are in fact networked electronic documents written in HTML or XML that can be characterized as neither natural language, nor structured (usually, the term semi-structured data is used to characterize them). Many Web data sources can be nicely abstracted as providing relational data as sets of relational tuples. Examples include: search engines result pages, product catalogues, news

sites, product information sheets, travel resources, multimedia repositories, Web directories, a.o.

Logic wrappers (L-wrappers hereafter) are a new technology for constructing wrappers for relational data extraction from the Web. This technology borrows ideas from the areas of logic programming and inductive learning [6, 8].

L-wrappers have a declarative semantics, and therefore their specification is decoupled from their implementation. L-wrappers can be semi-automatically generated using inductive logic programming. In this paper we describe an approach for the efficient implementation of L-wrappers using XSLT transformation language ([7]) – a standard language for processing XML documents.

The paper is structured as follows. Section 2 introduces L-wrappers and XSLT₀ transformation language. Section 3 describes the algorithm for translating L-wrappers into XSLT₀ programs. Section 4 illustrates the translation on a simple example and briefly discusses a more realistic experiment. The last section concludes.

2. Background: L-Wrappers and XSLT₀

2.1. L-Wrappers

HTML is the *lingua franca* for Web publishing. An HTML document can be transformed into an well-formed (i.e tree-structured) XML document expressed in XHTML. Therefore, we can safely assume that Web data sources are modeled as labeled ordered trees.

We adopt a standard relational model, i.e. we associate to each Web data source a set of distinct attributes. A wrapper is a program that takes a labeled ordered tree and returns a subset of tuples of extracted nodes. L-wrappers are sets of patterns defined as logic rules that can be learned by apply-

* This work was carried out as part of the CNCSIS grant 185/2006: "Technologies and Intelligent Software Tools for Automated Construction of E-Catalogues of Products Using Knowledge Acquisition from the Web"

ing general-purpose inductive logic programming systems (like FOIL [9] or Aleph [1]).

Labeled ordered trees are abstracted as sets of nodes on which certain relations and functions are defined. Basically, we define two relations between tree nodes: the "parent-child" relation and the "next-sibling" linear ordering relation on the set of children of a node. Furthermore, a label is attached to each tree node, modeling a specific tag from a finite set of tag symbols Σ .

An XHTML document is composed of a structural part and a content part. The structural part consists of a labeled ordered tree of document nodes or elements. The content part of a document consists of the actual text in the text elements. We assume that there is a special tag $t \in \Sigma$ that designates a text element (in this way we treat text and element nodes in an uniform way).

Figure 1 shows a labeled ordered tree with 13 nodes and tags in the set $\Sigma = \{a, b, c, d, t\}$. Nodes labeled t denote text elements and their content is shown as text labels linked to nodes via dotted lines.

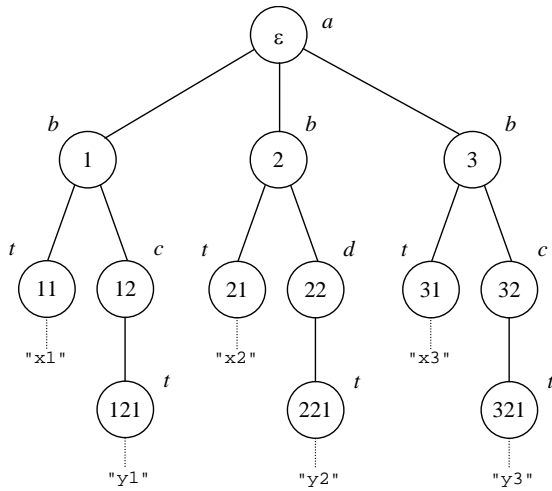


Figure 1. Document as labeled ordered tree

Within this framework, a pattern is a directed graph with labeled arcs and vertices. Arc labels denote conditions that specify the tree delimiters of the extracted information, according to the parent-child and next-sibling relationships (eg. is there a parent node ?, is there a left sibling ?, a.o). Vertex labels specify conditions on nodes (eg. is the tag label td ?, is it the first child ?, a.o). A subset of graph vertices is used for selecting the items for extraction.

