Evolving a Set of DTDs according to a Dynamic Set of XML Documents

Elisa Bertino[1] *Giovanna Guerrini*[2] *Marco Mesiti*[3] *Luigi Tosetto*[3]

[1] Dipartimento di Scienze dell'Informazione - Università di Milano
Via Comelico 39 - I20135 Milano, Italy
bertino@dsi.unimi.it
[2] Dipartimento di Informatica - Università di Pisa
Corso Italia 40 - I56125 Pisa, Italy
guerrini@disi.unige.it
[3] Dipartimento di Informatica e Scienze dell'Informazione - Università di Genova
Via Dodecaneso 35 - I16146 Genova, Italy
mesiti@disi.unige.it, luigi.tosetto@inwind.it

Abstract. In this paper we address the problem of evolving a set of DTDs so to obtain a description as precise as possible of the structures of the documents actually stored in a source of XML documents. This problem is highly relevant in such a dynamic and heterogeneous environment as the Web. The approach we propose relies on the use of a classification mechanism based on document structure and on the use of data mining association rules to find out frequent structural patterns in data.

1 Introduction

XML [8] has recently emerged as the most relevant standardization effort for document representation and exchange on the Web. XML documents share most of the features of semi-structured data [1], but they also have some peculiarities, such as the notion of DTD. At the same time, databases are more and more often integrated with Web applications. The availability of a unified format for data interchange on the Web and for the representation in the database is therefore very attractive. It avoids all intermediate steps for converting and preparing data, taking advantage of the presence, in XML documents, of both the structure and the meaning of data.

A database schema provides all the information on data structure and semantics thus allowing the organization of data so that database operations can be executed efficiently. XML documents stored in a database are structured according to the hierarchical composition of elements specified by a set of DTDs. This structure can evolve over time. The need of maintaining different information in a document or of organizing the same information in different ways can indeed lead to modifications to the document structure. A document structure can vary for several reasons. First, the real world (i.e., the application domain) reflected in the document can change. Second, a change of the document structure can

be decided by the document producer, because the user interests in data represented in the document change. Moreover, in such a flexible and heterogeneous environment as the Web, it is not realistic to fix the structure of a document once for all without the possibility of subsequent evolutions, nor to fix it in a centralized manner, so that anybody else producing documents of that kind will adhere to it.

Changes in document structures obviously have a major impact on the database storing the documents. If document structures evolve the database storing the documents can neither be efficient, in that an efficient storing and a fast access are no longer possible, nor correct, in that the database schema no longer reflects document structure and semantics.

In this paper, we propose an approach to evolve a set of DTDs, representative of the documents already stored in a database, so to adapt it to the structure of new documents entering the database. To obtain a representative structure of a set of documents, the first step is to group together similar documents. Therefore, each document entering the database is classified against the set of DTDs the database schema consists of, to determine the DTD in the set best describing the structure of the document. A possibility is to use validators in this preliminary classification phase. This approach, however, has the drawback that classification based on validators is very rigid, with a boolean answer. Requiring the validity of each document entering the database with respect to a DTD in the schema would lead, in such a heterogeneous environment as the Web, to reject a large amount of documents, thus resulting in a considerable loss of information. Thus, we rely on a more flexible classification approach [2], based on an algorithm to measure the structural similarity between a document and a DTD that produces a numeric rank in the range $[0, 1]$. Actually, in this paper we rely on a slightly modified version of the algorithm that allows one to obtain a numeric evaluation of the structural similarity of each single element in a document with respect to the corresponding element declaration in the DTD. This measure will be referred to as *local similarity* and allows one to support an evolution procedure with a variable level of granularity, ranging from the entire document to a single element.

The scenario we refer to in this paper is the following. We consider a source of XML documents, gathered from the Web or created as local data, and a set of DTDs. These DTDs can be extracted from a sample document set, or they can be defined by an expert of the application domain. Through the classification approach proposed in [2], each document, created outside the source, can be associated with a DTD in the source, the one best describing its structure. If a document, matched against each DTD in the source, does not produce a similarity value above a fixed threshold, it is stored in a separate *repository*, containing unclassified documents. Otherwise, the document is handled as an instance of the DTD for which the evaluation produced the highest similarity value. After having classified a certain number of documents, the documents instances of a DTD can present some regularities that, if captured by the DTD, would restrict the divergence between the structure of documents as specified

by the DTD and the actual structures of documents instances of the DTD. The goal of the evolution approach we propose is to capture these regularities thus adapting the set of DTDs to the set of documents. A variable to be taken into account in the evolution process is how to weight new documents with respect to the already classified ones, in order to obtain a set of DTDs describing the structure of most documents at a level of detail accurate enough.

The evolution process consists of two phases: a *recording phase* in which peculiarities of documents entering the source are acquired, for the portions of documents not conforming to the corresponding DTD, and an *evolution phase*, in which the recorded information are used to update the schema and to adapt it to the actual population of the source. The recorded information are associated with each single element of each DTD, in ad-hoc data structures. The evolution phase is based on the use of data mining association rules [4] to find out frequent structural patterns in documents.

The paper is structured as follows. Section 2 presents an overview of our approach. Section 3 discusses the recording phase. Section 4 is devoted to the evolution phase, whereas Section 5 discusses related work. Finally, Section 6 concludes the paper.

2 Overview of the Approach

Given a dynamic set of XML documents and a set of DTDs we aim at obtaining a new set of DTDs reflecting as much as possible the actual structure of documents. Guided by *exceptions*, that is, data that are not captured by the current schema, the schema itself is updated by means of heuristic policies. The goal is thus to obtain a new set of DTDs, originated from the previous set of DTDs updated so that deviations in the structures of documents in the source are taken into account. After having classified a certain number of documents, the documents instances of a DTD can present some regularities that, if captured by the DTD, would restrict the divergence between the structure of documents as specified by the DTD and the actual structures of documents instances of the DTD. These regularities can be summarized as follows:

- Some documents miss some elements specified in the DTD. In this case, in the evolution process, by taking into account the percentage of documents missing the element, some elements can be removed from the DTD.
- Some documents contain some *new* elements, not defined in the DTD. In this case, in the evolution process, these elements, for which the correct definition must be deduced, will be inserted in the DTD.
- Elements in the document and in the DTD match, but the underlying structures do not, that is, the constraints defined by operators in the DTD are not met. In this case, the portion of the DTD which does not conform to the operators must be rebuilt, generating a new structure binding elements together.

The evolution process cannot be started for each DTD instance which is not valid for the DTD. The evolution has an impact on applications working

on the source, and should thus be performed whenever the source contains a certain amount of documents that are not valid for the DTD with which they have been associated, and that present some regularities. The exact number of such documents depends on the features of document themselves and on the application domain, that influences the precision level at which the structure of documents in the source should be captured. There is an obvious trade-off between the frequency and the precision of the evolution process on one side and its cost, both in terms of execution time and storage space required for maintaining auxiliary information, on the other side.

The approach we take is therefore the following. After having classified each document, some structural information of the document are extracted (*recording phase*). These information are then used during the *evolution phase*. The recording phase allows one to carry on the evolution phase without need of analyzing again the documents, since information that are relevant for the evolution have already been associated with the DTD. Such an approach results in a faster evolution phase, even though it requires some storage space.

The need of evolving the set of DTDs can arise for both classified and unclassified documents (i.e., also for documents stored in the repository). In the following we do not address the problem of generating a DTD from documents with similar structures in the repository, rather we focus on the evolution of a DTD based on its instances. Documents in the repository cannot take advantage of the presence of a DTD in the source. Therefore, for such documents our approach or other approaches already developed for extracting structural information from the documents, as those described in Section 5, can be equivalently applied.

