# XTRACT: A System for Extracting Document Type Descriptors from XML Documents

**Minos Garofalakis**
Bell Laboratories
*minos@bell-labs.com*

**Aristides Gionis***
Stanford University
*gionis@cs.stanford.edu*

**Rajeev Rastogi**
Bell Laboratories
*rastogi@bell-labs.com*

**S. Seshadri**
Bell Laboratories
*seshadri@bell-labs.com*

**Kyuseok Shim**†
KAIST and AITrc
*shim@cs.kaist.ac.kr*

## Abstract

XML is rapidly emerging as the new standard for data representation and exchange on the Web. An XML document can be accompanied by a *Document Type Descriptor* (DTD) which plays the role of a schema for an XML data collection. DTDs contain valuable information on the structure of documents and thus have a crucial role in the efficient storage of XML data, as well as the effective formulation and optimization of XML queries. In this paper, we propose XTRACT, a novel system for inferring a DTD schema for a database of XML documents. Since the DTD syntax incorporates the full expressive power of *regular expressions*, naive approaches typically fail to produce concise and intuitive DTDs. Instead, the XTRACT inference algorithms employ a sequence of sophisticated steps that involve: (1) finding patterns in the input sequences and replacing them with regular expressions to generate "general" candidate DTDs, (2) factoring candidate DTDs using adaptations of algorithms from the logic optimization literature, and (3) applying the Minimum Description Length (MDL) principle to find the best DTD among the candidates. The results of our experiments with real-life and synthetic DTDs demonstrate the effectiveness of XTRACT's approach in inferring concise and semantically meaningful DTD schemas for XML databases.

## 1 Introduction

**Motivation and Background.** The genesis of the Extensible Markup Language (XML) was based on the thesis that structured documents can be freely exchanged and manipulated, if published in a standard, open format. Indeed, as a corroboration of the thesis, XML today promises to enable a suite of next-generation Web applications ranging from intelligent web searching to electronic commerce.

In many respects, XML data is an instance of *semi-structured data* [1]. XML documents comprise hierarchically nested collections of *elements*, where each element can be either atomic (i.e., raw character data) or composite (i.e., a sequence of nested subelements). Further, *tags* stored with elements in an XML document describe the semantics of the data rather than simply specifying how the element is to be displayed (as in HTML). Thus, XML data, like semistructured data, is hierarchically structured and self-describing.

A characteristic, however, that distinguishes XML from semistructured data models is the notion of a *Document Type Descriptor* (DTD) that may optionally accompany an XML document. A document's DTD serves the role of a schema specifying the internal structure of the document. Essentially , a DTD specifies for every element, the *regular expression* pattern that subelement sequences of the element need to conform to. DTDs are critical to realizing the promise of XML as the data representation format that enables free interchange of electronic data (EDI) and integration of related news, products, and services information from disparate data sources. This is because, in the absence of DTDs, tagged documents have little meaning. However, once the major software vendors and corporations agree on domain-specific standards for DTD formats, it would become possible for inter-operating applications to extract, interpret, and analyze the contents of a document based on the DTD that it conforms to.

In addition to enabling the free exchange of electronic documents through industry-wide standards, DTDs also provide the basic mechanism for defining the structure of the underlying XML data. As a consequence, DTDs play a crucial role in the efficient storage of XML data as well as the formulation, optimization, and processing of queries over a collection of XML documents. For instance, in [23], DTD information is exploited to generate effective relational schemas, which are subsequently employed to efficiently store and query entire XML documents in a relational database. In [7], frequently occurring portions of XML documents are stored in a relational system, while the remainder is stored in an overflow graph; once again, the DTD is exploited to simplify overflow mappings. Similarly, DTDs can be used to devise efficient plans for queries and thus speed up query evaluation in XML databases by restricting the search to only relevant portions of the data (see, for example, [8, 13]). The basic idea is to use the knowledge of the structure of the data captured by the DTD to prune elements that cannot potentially satisfy the path expression in the query. Finally, by shedding light on how the underlying data is structured, DTDs aid users in forming

*Work done while visiting Bell Laboratories.
†Work done while visiting Bell Laboratories.

meaningful queries over the XML database.