Intuitively, an arc labeled ' n ' denotes the "next-sibling" relation while an arc labeled ' c ' denotes the "parent-child" relation. As concerning vertex labels, label ' f ' denotes "first child" condition, label ' l ' denotes "last child" condition and

label $\sigma \in \Sigma$ denotes "equality with tag σ " condition.

Patterns are matched against parts of a target labeled ordered tree. A successful matching asks for the labels of pattern vertices and arcs to be consistent with the corresponding relations and functions over tree nodes. The result of applying a pattern to a labeled ordered tree is a set of extracted node tuples.

An L-wrapper can be defined as a set of patterns that share the set of attributes from the relation scheme of the information source. In this paper we restrict our attention to single-pattern L-wrappers. They can be concisely defined in two steps: i) define the pattern graph together with arc labels that model parent-child and next-sibling relations and ii) extend this definition with vertex labels that model conditions on vertices, extraction vertices and assignment of extraction vertices to attributes.

Definition 1 (Pattern graph) Let \mathcal{W} be a set denoting all vertices. A pattern graph G is a quadruple $\langle A, V, L, \lambda_a \rangle$ such that $V \subseteq \mathcal{W}$, $A \subseteq V \times V$, $L \subseteq V$ and $\lambda_a : A \rightarrow \{c', n'\}$. The set \mathcal{G} of pattern graphs is defined inductively as follows:

i) If $v \in \mathcal{W}$ then $\langle \emptyset, \{v\}, \{v\}, \emptyset \rangle \in \mathcal{G}$

ii) If $G = \langle A, V, L, \lambda_a \rangle \in \mathcal{G}$, $v \in L$, and $w, u_i \in \mathcal{W} \setminus V$, $1 \leq i \leq n$ then a) $G_1 = \langle A \cup \{(w, v)\}, V \cup \{w\}, (L \setminus \{v\}) \cup \{w\}, \lambda_a \cup \{((w, v), 'n')\} \rangle \in \mathcal{G}$; b) $G_2 = \langle A \cup \{(u_1, v), \dots, (u_n, v)\}, V \cup \{u_1, \dots, u_n\}, (L \setminus \{v\}) \cup \{u_1, \dots, u_n\}, \lambda_a \cup \{((u_1, v), 'c'), \dots, ((u_n, v), 'c')\} \rangle \in \mathcal{G}$; c) $G_3 = \langle A \cup \{(w, v), (u_1, v), \dots, (u_n, v)\}, V \cup \{w, u_1, \dots, u_n\}, (L \setminus \{v\}) \cup \{w, u_1, \dots, u_n\}, \lambda_a \cup \{((w, v), 'n'), ((u_1, v), 'c'), \dots, ((u_n, v), 'c')\} \rangle \in \mathcal{G}$

Intuitively, if $\langle A, V, L, \lambda_a \rangle$ is a pattern graph then V are its vertices, A are its arcs, $L \subseteq V$ are its leaves (vertices with in-degree 0) and λ_a indicates parent-child and next-sibling arcs. Note also that a pattern graph is tree-shaped with arcs pointing up.

Definition 2 (Single-pattern L-wrapper) Let \mathcal{A} be the set of attribute names. A single-pattern L-wrapper is a tuple $W = \langle V, A, U, D, \mu, \lambda_a, \lambda_c \rangle$ such that $\langle A, V, L, \lambda_a \rangle$ is a pattern graph, $U = \{u_1, u_2, \dots, u_k\} \subseteq L$ is the set of pattern extraction vertices, $D \subseteq \mathcal{A}$ is the set of attribute names, $\mu : D \rightarrow U$ is a one-to-one function that assigns a pattern extraction vertex to each attribute name, and $\lambda_c : V \rightarrow \mathcal{C}$ is the labeling function for vertices. $\mathcal{C} = \{\emptyset, \{f'\}, \{l'\}, \{\sigma\}, \{f', l'\}, \{f', \sigma\}, \{l', \sigma\}, \{f', l', \sigma\}\}$ is the set of conditions, where σ is a label in the set Σ of tag symbols.

Note that, according to definition 2, we assume that extraction vertices are among the leaves of the pattern graph.