A graphical overview of our approach is shown in Figure 1, in which rectangles denote the main functional components of the approach, cylinders denote data stores, thick arrows denote the control flow[1], and thin arrows denote data flow. In what follows we briefly describe each phase of the approach. The *initialization* phase consists of the definition of the initial set of DTDs and of the setting of the similarity threshold σ. This threshold determines how close the structure of the documents classified in a DTD is to the structure specified by the DTD.

Each time a document, created outside the source, enters the source it is initially inserted in a queue of "to-be-processed" documents. When it is then selected, it is associated with a DTD in the source, that is, the one best describing its structure, through the *classification* algorithm proposed in [2]. If a document, matched against each DTD in the source, does not produce a similarity value above the threshold σ, it is inserted in a separate *repository*, containing unclassified documents. Otherwise, the document is handled as an instance of the DTD for which the evaluation produced the highest similarity value.

[1] Note that there are two iterated activities: one involving the *classification* and *check* phases, corresponding to the fact that new documents are classified till the evolution process is triggered, and a second one corresponding to the iteration of the whole evolution approach over time. This cycle includes all the activities in our approach, but the ones in the *initialization* phase.

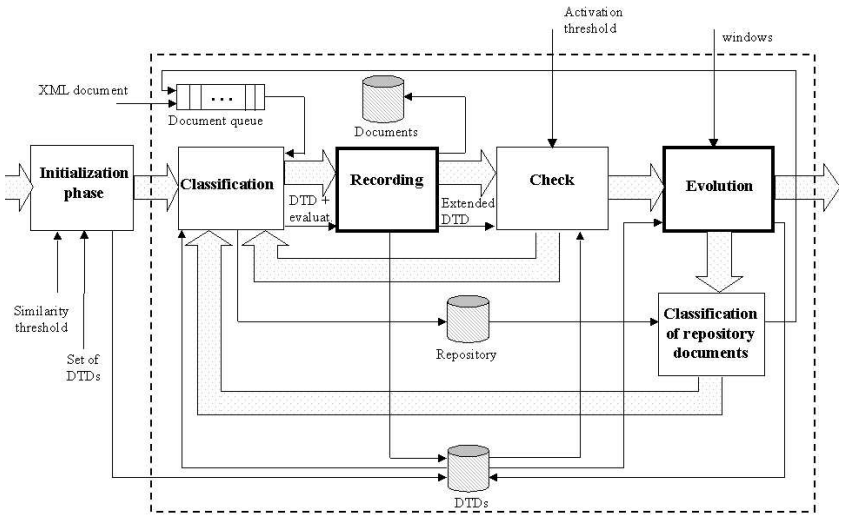

Figure 1. Overview of the evolution process

Once the classification phase is completed and thus the DTD of which the document is an instance has been selected, some structural information are extracted from the document. Specifically, information about valid and non valid parts of the documents are extracted, distinguishing among common, plus, and minus elements. *Common* elements are those elements appearing both in the document and in the DTD; *plus* elements are those appearing in the document but not in the DTD; *minus* elements are those appearing in the DTD but not in the document. In the *recording* phase these information are associated with the DTD in a data structure referred to as *extended DTD*. The use of such information avoids analyzing again the document in the subsequent phases. Moreover, these information are structural rather than content information, and they are aggregate over the whole set of analyzed documents, thus they do not require much storage space.

The classification and recording activities are iterated till the evolution phase is triggered. The evolution phase is activated after a certain number of documents have been classified. Specifically, we decided to trigger the evolution phase for a DTD when, among the documents classified in that DTD, the percentage of non valid documents, and the percentage of non valid elements in the documents, are above a fixed threshold. Thus, we compute the sum of the percentage of non valid elements for each document, normalized by the number of analyzed

documents. The evolution phase for a DTD T is thus triggered when

$$\frac{\sum_{D \in Doc_T} \frac{\#\{e|e \text{ element in } D \text{ non valid for } T\}}{\#\{e|e \text{ element in } D\}}}{\#Doc_T} > \tau$$

where Doc_T is the set of all documents classified in T, and τ is an a-priori fixed activation threshold. The *check* phase allows to determine whether the evolution phase should be activated.

The *evolution* phase of the evolution process is responsible for generating a new set of DTDs and should be able to work at different granularities, ranging from a very coarse granularity, regenerating the whole DTD, to a very fine granularity, regenerating the structure of a single element in the DTD. Exploiting the information collected in the recording phase, some association rules are extracted that represent relationships between presence/absence of subelements of an element. Based on such rules, on some heuristic policies we have identified, and on the evolution windows discussed in Section 4, the new DTD is generated.

Finally, after the evolution phase, the documents in the repository are classified again against the restructured set of DTDs in order to check whether the similarity is now above the threshold σ for some DTD in the source so that the document can be considered as instance of such DTD.

3 Recording Phase

The recording phase for a document d is performed in two steps. In the first step, among the DTDs in the source, the DTD most similar to document d is selected. The similarity degrees are computed and the corresponding evaluations stored in the auxiliary structures associated with the DTD. Then, in the second step, once the most similar DTD has been selected, the more relevant structural information are extracted from d and stored in auxiliary data structures associated with such DTD. Since the similarity degrees have been computed in the first step, the second step is very quick. Because of the operations performed in these two steps, the document will not be considered again during the evolution phase.

Documents and DTDs are represented as labeled trees. Formal definitions and properties of this representation can be found in [2]. The labels used for labeling a document are from a set $\mathcal{EN}$ of element tags, and a set $\mathcal{V}$ of `#PCDATA` values. By contrast, the labels used for labeling a DTD are from the set $\mathcal{EN} \cup \mathcal{ET} \cup \mathcal{OP}$, where $\mathcal{ET} = \{\#\texttt{PCDATA}, \texttt{ANY}\}$ is the set of basic types for elements, and $\mathcal{OP} = \{\texttt{AND}, \texttt{OR}, ?, *, +\}$ is a set of operators. The `AND` operator represents a sequence of elements, the `OR` operator represents an alternative of elements[2], the `?` operator represents an optional element, whereas the `*` and `+` operators represent repeatable elements (0 or more times and 1 or more times, respectively). Examples of an XML document and DTD and of their tree representations are shown in Figure 2. Given a subtree T of a document or a DTD, function $\alpha\beta$ returns the set of tags associated with the direct subelements of T. Note that whenever function $\alpha\beta$

[2] Note that at least one of the possible alternatives must be selected.

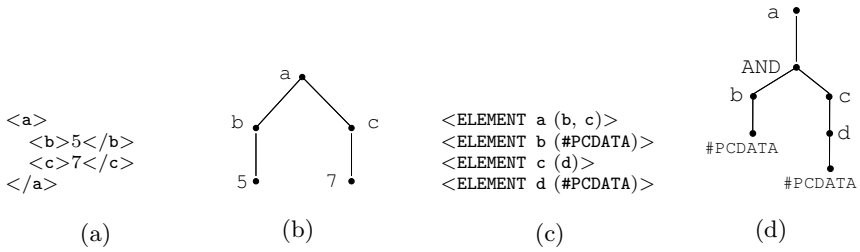

Figure 2. (a,c) Examples of XML document and DTD and (b,d) their tree representations

is applied to a node in a DTD, it returns the direct subelements independently from the operators used in the element type declaration. For instance, referring to the DTD in Figure 2 (c,d), $\alpha\beta(\texttt{a}) = \{\texttt{b}, \texttt{c}\}$.