Despite their importance, however, DTDs are *not mandatory* and an XML document may not always have an accompanying DTD. In fact, several recent papers (e.g., [12, 25]) claim that it is frequently possible that only specific portions of XML databases will have associated DTDs, while the overall database is still "schema-less". This may be the case, for instance, when large volumes of XML documents are automatically generated from data stored in relational databases, flat files (e.g., HTML pages, bibliography files), or other semistructured data repositories. Since very little data is in XML format today, it is very likely that, at least initially, the majority of XML documents will be automatically generated from pre-existing data sources by a new generation of software tools. In most cases, such automatically-created document collections will not have an accompanying DTD.

Therefore, based on the above discussion on the virtues of a DTD, it is important to devise algorithms and tools that can infer an accurate, meaningful DTD for a given collection of XML documents (i.e., *instances* of the DTD). This is *not* an easy task. Since the DTD syntax incorporates the full specification power of regular expressions, manually deducing such a DTD schema for even a small set of XML documents created by a user could prove to be a process of daunting complexity. Furthermore, as we show in this paper, naive approaches fail to deliver meaningful and intuitive DTD descriptions of the underlying data. Both problems are, of course, exacerbated for *large* XML document collections. In light of the several benefits of DTDs, we can motivate a myriad of potential applications for efficient, automated DTD discovery tools. For example, consider an employment web site that integrates information on job openings from thousands of different web sites including company home pages, newspaper classified sites, and so on. These XML documents, although related, may not all have the same structure and, even if some of the documents are accompanied by DTDs, the DTDs may not be identical. An alternative to manually transforming all the XML documents to conform to a single format would be to simply store the documents in their original formats and use DTD discovery tools to derive a single DTD description for the entire database. This inferred DTD can then help in the formulation, optimization, and processing of queries over the database of stored XML documents. Further, the ability to extract DTDs for a range of XML formats supported by the major participants in a specific industrial setting can also aid in the DTD standardization process for the industry.

**Our Contributions.** In this paper, we describe the architecture of XTRACT, a novel system for inferring an accurate, meaningful DTD schema for a repository of XML documents. A naive and straightforward solution to our DTD extraction problem would be to infer as the DTD for an element, a "concise" expression which describes *exactly* all the sequences of subelements nested within the element in the entire document collection. As we demonstrate in Section 3, however, the DTDs generated by this approach tend to be voluminous and unintuitive (especially for large XML document collections). In fact, we discover that accurate and meaningful DTD schemas that are also intuitive and appealing to humans (i.e., resemble what a human expert is likely to come up with) tend to *generalize*. That is, "good" DTDs are typically regular expressions describing subelement sequences that *may not actually occur* in the input XML documents. (Note that this, in fact, is always the case for DTD regular expressions that correspond to infinite regular languages, e.g., DTDs containing one or more Kleene stars "*" [15].) In practice, however, there are numerous such candidate DTDs that generalize the subelement sequences in the input, and choosing the DTD that best describes the structure of these sequences is a non-trivial task. In the inference algorithms employed in the XTRACT system, we propose the following novel combination of sophisticated techniques to generate DTD schemas that effectively capture the structure of the input sequences.

•**Generalization.** As a first step, the XTRACT system employs novel heuristic algorithms for finding patterns in each input sequence and replacing them with appropriate regular expressions to produce more general candidate DTDs. The main goal of the generalization step is to judiciously introduce metacharacters (like Kleene stars "*") to produce regular subexpressions that generalize the patterns observed in the input sequences. Our generalization heuristics are based on the discovery of frequent, neighboring occurrences of subsequences and symbols within each input sequence. In their effort to introduce a sufficient amount of generalization while avoiding an explosion in the number of resulting patterns, our techniques are inspired by practical, real-life DTD examples.

•**Factoring.** As a second step, the XTRACT system *factors* common subexpressions from the generalized candidate DTDs obtained from the generalization step, in order to make them more concise. The factoring algorithms applied are appropriate adaptations of techniques from the logic optimization literature [4, 24].