Figure 2 shows an extraction path. The extraction vertices are marked with small arrows (vertices F and D on that figure). Note also that the figure shows the attributes extracted by the extraction vertices (attribute x extracted by

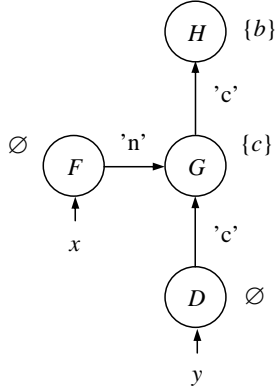


Figure 2. A single-pattern L-wrapper

vertex F and attribute y extracted by vertex D). A pair of nodes (n_F, n_D) is extracted by this wrapper if and only if it has the following properties: i) n_F has tag t ; ii) parent of node n_D has tag c ; iii) next sibling of n_F is parent of n_D ; iv) grandparent of node n_D has tag b .

2.2. XSLT₀ Transformation Language

XSLT₀ is a subset of XSLT that retains most of its features but additionally has a formal operational semantics, and a cleaner and more readable syntax.

In what follows we just briefly review XSLT₀ and its pseudocode notation. For more details on its abstract model and formal semantics, the reader is invited to consult reference [4].

An XSLT₀ program is a set of rules. A rule can be either selecting or constructing. In what follows we are focusing only on constructing rules, as we are using only constructing rules in translating L-wrappers into XSLT₀.

A (q, σ) constructing rule has the following form:

```

template  $q(\sigma, x_1, \dots, x_n)$ 
  vardef
     $y_1 := r_1; \dots; y_m := x_m$ 
  return
    if  $c_1$  then  $z_1; \dots; \mathbf{if}$   $c_k$  then  $z_k$ 
end

```

Here:

- q is an XSLT mode (actually a constructing mode) and σ is either a tag symbol in Σ or $*$
- $x_1, \dots, x_n, y_1, \dots, y_m$ are variables
- Each r_i is an expression (possibly involving variables $x_1, \dots, x_n, y_1, \dots, y_{i-1}$) that evaluates to a data value (i.e. the value of an attribute or the content of an element)

- Each c_i is a test (possibly involving variables $x_1, \dots, x_n, y_1, \dots, y_m$) and thus it evaluates either to true or false
- Each z_i is a forest (i.e. a (possibly empty) sequence of tree fragments) that is created by the constructing rule. The leaves of this forest are expressions of the form $q'(p, \bar{z})$ such that q' is a constructing mode, p is an XPath pattern ([2]) (possibly with variables), and \bar{z} is a sequence (possibly empty) of variables from the set $\{x_1, \dots, x_n, y_1, \dots, y_m\}$.

Additionally, we require the existence of a constructing mode *start* such that each z_i of a $(start, \sigma)$ constructing rule is a tree (rather than a forest). This constraint ensures that the output of an XSLT₀ program is always a tree. Also, if none of the tests c_i succeeds, the empty forest will be output. Finally, to ensure that the model is deterministic, we require that for any two (q_1, σ_1) and (q_2, σ_2) rules either $q_1 \neq q_2$ or $q_1 = q_2$ and $\sigma_1 \neq \sigma_2$ and $\sigma_1 \neq *$ and $\sigma_2 \neq *$.

The computation defined by an XSLT₀ program transforms an input labeled ordered tree t into an output tree. The computation can be seen as a sequence of steps.

At each step the state of the computation is recorded as a tree such that some of its leaves are configurations (all the rest of nodes are tag symbols). A configuration is a triple (u, q, \bar{d}) such that u is a node of tree t , q is a mode and \bar{d} is a sequence of data values. The initial state is a tree made-up of a single leaf node $(r(t), start, \epsilon)$ ($r(t)$ is the root node of tree t).