In the remainder of the section we discuss through an example local and global similarity. Then, we present the notion of *extended DTD*, a DTD enriched with additional data structures for storing structural information of documents classified against it.

3.1 Global and Local Similarity

The global similarity of an element e_d of the document and an element e of the DTD is the evaluation of "how much" the subelements of e_d meet the constraints imposed by declaration of e in the DTD, taking into account the DTD declarations of subelements of e as well. Global similarity is the numeric counterpart of the (boolean) notion of validity. If a document is valid w.r.t. a DTD their global similarity is full (i.e. the global similarity degree is 1). By contrast, local similarity does not take into account the declarations of subelements of e in the DTD. That is, it just evaluates "how much" the direct subelements of e_d match the direct subelements specified in the declaration of e in the DTD.

Example 1. Consider the document and DTD of Figure 2. Element `a` in the document contains subelements `b` and `c`. The DTD declaration requires the same subelements. Therefore, their local similarity is full. By contrast, if we move down in the DTD declaration for element `c`, an element `d` is required, whereas element `c` in the document has a data content. Therefore, the global similarity of the document and the DTD is not full. ∘

Distinguishing between global and local similarity is important for DTD evolution. Because of local similarity, element `a` in the DTD declaration of Example 1 does not need to be modified. By contrast, element `c` could be modified if a huge number of documents, with the same structure as the one above, have been classified against such DTD.

Global and local similarity are computed by means of a recursive function. Such function is an enhanced version of the one presented in [2] for computing global similarity. Intuitively, the function visits at the same time the tree representations of a document and a DTD associating with each node an evaluation of plus, common and minus components between the two structures at that level of the tree representations. The evaluation is represented by means of triples (p, m, c) in which p, m, c are the evaluation of plus, minus, and common components, respectively. Starting from these triples, an evaluation function $\mathcal{E}$ [2] is then used for computing the global and local similarity, respectively.

3.2 Extended DTD

The DTD is extended with auxiliary data structures for containing the relevant information for the evolution phase. Such data structures are associated with each node of the DTD. The DTD with the auxiliary data structures is called *extended DTD*. In each element of the DTD we store information about the elements with the same tag found in that position in the hierarchical structure of the document. Such information are of two different kinds depending on the local similarity degree between the element of the document and the one of the DTD. If an element e_d in the document has a full local similarity w.r.t. the element e of the DTD, the counters of "number of valid instances" and "number of documents containing valid elements" are incremented. By contrast, if the local similarity is not full the counter "number of non valid instances" is incremented, the labels in $\alpha\beta(e_d)$ are added to the set of labels found for e, and the set $\alpha\beta(e_d)$ is added to the *set of sequences* for e. A *sequence* is a set of tags of direct subelements of e_d disregarding order and repetitions.[3] Moreover, for each label l belonging to $\alpha\beta(e_d)$, the following *structural information* are stored.

- The number of non valid instances of e containing l.
- The number of non valid instances of e containing l in which l is repeated more than once.
- If l does not appear in the declaration of e (i.e. $l \notin \alpha\beta(e)$), the structural information about subelements of l. This information is used for extracting from the instances with the same label a DTD declaration for l.
- If l is repeated more than once, the subsets of $\alpha\beta(e_d)$ containing l and other tags that are repeated the same number of times, which we refer to as *groups*. With each group S, included in the extended DTD, a counter r is associated. The counter is incremented each time S is found in the elements classified against e. Groups are used during the evolution phase for determining relationships among subelements of e.

Example 2. Let `T` be the DTD in Figure 3(a) and $\mathcal{D}_1$ and $\mathcal{D}_2$ be two sets of documents whose structures are reported in Figure 3(b). The label of root elements is `a` both for documents in $\mathcal{D}_1$ and $\mathcal{D}_2$ and all documents contain a sequence

[3] The set of sequences associated with e due to the classification of non valid elements will be used in the evolution phase to determine relationships among subelements.

of b and c elements. However, such sequence in documents in $\mathcal{D}_1$ is followed by a sequence of d elements, whereas in documents in $\mathcal{D}_2$ it is followed by an e element. Documents in both sets are not valid w.r.t. T. Figure 3(c) presents a sketch of the extended DTD. Element a is associated with the set $\{\texttt{b}, \texttt{c}, \texttt{d}, \texttt{e}\}$ of element tags found in the documents classified against T. Moreover, it contains information about the fact that $\{\texttt{b}, \texttt{c}\}$ forms a group since elements b and c are repeated the same number of times and the fact that element d is repeatable and optional (there are documents that do not contain it). ◦

By means of the information stored in these data structures the *invalidity ratio* ($\mathcal{I}$) for each element can be determined. If n is the number of elements in the documents classified against an element e of the DTD, and m is the number of elements for which the local similarity is less than 1 ($m \leq n$), then $\mathcal{I}(e) = \frac{m}{n}$. Relying on such value the evolution algorithm determines how to evolve the element. The possible modifications range from maintaining the current definition of the DTD declaration for element e to completely updating its structure to adhere to the new instances of the element found since last evolution.

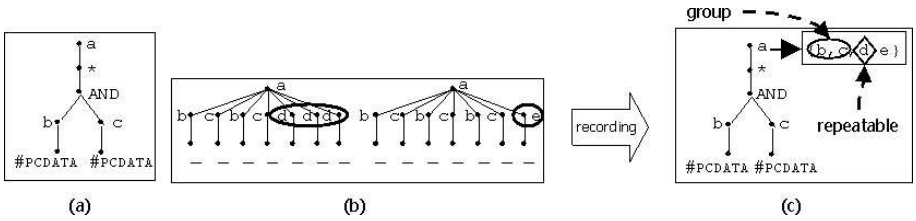

Figure 3: Recording structural information: (a) DTD, (b) actual structure of documents classified against the DTD, (c) extended DTD

4 Evolution Phase

In this section we first present the general ideas of the evolution phase. Then we get into details of one of the most relevant cases: the one in which the structure of an element of the DTD needs to be redetermined.

4.1 General Ideas

When the evolution phase is triggered the documents contained in the source can be partitioned in two subsets: $\mathcal{DOC}_{old}$, containing the documents that the current set of DTDs properly represents, and $\mathcal{DOC}_{cur}$ containing the documents classified in the source since the last evolution. Documents in $\mathcal{DOC}_{old}$ do not "explicitly" contribute to determine the structure of evolved DTDs, but they "implicitly" do, since their structure is already captured by the current set of DTDs.

Different requirements should be considered by the evolution algorithm in order to determine the structure of evolved DTDs. First, an appropriate relevance to documents in $\mathcal{DOC}_{old}$ w.r.t. documents in $\mathcal{DOC}_{cur}$ should be specified. Indeed, if documents in $\mathcal{DOC}_{old}$ are more relevant than the others, the new DTDs will be slightly different from the previous ones. By contrast, if documents in $\mathcal{DOC}_{cur}$ are more relevant than the others, the new DTDs could be much more different from the previous ones. Note that, in the latter case, this can lead to reduce the similarity of documents in $\mathcal{DOC}_{old}$ w.r.t. the new DTDs. This is however reasonable if we are not very confident about the structures of documents as specified by the current DTDs, for instance in the early stage of a source. Another requirement is to keep into account the most frequent deviations from the structure described by a DTD. For example, it could happen that most of the documents classified in a DTD contain a common new feature, that the DTD does not include. In such case, the new DTD will provide such feature because this can help applications working on such data. By contrast, if documents are really heterogeneous, adding/removing elements to/from the DTD will not help to better represent the documents in the source. Therefore, the DTD should not change. A last requirement is that relevant modifications can involve only few elements of the DTD. Therefore, DTD evolution should be performed only on such elements leaving the others unchanged. This "locality" of modifications is really important in order to reduce the impact on applications working on the source.