•**Minimum Description Length (MDL) Principle.** In the final and most important step, the XTRACT system employs Rissanen's *Minimum Description Length* (MDL) principle [21, 22] to derive an elegant mechanism for composing a near-optimal DTD schema from the set of candidate DTDs generated by the earlier two steps. (Our MDL-based notion of optimality will be defined formally later in the paper.) The MDL principle has its roots in information theory and, essentially, provides a principled, scientific definition of the optimal "theory/model" that can be inferred from a set of data examples [20]. Abstractly, in our problem setting, MDL ranks each candidate DTD depending on the number of bits required to describe the input collection of sequences *in terms of the DTD* (DTDs requiring fewer bits are ranked higher). As a consequence, the optimal DTD according to the MDL

principle is the one that is general enough to cover a large subset of the input sequences but, at the same time, captures the structure of the input sequences with a fair amount of detail, so that they can be described easily (with few additional bits) using the DTD. Thus, the MDL principle provides a formal notion of "best DTD" that exactly matches our intuition. Using MDL essentially allows XTRACT to control the amount of generalization introduced in the inferred DTD in a principled, scientific and, at the same time, intuitively appealing fashion. We demonstrate that selecting the optimal DTD based on the MDL principle has a direct and natural mapping to the *Facility Location Problem* (FLP), which is known to be NP-complete [14]. Fortunately, efficient approximation algorithms with guaranteed performance ratios have been proposed for the FLP in the literature [6], thus allowing us to efficiently compose the final DTD in a near-optimal manner.

We have implemented our XTRACT DTD derivation algorithms and conducted an extensive experimental study with both real-life and synthetic DTDs. Our findings show that, for a set of random inputs that conform to a predetermined DTD, XTRACT always produces a DTD that is either identical or very close to the original DTD. We also observe that the quality of the DTDs returned by XTRACT is far superior compared to those output by the IBM Alphaworks DDbE (Data Descriptors by Example) DTD extraction tool[1], which is unable to identify a majority of the DTDs. Further, a number of the original DTDs correctly inferred by XTRACT contain several regular expressions terms, some nested within one another. Thus, our experimental results clearly demonstrate the effectiveness of XTRACT's methodology for deducing fairly complex DTDs.

Several extensions to DTDs, e.g., Document Content Descriptors (DCDs) and XML Schemas, are being evolved by the Web community. These extensions aim to add typing information, since DTDs treat all data as strings. Therefore, XTRACT, can be used with little or no changes for inferring DCDs and XML Schemas in conjunction with other mechanisms for inferring the types. However, these proposals are still evolving – therefore, we do not concentrate on these extensions in this paper.

The work reported in this paper has been done in the context of the $\mathcal{SERENDIP}$ data mining project at Bell Laboratories (`www.bell-labs.com/projects/serendip`).

## 2 Related Work

The problem of mining DTDs from a collection of XML documents is, to the best of our knowledge, novel and has not been previously addressed in the literature. A few DTD extraction software tools can be found on the Web (e.g., the IBM Alphaworks DDbE product) – however, it has been our experience that these tools are somewhat naive in their approach and the quality of the DTDs inferred by them is poor (see Section 7).

[1] `www.alphaworks.ibm.com/formula/xml/`

The problem of extracting a schema from semistructured data has been addressed in [8, 13, 18]. Although, XML can be viewed as an instance of semistructured data, the kinds of schema considered in [8, 13, 18] are very different from a DTD. The schemas extracted by [8, 13, 18] attempt to find a typing for semistructured data. Assuming a graph-based model for semistructured data (nodes denote objects and labels on edges denote relationships between them), finding a typing is tantamount to grouping objects that have similarly labeled edges to and from similarly typed objects. The typing then describes this grouping in terms of the labels of the edges to (from) this type of objects and the types of the objects at the other end of the edge. In contrast, one can perhaps view the DTD as having already grouped all objects based on their incoming edges (tag of the element) into the same type and then describing the possible sequence of outgoing edges (subelements) as a regular expression. It is the fact that the outgoing edges from a type can be described by an arbitrary regular expression that distinguishes DTDs from the schemas in semistructured databases. Since the schemas in semistructured databases are expressed using plain sequences or sets of edges, they cannot be used to infer DTDs corresponding to arbitrary regular expressions.