Let us assume that current state contains a configuration (u, q, \bar{d}) as one of its leaf nodes. A step consists in transforming the current tree into a new tree by applying a (q, σ) rule such that either σ equals the label of u or $\sigma = *$. The intuition behind the application of a constructing rule is as follows. First, variables y_i and conditions c_j are evaluated. Let us assume that, as result of evaluating tests c_j -s, forest z with leaves of the form $q'(p, \bar{z})$, is returned. Second, pattern p is applied to node u of t yielding a sequence of nodes u_1, \dots, u_l in document order. Third, a new forest f is computed by substituting the variables in z with their values and leaves $q'(p, \bar{z})$ of z with sequences of configurations $(u_1, q', \bar{d}'), \dots, (u_l, q', \bar{d}')$. Here, \bar{d}' are the new values assigned to variables x_i and y_j . Fourth, next state is computed by substituting configuration (u, q, \bar{d}) of the current state with f . The transformation stops when the current state is a labeled ordered tree (i.e it does not contain any configurations as leaves). The result of a computation is the tree representing the its state.

3. Mapping L-Wrappers to XSLT₀

Let $W = \langle V, A, U, D, \mu, \lambda_a, \lambda_c \rangle$ be a single-pattern L-wrapper and let $L \subseteq V$ be the set of leaves of its pattern graph. Recall that we assumed that all extraction vertices

are in L , i.e. $U \subseteq L$. Note that if u and v are vertices of the pattern graph then $u \rightsquigarrow v$ denotes the path from vertex u to vertex v in this graph.

Let $L = \{v_1, \dots, v_n\}$ be the leaves and let v of the pattern graph. The idea of the translation algorithm is as follows. We start from root w and move down in the graph to w_1 , i.e. $w_1 \rightsquigarrow w$. Then we move from w_1 to w_2 via their first common ancestor w'_1 i.e. $w_1 \rightsquigarrow w'_1$ and $w_2 \rightsquigarrow w'_1$, ..., and we move from w_{n-1} to w_n via their first common ancestor w'_{n-1} i.e. $w_{n-1} \rightsquigarrow w'_{n-1}$ and $w_n \rightsquigarrow w'_{n-1}$.

Template rules are generated according to this traversal pattern of the pattern graph. The first rule is generated according to path $w_1 \rightsquigarrow w$. The next $n - 1$ rules are generated according to paths $w_i \rightsquigarrow w'_i$ and $w_{i+1} \rightsquigarrow w'_i$, $1 \leq i \leq n - 1$. Finally, the last rule is generated for vertex w_n . Therefore, a total of $n + 1$ rules are generated.

The resulting GEN-WRAPPER translation algorithm is shown below. Note that function VAR-GEN generates a new variable name based on the attribute associated to an extraction vertex.

GEN-WRAPPER(W)

1. **let** $w \in L_W$
2. $L \leftarrow L_W \setminus \{w\}$
3. GEN-FIRST-TEMPLATE(w)
4. **while** $L \neq \emptyset$ **do**
5. $w_0 \leftarrow w$
6. $V \leftarrow \emptyset$
7. let w' be another vertex in $L \setminus \{w_0\}$
6. let w be the first common ancestor of w_0 and w'
9. **if** $w_0 \in U_W$ **then**
10. $var \leftarrow \text{VAR-GEN}(\mu_W(w_0))$
11. GEN-TEMPLATE-WITH-VAR($w_0 \rightsquigarrow w$,
12. $w' \rightsquigarrow w, var, V$)
13. $V \leftarrow V \cup \{var\}$
14. **else**
15. GEN-TEMPLATE-NO-VAR($w_0 \rightsquigarrow w$,
16. $w' \rightsquigarrow w, V$)
17. $w \leftarrow w'$
18. $L \leftarrow L \setminus \{w\}$
19. GEN-LAST-TEMPLATE(w, V)

GEN-FIRST-TEMPLATE algorithm generates the first template rule. Let p_1 be an XPath pattern that accounts for the conditions on the path $w_1 \rightsquigarrow w$. Then, the template rule that is generated first takes the following form:

```

template start(/)
  return
    result(selw1(p1))
end

```

GEN-TEMPLATE-WITH-VAR and GEN-TEMPLATE-NO-VAR algorithms generate a template rule for paths $w_i \rightsquigarrow w'_i$ and $w_{i+1} \rightsquigarrow w'_i$, $1 \leq i \leq n - 1$, depending if ver-

tex w_i is an extraction vertex or not. Note that a new variable is generated only if w_i is an extraction vertex. Let p_{i+1} be the XPath pattern that accounts for the conditions on the paths $w_i \rightsquigarrow w'_i$ and $w_{i+1} \rightsquigarrow w'_i$.