In order to address the above requirements the evolution algorithm works element by element by checking whether it is better to change the structure of the element or to leave it unchanged. The check is performed considering the invalidity ratio of elements introduced in Section 3. Such value is used for determining the behavior of the evolution algorithm depending on a threshold ψ ($0 \leq \psi \leq 0.5$). By means of this threshold it is possible to define three *windows*. If the value $\mathcal{I}(e)$ is in the range $[0, \psi]$, element e falls in the *old* window, whereas it falls in *new* window if $\mathcal{I}(e)$ is in the range $[1 - \psi, 1]$. In the other cases (i.e., $\mathcal{I}(e) \in (\psi, 1 - \psi)$) element e falls in the *misc* window. Changing the value of the ψ parameter we can give more or less relevance to non valid elements w.r.t. valid ones.

If an element falls in the *old* window then the amount of elements that do not conform to the DTD declaration is too low to trigger the modification of the DTD declaration. More relevance is thus given to documents in $\mathcal{DOC}_{old}$ and the evolution algorithm leaves the DTD declaration of this element unchanged. However, it is possible in this case to adapt the DTD structure to the valid elements classified against such element. For example, suppose to have a DTD declaration for element `a` that requires the presence of the subelement `b` repeated from 0 to many times (by means of the `*` operator). If all the elements `a` classified against this DTD contain at least an element `b`, it is possible to change the `*` operator in the `+` operator. This will give more relevance to the new documents w.r.t. the ones in $\mathcal{DOC}_{old}$. This operation has been called *restriction of operators*.

For each operator the possible restrictions have been identified and the respective conditions formalized.

If an element falls in the *new* window, then the amount of elements that do not conform to the DTD declaration is high. Thus the DTD structure must be changed. In this case the algorithm will determine a new structure for the element, possibly adding and removing subelements, and changing the operators binding them, in order to obtain a DTD declaration of such element that correctly describes the new instances of the DTD. The *new* window handling will be presented in the remainder of the section.

If an element does not fall in any of the previous windows, the evolution algorithm gives the same relevance to documents in $\mathcal{DOC}_{old}$ and in $\mathcal{DOC}_{cur}$. Therefore, documents in $\mathcal{DOC}_{cur}$ are used for obtaining the new structure of the DTD declaration of the element. Then, such definition is bound, by means of the `OR` operator, with the previous declaration of the DTD. A better formulation of the DTD is then obtained by means of "DTD re-writing rules" like the ones described in [2], that allows one to rewrite a DTD in a simpler, yet equivalent[4], one.

4.2 Determining the new Structure of an Element

Consider an element e of the DTD, let $E = \{e_{d_1} \ldots, e_{d_n}\}$ be the elements classified against e during the recording phase and $Label = \cup_{i=1}^{n} \alpha\beta(e_{d_i})$ be the labels of the subelements of the elements classified against e. If e falls in the *new* window then either elements $e_{d_1} \ldots, e_{d_n}$ contain some elements not appearing in the DTD, or these elements only contain subelements declared in the DTD but the operators used to bind subelements are not met. The new DTD declaration for element e is obtained by exploiting the structural information stored in the extended DTD during the recording phase by means of association rules and a set of heuristic policies, that are described in the remainder of the section.

Association Rules Having extracted from the elements classified against e the tags used in the subelements and the groups of tags that are found together, we want to determine how "frequently" the presence of some elements implies the presence of others. This result has been achieved by using *association rules* typical of data mining. An association rule is a rule of the form $X \rightarrow Y$, where X and Y are sets of items from a set I and $X \cap Y = \emptyset$. An association rule specifies that the presence of items X determine the presence of items Y. A *sequence* S is a set of items s.t. $S \subseteq I$. An association rule $X \rightarrow Y$ has *support* c in a set of sequences if the $c\%$ of sequences in the set contains $X \cup Y$. It has *confidence* c if the $c\%$ of sequences containing X, also contain Y.[5]

[4] That is, with the same set of valid documents.

[5] In our case "items" are element tags and the set of sequences is the one associated with element e.

Example 3. Consider the set of sequences $S = \{\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}, \{\mathtt{a}, \mathtt{b}\}, \{\mathtt{b}, \mathtt{c}, \mathtt{d}\}\}$ and the association rule $R = c \rightarrow a, b$. $Support(R) = Support(\{a, b, c\}) = \frac{1}{3}$. $Confidence(R) = \frac{Support(\{a,b,c\})}{Support(\{c\})} = \frac{1}{2}$. ◦

Association rules can also be used for determining relationships like "the presence of some elements implies the absence of others" by considering the concept of *absent element*. Given an element e_{d_i} $(1 \leq i \leq n)$ the absent elements for e_{d_i} are $Label \setminus \alpha\beta(e_{d_i})$. If $\mathtt{b}$ is an absent element for e_{d_i}, the notation $\bar{\mathtt{b}}$ is used for representing its absence.

Example 4. Consider the sequences of Example 3 and suppose that each sequence corresponds to the subelement tags found in a document element. Then, the set *Label* is $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$. The only absent element for the sequence $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ is $\mathtt{d}$, whereas $\mathtt{c}$ and $\mathtt{d}$ are absent for the sequence $\{\mathtt{a}, \mathtt{b}\}$. The sequences in S can then be represented as $S = \{\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \bar{\mathtt{d}}\}, \{\mathtt{a}, \mathtt{b}, \bar{\mathtt{c}}, \bar{\mathtt{d}}\}, \{\bar{\mathtt{a}}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}\}$. ◦

Note that the introduction of absent elements in the sequences makes it possible to determine from the set of sequences associated with e also association rules of the form "if element $\mathtt{b}$ is absent then element $\mathtt{c}$ is present". Such kind of rules is useful for determining subelements of element e that never appear together (i.e., that are bound through the `OR` operator).

Starting from a threshold μ representing the minimum support for a sequence of element tags, and an element e that falls in the new window, the steps followed by the evolution algorithm are the following.

1. For each sequence associated with e, the algorithm updates the sequences of elements to include the absent elements, according to the notion of absent element introduced above, as we have done in Example 4.
2. Those sequences are then used for computing the most frequent sequences. The most frequent sequences are those having support greater than μ. The other sequences are discarded since they are not representative enough.
3. Association rules are extracted from the most frequent sequences. These rules represent the relationships among subelements of the elements $e_{d_1} \ldots, e_{d_n}$. The relationships represented by those rules are of two types: "the presence of these elements implies the presence of these elements"; "the absence of these elements implies the presence of these elements".
4. Association rules with the maximal confidence (i.e., 1) are extracted.

Let $\mathcal{R}ules$ be the set of rules computed according to the previous steps. Those rules represent the most common relationships found in the elements $e_{d_1} \ldots, e_{d_n}$ and are used by the heuristic policies we have devised to determine the DTD declaration of element e.

Heuristic Policies Different policies have been developed in order to determine the operators that bind together the subelements of an element e that falls in the *new* window. Such policies are based on the association rules previously

discussed, on the information about the repetition of each single subelement, and on the groups to which the subelements belong. Our approach is based on two basic principles. Let $X = \{x\},\ Y = \{y\},\ X, Y \subseteq \mathcal{EN}$ be two subsets of the tag element set.

P1 (*Extraction of the* `AND`*-binding between two elements*). If $\{x \rightarrow y, y \rightarrow x\} \subseteq \mathcal{R}ules$, then in most cases subelements x and y are present together. Therefore, these elements are bound by the `AND` operator.