The inference of formal languages from examples has a long and rich history in the field of computational learning theory, and more related to our work is the extensive study of the inference of *Deterministic Finite Automata* (DFAs) [2, 10, 11] (see also [19] for a detailed survey of the topic). The above line of work is purely theoretical and focuses on investigating the computational complexity of the language inference problem, while we are mainly interested in devising practical algorithms for real-world applications. In this sense, our research is more closely related to the work in [5] which addresses the problem of approximating *roughly equivalent* regular expressions from a long enough string, and the work in [16] where the *MDL* principle is used to infer a *pattern language* from positives examples. However, the problem tackled in [16] is much simpler than ours, since it assumes that the set of simple patterns whose subset is to be computed is available. Furthermore, the patterns considered in [16] are simple sequences that are permitted to contain single symbol wildcards. In our problem setting, unlike [16], patterns are general regular expressions and are not known apriori.

## 3 Problem Formulation and Overview

In this section, we present a precise definition of the problem of inferring a DTD from a collection of XML documents and then present an overview of the steps performed by the XTRACT system. Briefly, an XML document consists of nested element structures starting with a root element. Subelements of an element can either be elements or simple character data. A DTD is a grammar for describing the structure of an XML document. A DTD constrains the structure of an element by specifying a regular expression

that its subelement sequences have to conform to. Figure 1 illustrates an example XML document, in which the root element (`article`) has two nested subelements (`title` and `author`), and the `author` element in turn has two nested subelements. Figure 2 illustrates a DTD that our example XML document conforms to. More details on the XML specification can be found in [3]. For brevity, in the remainder of the paper, we denote elements of an XML document by a single letter from the lower-case alphabet.

```
<article>
  <title>
    A Relational Model for Large Shared
    Data Banks
  </title>
  <author>
    <name> E. F. Codd </name>
    <affiliation> IBM Research </affiliation>
  </author>
</article>
```

Figure 1: An Example XML Document

```
<!ELEMENT article(title, author*)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author(name, affiliation)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT affiliation (#PCDATA)>
```

Figure 2: An Example DTD

### 3.1 Problem Definition

Our primary focus in this paper is to infer a DTD for a collection of XML documents. Thus, for each element that appears in the document collection, our goal is to derive a regular expression that subelement sequences for the element (in the XML documents) conform to. Note that an element's DTD is completely independent of the DTD for other elements, and only restricts the sequence of subelements nested within the element. Therefore, for simplicity of exposition, in the remainder of the paper, we concentrate on the problem of extracting a DTD for a single element. We do not address the problem of computing attribute lists for an element – since these are simple lists, their computation is not particularly challenging.

Let $e$ be an element that appears in the XML documents for which we want to infer a DTD. It is straightforward to compute the sequence of subelements nested within each `<e> </e>` pair in the XML document collection. Let $I$ denote the set of $N$ such sequences, one sequence for every occurrence of element $e$ in the data. The problem we address in this paper can be stated as follows.

**Problem Statement.** Given a set $I$ of $N$ input sequences nested within element $e$, compute a DTD for $e$ such that every sequence in $I$ conforms to the DTD.□

As stated, an obvious solution to the problem is to find the most "concise" regular expression $R$ whose language is $I$. One mechanism to find such a regular expression is to factor as much as possible, the expression corresponding to the *or* of sequences in $I$. Factoring a regular expression makes it "concise" without changing the language of the expression. For example, $ab|ac$ can be factored into $a(b|c)$. An alternate method for computing the most concise regular expression is to first find the automaton with the smallest number of states that accepts $I$ and then derive the regular expression from the automaton. (Note, however, that the obtained regular expression may not be the shortest regular expression for $I$.) In any case, such a concise regular expression whose language is $I$, is unfortunately not a "good" DTD in the sense it tends to be voluminous and unintuitive. We illustrate this using the DTD of Figure 2. Suppose we have a collection of XML documents that conform to this DTD. Abbreviating the `title` tag by $t$, and the `author` tag by $a$, it is reasonable to expect the following sequences to be the subelement sequences of the `article` element in the collection of XML documents: $t$, $ta$, $taa$, $taaa$, $taaaa$. Clearly, the most concise regular expression for the above language is $t|t(a|a(a|a(a|aa)))$ which is definitely much more voluminous and much less intuitive than a DTD such as $ta^*$.