The structure of a template rule for the case when a new variable is generated is shown below.

```

template selwi(*, V)
  vardef
    var := content(.)
  return
    selwi+1(pi+1, V ∪ {var})
end

```

The structure of a template rule for the case when a new variable is not generated is shown below.

```

template selwi(*, V)
  return
    selwi+1(pi+1, V)
end

```

GEN-LAST-TEMPLATE algorithm generates the last template rule. The constructing part of this rule fully instantiates the returned tree fragment, thus stopping the transformation process of the input document tree. Depending if vertex w_n is an extraction vertex or not, this template rule generates or not a new variable. We assume that the set D of attribute names is $\{d_1, \dots, d_m\}$. Note that because a new variable is generated for each extraction vertex, it follows that the number of generated variables is m .

If a new variable is generated, the last generated rule has the following form:

```

template selwn(*, V)
  vardef
    var := content(.)
  return
    tuple((d1(var1), ... dm(varm)))
end

```

Here $V \cup \{var\} = \{var_1, \dots, var_m\}$.

If a new variable is generated, the last generated rule has the following form:

```

template selwn(*, V)
  return
    tuple((d1(var1), ... dm(varm)))
end

```

Here $V = \{var_1, \dots, var_m\}$.

4. Example and Experiment

4.1. Example

Applying algorithm GEN-WRAPPER to the L-wrapper shown in figure 2, we obtain the following XSLT₀ program comprising three rules:

```

template start()
  return
    result((selx(p1)))
end

template selx(*)
  vardef
    vx := content(.)
  return
    sely(p2, vx)
end

template sely(*, vx)
  vardef
    vy := content(.)
  return
    tuple(x(vx), y(vy))
end

```

XPath pattern $p_1 = //b/c/preceding-sibling::*[1]$ is determined by tracing the path $H \rightarrow G \rightarrow F$ in the pattern graph (see figure 2).

XPath pattern $p_2 = following-sibling::*[1]/*$ is determined by tracing the path $F \rightarrow G \rightarrow D$ in the pattern graph (see figure 2).

Note that executing this XSLT₀ program on the labeled ordered tree shown in figure 1 produces the following XML comprising extracted tuples:

```

<?xml version="1.0" encoding="utf-8"?>
<result>
  <tuple>
    <x>x1</x>
    <y>y1</y>
  </tuple>
  <tuple>
    <x>x3</x>
    <y>y3</y>
  </tuple>
</result>

```

4.2. Experiment

We have successfully applied the L-wrappers technology to i) collecting product descriptions from product information sheets [5] and ii) mining travel resources on Web sites of online transaction brokers [8].

We now demonstrate the use of L-wrappers to extract travel information from the Travelocity Web site¹ (see figure 3). That Web page displays hotel information comprising the hotel name, address and description, the check-in and check-out dates, the types of rooms offered and the corresponding average nightly rate. Adopting the relational model, we associate to this resource

¹ <http://www.travelocity.com>

Room Type	Wed	Thu	Avg Nightly Rate*	
Standard Room	\$216.54	\$216.54	\$216.54	Select

Room Type	Wed	Thu	Avg Nightly Rate*	
Standard Room	\$211.92	\$211.92	\$211.92	Select
Deluxe Room	\$267.15	\$267.15	\$267.15	Select

Figure 3. An XHTML document fragment and its graphic view

the following set of attribute names related to hotels: {name, address, description, period, roomtype, price}.

We have used FOIL to learn a single pattern L-wrapper for this example. The resulting rule was then represented as a pattern graph. Applying the translation algorithm outlined in section 3, a set of 7 template rules resulted (see table 1).

For wrapper execution we can use any of the available XSLT transformation engines. In our experiments we have used Oxygen XML editor ([3]), a tool that incorporates some of these engines. The experimental results confirmed the efficacy of the approach: values 0.87 and 1 were recorded for precision and recall measures.