P2 (*Extraction of the* `OR`*-binding between two elements*). If $\{x \rightarrow \bar{y}, \bar{y} \rightarrow x\} \subseteq \mathcal{R}ules$, then in most cases when subelement x is present subelement y is absent and viceversa. Therefore, these elements are alternative and are thus bound by an `OR` operator.

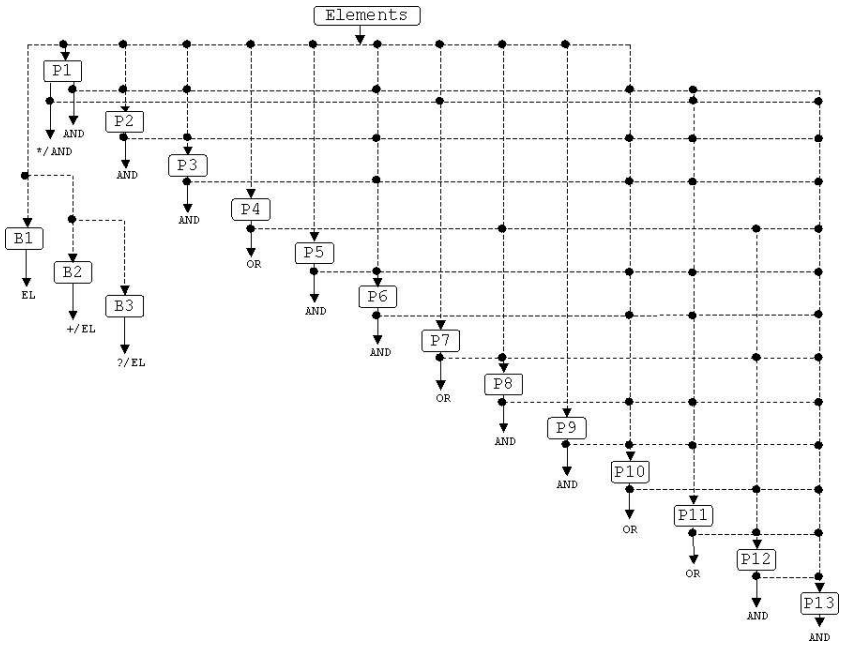

Figure 4: Policies and their relationships

These principles do not take into account that subelements can be repeated more than once and not always the same number of times. However, such principles can be generalized to sets of elements containing more than one element and that can be repeated more than once.

The algorithm for extracting the new structure of an element takes as input a set C of elements, a set $\mathcal{R}ules$ of association rules, and information about the repetitions of each element in C and of subset of elements in C. The set C is

generated by considering the elements associated with the tags in *Label*. Relying on our representation of elements, C is thus a set of trees.

Starting from these input parameters the algorithm applies 13 policies we have identified for determining the new structure of the element e. Each policy is composed of two parts: the condition and the re-writing parts. The first part of the policy specifies a condition on a subset C_1 of C and the association rules that should belong to $\mathcal{R}ules$ for the policy to be applicable. If the condition is verified, in the re-writing part of the policy, the set C_1 is removed from C and a new tree is added to C. Such tree represents the new structure for the elements contained in C_1. Each policy is applied exhaustively, that is, the algorithm finds all the subsets of C for which the policy condition is verified and, when the next policy is applied, it is not possible to consider again the previous one. Policies are thus applied in turn till set C becomes a singleton. When C is a singleton the process terminates and the obtained element represents the new binding of the subelements of the element to be restructured.

In addition to the 13 policies there are three policies that handle basic cases. The basic cases arise when the starting set C is already a singleton (that is, $C = \{T\}$). Then, no rule can be applied and the actual tree produced depends on the optionality and repeatability of T. Specifically, if T is neither optional nor repeatable it is left unchanged. Otherwise, it is replaced by $T' = (v', [T])$, where v' is a new vertex whose label is `?`, `+`, or `+`, depending on whether T is optional, repeatable, or optional and repeatable.

The 13 policies are defined in Appendix A. Figure 4 graphically depicts their interactions. The input of a policy can be an element, i.e., a tree whose root label belongs to $\mathcal{EN}$, or an operator, i.e. a tree whose root label belongs to $\mathcal{OP}$, and produces a tree labeled by an operator. Policies 1, 4, 5, 9 take as input only subsets of C labeled by an element tag, whereas policies 11, 12, 13 take as input only trees labeled by operators. The other policies consider both trees labeled by element tags and operators. In the figure we point out the type of element considered and the root label of the generated tree. A bullet in the grid means that the output of a rule (the horizontal line) can be the input of another rule (the one reached by the vertical line).

Example 5. Consider the document, the DTD and the extended DTD of Example 2. Suppose that element `a` falls into the new window. The set $C = \{\mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\}$ is computed. The set $\mathcal{R}ules$ contains the rules $\{\mathtt{b} \rightarrow \mathtt{c}, \mathtt{c} \rightarrow \mathtt{b}, \mathtt{d} \rightarrow \bar{\mathtt{e}}, \bar{\mathtt{e}} \rightarrow \mathtt{d}\}$. Such rules have been identified by applying the steps we have outlined in the above discussion on association rules. The set $\mathcal{R}ules$ also contains other rules that, however, are not relevant for this example.

Consider the group $\{\mathtt{b}, \mathtt{c}\}$. The elements of such group are always repeated the same number of times. Policy 1 can be applied on such group (the condition part is reported in Figure 5). Therefore elements `b` and `c` can be removed from C and the tree (1) in Figure 5 can be added to the DTD.

On the new set C neither policy 1 can be applied, nor policies 2 and 3. By contrast, policy 4 (whose condition part is reported in Figure 5) can be applied on $\{\mathtt{d}, \mathtt{e}\}$. Indeed, when element `d` is present, element `e` is absent and viceversa

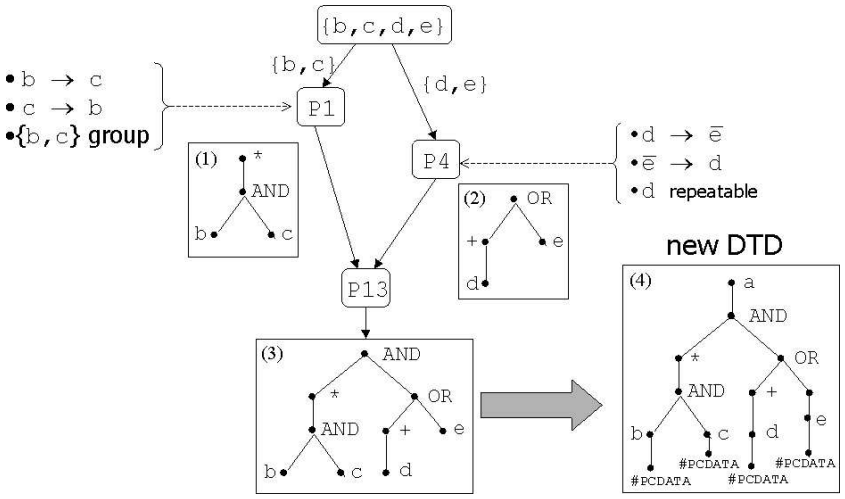

Figure 5: Application of the evolution algorithm

(the two elements are alternative). Therefore elements `e` and `d` can be removed from C and the tree (2) in Figure 5 can be added to the DTD.

C is now only composed of trees whose root labels are operators (in one case the `*` operator, in the other the `OR` operator). Thus, only policies 11, 12, 13 can be applied on them. Conditions of policies 11 and 12 do not hold, rather the condition of policy 13 holds; thus such policy is applied and the two trees are replaced in C by a new tree (tree (3) in Figure 5) whose root label is the `AND` operator and whose children are the previous two trees. Set C becomes a singleton and, thus, the new DTD declaration for element `a` is determined.

Note that, since element `d` and `e` are plus elements, no declaration is provided for them. However, by recursively applying the evolution algorithm for each of them, considering as DTD an empty DTD, their actual structure can be extracted as shown in Figure 5 (tree (4)). ◦

5 Related Work

A related problem that has been investigated for semi-structured data [7, 9] and XML [3, 6] is that of structure extraction, that is, the discover of schema information (in the form of a data guide or of a DTD, respectively) from a set of documents.

In [7] a technique is proposed for extracting implicit schema information, in the form of an *approximate type assignment*, from semi-structured data. The type assignment is approximate since data are not required to exactly conform to the type assigned to them. The approach is based on the idea of defining Datalog programs to type a set of semi-structured objects. Starting from a perfect typing (that is, a type for any distinct object), the number of types is then reduced by

collapsing the types of objects with a slightly different structure. To determine which types to collapse some measures are used based on the number of type instances and on Manhattan distance between types.

The Road Map approach [9] is based on the idea of extracting, from a set of semi-structured documents on the same topic, their common structure, thus discovering the similarities among the documents. The approach can be parameterized according to user needs. For instance, the user can set the minimal frequency of expressions due to common structures in documents. The approach is based on a *weaker than* order among document structures represented as trees, and on the association of a numeric support with structures. The approach allows one to extract a structure that describes the maximum number of documents.

XTRACT [3] is based on an algorithm for extracting, given a set of documents, a DTD for these documents being at the same time concise (that is, small) and precise (that is, capturing all the document structures). The algorithm is based on three steps:

- heuristic algorithms are used for finding patterns in input documents and replacing them with appropriate regular expressions to produce more general candidate DTDs;
- common subexpressions are factored out from the generalized candidate DTDs obtained from the previous step, in order to make them more concise;
- among the candidate DTDs the one providing the most concise representation of input documents in terms of the DTD is chosen.

The approach described in [6] has been developed for re-engineering the structures of Web documents. The starting point is the extraction of each document structure, in the form of a document tree, and the grouping of documents with similar structure in clusters. This grouping is performed through pattern matching algorithms applied to document trees. Then, a general structure is extracted for each document cluster, through a data structure referred to as *spanning graph*, which is a DAG containing information on the structures of the original document trees, and it is incrementally built while document trees are analyzed. From this structure a DTD is finally obtained, through the use of a set of heuristic rules, that is representative of all the documents in the group.

Despite of their differences, all those proposals have the common goal of developing approaches to extract schema information from data, that is, to extract structures from raw data. They have some similarities with our approach, since the goal is common, i.e., to obtain, from examined data, an adequate representation of their structure. This structure should be representative of as many documents as possible, but, at the same time, should be as accurate as possible, as in [9]. Another common aspect is that all the techniques, like ours, rely on the maintainance of internal information extracted from documents. Specifically, the data structure we employ, i.e., the *extended DTD*, is similar to the *Spanning Graph* proposed in [6]. Another common point with [6] is the preliminary phase of document clustering based on similarity, though this clustering is achieved in different ways. Finally, heuristic policies are employed in XTRACT to generalize structural expressions.

However, all those approaches substantially differ from ours in that they do not address how the structure extraction mechanisms can exploit some a-priori knowledge on the data schema. We remark that this knowledge, that we assume in our approach, often occurs in practice, for instance when integrating semi-structured data, discovered on the Web, with data having a known structure. Moreover, those approaches work by examining a set of documents at a time, and extracting the schema from these documents. It is not specified whether and how the insertion of new data, once the schema is set, can result in schema modifications. Our approach, by contrast, is incremental. Structural information are extracted and recorded for each document when the document is classified, and the evolution process, triggered by an appropriate event, can be applied several times, to adapt the current schema to the newly classified documents. Thus, two separate steps (recording and evolution steps) characterize our approach, whereas these two steps are merged in other approaches. Another difference is related to the granularity of the process, which is finer, i.e. a single element in our approach, whereas it is the whole document in other approaches. Thus, local reorganizations are possible in our approach. Moreover, an important improvement of our approach with respect to [6] is that our approach allows one to generate DTDs containing the `OR` operator, that is not generated in [6]. A final difference with XTRACT, is that we do not address the problem of generating DTDs that are as concise as possible. However, the use of "re-writing rules" as proposed in [2] makes it possible the definition of simpler DTDs.

6 Conclusions

In this paper we have proposed an approach to adapt a set of DTDs to a dynamic set of XML documents. We are currently experimentally evaluating the proposed approach, with the main goal of assessing the quality of the obtained DTDs. Since a DTD can be considered as a kind of XML schema, we are currently extending the approach to the evolution of XML schemas. A related problem that is currently under investigation is how to adapt documents, already stored in the source, to the new structure prescribed by the evolved set of DTDs.

There are a number of directions along which the proposed approach can be extended. The first one concerns the possibility of evolving tag names as well as their structure by relying on the use of a Thesaurus [5]. The Thesaurus allows one to evaluate structural similarity shifting from tag equality to tag similarity, as sketched in [2]. A second direction is related to the development of an evolution trigger language, by using which applications can specify and automatically activate DTD evolution.

References

1. S. Abiteboul. Querying Semi-Structured Data. In F. Afrati and P. Kolaitis, editors, *Database Theory - ICDT'97*, pages 1–18, 1997.

2. E. Bertino, G. Guerrini, and M. Mesiti. Measuring the Structural Similary among XML Documents and DTDs. Technical Report DISI-TR-02-02, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, December 2001. Available at `http://www.disi.unige.it/person/MesitiM`.
3. M. N. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: A System for Extracting Document Type Descriptors from XML Documents. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 165–176, 2000.
4. J. H. Han, and M. Kamber. Data Mining, Concepts and Techniques. Morgan Kaufmann. 2001.
5. A. Miller. WordNet: A Lexical Database for English. *Communications of the ACM*, 38(11):39–41, November 1995.
6. C. Moh, E. Lim, and W. Ng. Re-engineering Structures from Web Documents. In *Proc. of the Fifth ACM Int'l Conf. on Digital Libraries*, pages 67–76, 2000.
7. S. Nestorov, S. Abiteboul, and R. Motwani. Extracting Schema from Semistructured Data. In L. M. Haas and A. Tiwary, editors, *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 295–306, 1998.
8. W3C. Extensible Markup Language (XML) 1.0, 1998.
9. K. Wang and H. Liu. Discovering Typical Structures of Documents: a Road Map Approach. In *Proc of the Twentyfirst Annual Int'l ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 146–154, 1998.

A Heuristic Policies

In the definition of the heuristic policies we will use the following notations.

- Documents and DTDs are referred to as (T, φ) pairs, where T is a tree and φ is a vertex labeling function. Specifically, T is either a vertex v or a vertex v with a list $[T_1, \ldots, T_n]$ of subtrees.
- Given a tree, function *label* return its root label.
- Given a tree representing an element, function $\mathcal{R}$ returns the number of repetitions of such element. This information is maintained in the extended DTD.
- $C = \{T_1, \ldots, T_n\}$ is a set of trees. Initially, this set contains only trees labeled by an element tag.
- $L_n = \{x_1, \ldots, x_n\}$ is the set of labels associated with trees whose root labels are element tags. Initially, there is a 1:1 correspondence with the trees in C. A label x_k is always associated with the tree T_k.
- Given L_k subset of L_n, the relation $L_k \subseteq_M L_n$ means that the set L_k is maximal subset of L_n on which a property required by a policy holds. That is, if the set $L_k \cup \{\bar{x}\}$ with $\bar{x} \in L_n \setminus L_k$ is considered, then the property does not hold anymore.