5. Conclusions

This paper discusses the translation of L-wrappers to a subset of XSLT that has a formal semantics. This translation enables the efficient execution of L-wrappers using available XSLT transformation engines. We plan to implement this technique in a tool for data extraction from the Web. As future theoretical work, we would like to give a formal proof of the correctness of this mapping of L-wrappers to XSLT.

References

- [1] Aleph. <http://web.comlab.ox.ac.uk/oucl/research/areas/machine-learning/Aleph/aleph.html>.
- [2] XML Path Language (XPath) Version 1.0. <http://www.w3.org/TR/xslt>.

```

template start(html)
  return
    result(selclass(xp1))
end

template selperiod(*,varName)
  vardef
    varPeriod := content(.)
  return
    seldescription(xp3,varName,varPeriod)
end

template seladdress(*,varName,varPeriod,
varDescription)
  vardef
    varAddress := content(.)
  return
    selroomtype(xp5,varName,varPeriod,
varDescription,varAddress)
end

template selname(*)
  vardef
    varName := content(.)
  return
    selperiod(xp2,varName)
end

template seldescription(*,varName,varPeriod)
  vardef
    varDescription := content(.)
  return
    seladdress(xp4,varName,varPeriod,
varDescription)
end

template selroomtype(*,varName,varPeriod,
varDescription,varAddress)
  vardef
    varRoomtype := content(.)
  return
    display(xp6,varName,varPeriod,
varDescription,varAddress,varRoomtype)
end

template display(*,varName,varPeriod,varDescription,varAddress,varRoomtype)
  vardef
    varPrice := content(.)
  return
    tuple[name → varName; period → varPeriod; description → varDescription;
address → varAddress; roomtype → varRoomtype; price → varPrice]
end

xp1 = /*/*/following-sibling::*[1]/span/text()
xp2 = parent::*parent::*parent::*preceding-sibling::*[1]/parent::*parent::*
following-sibling::*[2]/*/*/*/*/*/*/*/*following-sibling::*[1][local-name()='b']/*
xp3 = parent::*preceding-sibling::*[1]/parent::*parent::*parent::*parent::*
parent::*preceding-sibling::*[2]/parent::*/*/*/*/*/*/*/*[position()=1]/*/*/*text()
xp4 = parent::*parent::*preceding-sibling::*[1][position()=1]/*[position()=1 and
local-name()='td' and following-sibling::*[1]]
xp5 = parent::*following-sibling::*[1]/parent::*parent::*parent::*parent::*
parent::*parent::*parent::*parent::*following-sibling::*[2]/*/*/*following-sibling::*[1]/*
/*following-sibling::*[1]/*[position()=1 and following-sibling::*[1]]/*
xp6 = parent::*following-sibling::*[1]/parent::*[position()=last()-1]/text()

```

Table 1. Description of the sample wrapper in XSLT₀ pseudocode

- [3] Oxygen XML Editor. <http://www.oxygenxml.com/>.
- [4] G. Bex, S. Maneth, and F. Neven. A formal model for an expressive fragment of xslt. *Information Systems*, (27):21–39, 2002.
- [5] C. Bădică and A. Bădică. Rule learning for feature values extraction from html product information sheets. In H. Boley and G. Antoniou, editors, *Proc. RuleML 2004, LNCS 3323*, pages 37–48, Hiroshima, Japan, 2004. Springer Verlag.
- [6] C. Bădică and A. Bădică. Logic wrappers and xslt transformations for tuples extraction from html. In S. Bressan, S. Ceri, E. Hunt, Z. Ives, Z. Bellahsene, M. Rys, and R. Unland, editors, *Proc. 3rd International XML Database Symposium XSym'05, LNCS 3671*, pages 177–191, Trondheim, Norway, 2005. Springer Verlag.
- [7] J. Clark. Xslt transformation (xslt) version 1.0. W3c recommendation, 1999.
- [8] E. Popescu, A. Bădică, and C. Bădică. Mining travel resources on the web using l-wrappers. In *Proc. ICAISC 2006, LNAI (to appear)*, Zakopane, Poland, 2006.
- [9] J. R. Quinlan and R. M. Cameron-Jones. Induction of logic programs: Foil and related systems. *New Generation Computing*, (13):287–312, 1995.