Policy 1 *(*Extraction of an `AND`-binding*). If $L_k \subseteq_M L_n$ exists s.t. the following condition holds*

$$\forall i, 1 \leq i \leq k, \{x_i \rightarrow x_1 \ldots x_{i-1} x_{i+1} \ldots x_k, \quad x_1 \ldots x_{i-1} x_{i+1} \ldots x_k \rightarrow x_i\} \subseteq \mathcal{R}ules$$

then the policy is applicable and the generated tree depends on the number of repetitions of the trees in L_k, and on the most frequent groups among those collected during the recording phase.

- *If* $\forall i, j,\ 1 \leq i, j \leq k,\ \mathcal{R}(T_i) = \mathcal{R}(T_j) = 1$, *then the tree* $(v, [T_1, \ldots, T_k])$ *is generated, with* v *a new vertex, and* $\varphi(v) =$ `AND`.
- *If* $\forall i, j,\ 1 \leq i, j \leq k,\ \mathcal{R}(T_i) = \mathcal{R}(T_j) = m$, $m > 1$, *and a group* $(\{x_1, \ldots, x_k\}, m)$ *is present, then the tree* $(v_1, [(v_2, [T_1, \ldots, T_k])])$ *is generated, with* v_1, v_2 *new vertices, and* $\varphi(v_1) =$ `*`, $\varphi(v_2) =$ `AND`.
- *If the previous conditions do not hold (the trees are repeated, but not the same number of times), the groups in a set* $\mathcal{G}$ *s.t. for each* $G \in \mathcal{G}$, $G \subseteq L_k$, *and for each* $G', G'' \in \mathcal{G}$ *s.t.* $G' \neq G'', G' \cap G'' = \emptyset$, *are considered. Let* $\bar{L} = L_k \setminus \cup_{G' \in \mathcal{G}} G'$, *be the set of elements which are repeated only once or are repeated more than once but separately. Let* S_T *be a set, initially empty, of trees. The following steps are performed.*
  - *For each* $T' \in \bar{L}$, *if* $\mathcal{R}(T') > 1$, *then the tree* $(v, [T'])$, *with* v *a new vertex and* $\varphi(v) = +$, *is added to* S_T, *otherwise* T' *itself is added to* S_T.
  - *For each* $G' \in \mathcal{G}$, $G' = \{x_1, .., x_h\},\ 1 \leq h \leq k$, $K' = \{T_1, .., T_h\}$, *where* T_i *is the tree corresponding to* x_i, $1 \leq i \leq h$, *the tree* $T' = (v_1, [(v_2, [T_1, \ldots, T_h])])$ *is generated, with* v_1, v_2 *new vertices,* $\varphi(v_1) =$ `+`, $\varphi(v_2) =$ `AND`. $S_T = S_T \cup \{T'\}$.

  The tree (v_1, S_T) *is then generated, with* v_1 *a new vertex and* $\varphi(v_1) =$ `AND`.

The generated tree is added to C *and the trees* $T_1, \ldots, T_k$ *are removed from* C. □

Policy 2 *(*Extraction of an `AND`-binding between a tree labeled by an element tag and a `*` labeled tree*). Let* $A = \{T | T \in C, label(T) = *\}$. *For each* $T \in A$, *if* $\exists x \in L_n$ *s.t.* $\alpha\beta(T) \rightarrow x \in \mathcal{R}ules$, *then the policy is applicable and the tree* $(v, [T, T_x])$ *is generated, with* v *a new vertex and* $\varphi(v) =$ `AND`. *The generated tree is added to* C *and the trees* T *and* T_x *are removed from* C. □

Policy 3 *(*Extraction of an `AND`-binding between a tree labeled by an element tag and an `AND` labeled tree*). Let* $A = \{T | T \in C, label(T) =$ `AND`$\}$. *For each* $T \in A$, $T = (v, [T'_1, \ldots, T'_h])$, *if* $\exists\ T' \in [T'_1, \ldots, T'_h]$ *s.t.* $label(T') \in \mathcal{EN}$, *and* $\exists\ L_k \subseteq L_n$, *s.t.* $\forall x_i \in L_k$:

- $x_i \rightarrow label(T') \in \mathcal{R}ules$,
- $x_i \rightarrow \overline{x}_1 \ldots \overline{x}_{i-1} \overline{x}_{i+1} \ldots \overline{x}_k \notin \mathcal{R}ules$,
- $x_i = label(T_i)$,

then the policy is applicable and the tree $(v, [T'_1, \ldots, T'_h, T'_1, \ldots, T'_k])$ *is generated. Each* T'_j, $1 \leq j \leq k$, *is of the form* $(v, [T_j])$. *If* $\mathcal{R}(T_j) = 1$ *then* $\varphi(v) =$ `?`, *otherwise* $\varphi(v) =$ `*`. *The generated tree is added to* C *and the trees* $T_1, \ldots, T_k, T$ *are removed from* C. □

Policy 4 *(*Extraction of an `OR`*-binding). If* $L_k \subseteq_M L_n$ *exists s.t. the following condition holds*

$$\forall i, 1 \leq i \leq k,\quad \{x_i \rightarrow \overline{x}_1 \ldots \overline{x}_{i-1} \overline{x}_{i+1} \ldots \overline{x}_k,\quad \overline{x}_1 \ldots \overline{x}_{i-1} \overline{x}_{i+1} \ldots \overline{x}_k \rightarrow x_i\} \subseteq \mathcal{R}ules$$

then the policy is applicable and the tree $(v, [T'_1, \ldots T'_k])$ *is generated, with* v *a new vertex and* $\varphi(v) =$ `OR`. *If* $\mathcal{R}(T_j) > 1$, $1 \leq j \leq k$, T'_j *is of the form* $(v, [T_j])$ *with* v *a new vertex and* $\varphi(v) =$ `+`. *Otherwise,* T'_i *is* T_i *itself. The generated tree is added to* C *and the trees* $T_1, \ldots, T_k$ *are removed from* C. □

Policy 5 *(*Extraction of a simple `AND`/`OR`-binding*). If* $L_k \subseteq_M L_n$ *exists s.t. the following condition holds*

$$\forall i, 1 \leq i \leq k-1,\ \{x_i \rightarrow \overline{x}_1 \ldots \overline{x}_{i-1} \overline{x}_{i+1} \ldots \overline{x}_{k-1},\quad x_i \rightarrow x_k\} \subseteq \mathcal{R}ules$$

then the policy is applicable and the tree generated depends on the number of repetitions of the tree T_k. If $\mathcal{R}(T_k) = 1$, the tree $(v_1, [(v_2, [T_1, \ldots, T_{k-1}]), T_k])$ is generated, with v_1, v_2 new vertices and $\varphi(v_1) =$ `AND`, $\varphi(v_2) =$ `OR`. Otherwise ($\mathcal{R}(T_k) > 1$), the tree $(v_1, [(v_2, [T_1, \ldots, T_{k-1}]), (v_3, [T_k])])$ is generated, with v_1, v_2, v_3 new vertices and $\varphi(v_1) =$ `AND`, $\varphi(v_2) =$ `OR` and $\varphi(v_3) =$ `+`. The generated tree is added to C and the trees $T_1, \ldots, T_k$ are removed from C. □

Policy 6 (Extraction of a complex `AND`/`OR`-binding). *Let $A = \{T | T \in C, label(T) =$ `AND`$\}$. For each $T \in A$, $T = (v, [T'_1, \ldots, T'_h])$, if $L_k \subseteq_M L_n$, exists s.t. the following condition holds: $\forall i$, $1 \leq i \leq k$,*

- $x_i \rightarrow \overline{x}_1 \ldots \overline{x}_{i-1} \overline{x}_{i+1} \ldots \overline{x}_k \in \mathcal{R}ules$,
- $\forall j, 1 \leq j \leq h$, *if* $label(T_j) \in \mathcal{EN}$ *then* $x_i \rightarrow label(T_j) \in \mathcal{R}ules$,

then the policy is applicable and the tree $(v_1, [T'_1, \ldots, T'_h, (v_2, [T_1, \ldots, T_k])])$ is generated, with v_1, v_2 new vertices and $\varphi(v_1) =$ `AND`, $\varphi(v_2) =$ `OR`. The generated tree is added to C and the trees $T_1, \ldots, T_k$ are removed from C. □

Policy 7 (Extraction of a weak `OR`-binding). *Let $A = \{T | T \in C, label(T) = *\}$. For each $T \in A$ s.t. $T = (v, [T'])$, $label(T') =$ `AND` and $\{y_1, \ldots, y_h\} = \alpha\beta(T')$, if $L_k \subseteq_M L_n$ exists s.t. the following condition holds: $\forall i$, $1 \leq i \leq k$,*

- $x_i \rightarrow \overline{y}_1 \ldots \overline{y}_h \in \mathcal{R}ules$,
- $x_i \rightarrow \overline{x}_1 \ldots \overline{x}_{i-1} \overline{x}_{i+1} \ldots \overline{x}_k \notin \mathcal{R}ules$,
- $\exists y_j \in \{y_1, \ldots, y_h\}$ *s.t.* $y_j \overline{x}_i \rightarrow y_1 \ldots y_{j-1} y_{j+1} \ldots y_h \in \mathcal{R}ules$,

then the policy is applicable and the tree $(v_1, [(v_2, [T']), (v'_1, [T_1]), \ldots, (v'_k, [T_k])])$ is generated, with v_1, v_2, $v'_1, \ldots, v'_k$ new vertices, $\varphi(v_1) =$ `OR`, $\varphi(v_2) =$ `+`. If $\mathcal{R}(T_i) = 1, 1 \leq i \leq k$, then $\varphi(v'_i) =$ `?`, $\varphi(v'_i) =$ `*` otherwise. The generated tree is added to C and the trees $T_1, \ldots, T_k, T$ are removed from C. □

Policy 8 (Extraction of an `AND`/`OR`-binding between a tree labeled by an element tag and a `OR` labeled tree). *Let $A = \{T | T \in C, label(T) =$ `OR`$\}$. For each $T \in A$, if $x \in L_n$ exists s.t.*

$$\forall y \in \alpha\beta(T), \quad y \rightarrow x \in \mathcal{R}ules$$

then the policy is applicable and the tree $(v, [T, T_x])$ is generated, with v a new vertex and $\varphi(v) =$ `AND`. The generated tree is added to C and the trees T, T_k are removed from C. □

Policy 9 (Extraction of an `AND`-binding by means of a dependency). *Let $\{x_a, x_b\} \subseteq L_n$. If $x_b \rightarrow x_a \in \mathcal{R}ules$ then the policy is applicable and the tree $(v_1, [T_a, (v_2, [T_b])])$ is generated, with v_1, v_2 new vertices, $\varphi(v_1) =$ `AND`. If $\mathcal{R}(T_b) = 1$, then $\varphi(v_2) =$ `?`, $\varphi(v_2) =$ `*` otherwise. The generated tree is added to C and the trees T_a, T_b are removed from C.* □

Policy 10 (Extraction of an `OR`-binding between `AND` labeled trees and trees labeled by an element tag). *Let $A = \{T | T \in C, label(T) =$ `AND`$\}$. If $L_k \subseteq_M L_n$, exists s.t. the following condition holds: $\exists B = \{T'_1, \ldots, T'_h\} \subseteq A$ s.t. $\forall T' \in B$*

- $\forall y \in \alpha\beta(T')$, $y \rightarrow \overline{y}'_1 \ldots \overline{y}'_n \overline{x}_1 \ldots \overline{x}_k \in \mathcal{R}ules$, *where* $\{y'_1, \ldots, y'_n\} = \cup_{T \in (B \setminus \{T'\})} \alpha\beta(T)$,
- $\forall i, 1 \leq i \leq k$, $x_i \rightarrow \overline{x}_1 \ldots \overline{x}_{i-1} \overline{x}_{i+1} \ldots \overline{x}_k \overline{y}'_1 \ldots \overline{y}'_m \in \mathcal{R}ules$, *where* $\{y'_1, \ldots, y'_m\} = \cup_{T \in B} \alpha\beta(T)$,

then the policy is applicable and the tree $(v, [T'_1, \ldots, T'_h, T_1, \ldots, T_k])$ *is generated, with* v *a new vertex and* $\varphi(v) = \texttt{OR}$*. The generated tree is added to* C *and the trees* $T'_1, \ldots, T'_h$*,* $T_1, \ldots, T_k$ *are removed from* C*.* □

Policy 11 *(*Extraction of an `OR`-binding between `AND` labeled trees*). Let* $\{T_1, \ldots, T_h\} \subseteq C$ *s.t.* $\forall i, 1 \leq i \leq h, label(T_i) = \texttt{AND}$*. Let* $\{T_{h+1}, \ldots, T_n\} \subseteq C$ *s.t.* $\forall i, h+1 \leq i \leq n, label(T_i) = *$*. If* $n > 1$ *then the policy is applicable and the tree* $(v, [T_1, \ldots, T_n])$ *is generated, with* v *a new vertex and* $\varphi(v) = \texttt{OR}$*. The generated tree is added to* C *and the trees* $T_1, \ldots, T_n$ *are removed from* C*.* □

Policy 12 *(*Extraction of an `AND`-binding between `OR` labeled trees*). Let* $\{T_1, \ldots, T_n\} \subseteq C$ *s.t.* $\forall i, 1 \leq i \leq n, label(T_i) = \texttt{OR}$*. If* $n > 1$ *then the policy is applicable and the tree* $(v, [T_1, \ldots, T_n])$ *is generated, with* v *a new vertex and* $\varphi(v) = \texttt{AND}$*. The generated tree is added to* C *and the trees* $T_1, \ldots, T_n$ *are removed from* C*.* □

Policy 13 *(*Extraction of the final `AND`-binding*). Let* $\{T_1, \ldots, T_h\} \subseteq C$ *s.t.* $\forall i, 1 \leq i \leq h, label(T_i) = \texttt{OR}$*. Let* $\{T_{h+1}, \ldots, T_l\} \subseteq C$ *s.t.* $\forall i, h+1 \leq i \leq l, label(T_i) = *$*. let* $\{T_{l+1}, \ldots, T_n\} \subseteq C$ *s.t.* $\forall i, l+1 \leq i \leq n, label(T_i) = \texttt{AND}$*. If* $n > 1$ *the policy is applicable and the tree* $(v, [T_1, \ldots, T_n])$ *is generated, with* v *a new vertex and* $\varphi(v) = \texttt{AND}$*. The generated tree is added to* C *and the trees* $T_1, \ldots, T_n$ *are removed from* C*.* □