In other words, the obvious solution above never "generalizes" and would therefore never contain metacharacters like $^*$ in the inferred DTD. Clearly, a human expert would at most times want to use such metacharacters in a DTD to succinctly convey the constraints he/she wishes to impose on the structure of XML documents. Thus, the challenge is to infer for the set of input sequences $I$, a "general" DTD which is similar to what a human would come up with. However, as the following example illustrates, there can be several possible "generalizations" for a given set of input sequences and thus we need to devise a mechanism for choosing the one that best describes the sequences.

**Example 3.1** Consider $I = \{ab, abab, ababab\}$. A number of DTDs match sequences in $I$ – (1) $(a \mid b)^*$, (2) $ab \mid abab \mid ababab$, (3) $(ab)^*$, (4) $ab \mid ab(ab \mid abab)$, and so on. DTD (1) is similar to ANY in that it allows any arbitrary sequence of $a$s and $b$s, while DTD (2) is simply an *or* of all the sequences in $I$. DTD (4) is derived from DTD (2) by factoring the subsequence $ab$ from the last two disjuncts of DTD (2). The problem with DTD (1) is that it represents a gross over-generalization of the input, and the inferred DTD completely fails to capture any structure inherent in the input. On the other hand, DTDs (2) and (4) accurately reflect the structure of the input sequences but do not generalize or learn any meaningful patterns which make the DTDs smaller or simpler to understand. Thus, none of the DTDs (1), (2) or (4) seem "good". However, of the above DTDs, (3) has great intuitive appeal since it is succinct and it generalizes the set of input sequences without losing too much information about the structure of the sequences. □

Based on the discussion in the above example, we can characterize the set of desirable DTDs by placing the following two qualitative restrictions on the inferred DTD.

**R1:** The DTD should be *concise* (i.e., small in size).

**R2:** The DTD should be *precise* (i.e, not cover too many sequences not contained in $I$).

Restriction R1 above ensures that the inferred DTD is easy to understand and succinct, thus eliminating, in many cases, concise regular expressions whose language is $I$. Restriction R2, on the other hand, attempts to ensure that the DTD is not too general and captures the structure of input sequences, thus eliminating a DTD such as ANY. While the above restrictions seem reasonable at an intuitive level, in general, there is a tradeoff between a DTD's "conciseness" and its "preciseness", and a good DTD is one that strikes the right balance between the two. The problem here is that conciseness and preciseness are qualitative notions – in order to resolve the tradeoff between the two, we need to devise quantitative measures for mathematically capturing these two qualitative notions.

### 3.2 Using the MDL Principle to Define a Good DTD

We use the MDL principle [21, 22] to define an information-theoretic measure for quantifying and thereby resolving the tradeoff between the conciseness and preciseness properties of DTDs. The MDL principle has been successfully applied in the past in a variety of situations ranging from constructing good decision tree classifiers [17, 20] to learning common patterns in sets of strings [16].

Roughly speaking, the MDL principle states that the best theory to infer from a set of data is the one which minimizes the sum of

(A) the length of the theory, in bits, and

(B) the length of the data, in bits, when encoded with the help of the theory.

We will refer to the above sum for a theory, as the *MDL cost* for the theory. The MDL principle is a general one and needs to be instantiated appropriately for each situation. In our setting, the theory is the DTD and the data is the sequences in $I$. Thus, the MDL principle assigns each DTD an MDL cost and ranks the DTDs based on their MDL costs (DTDs with lower MDL costs are ranked higher). Furthermore, parts (A) and (B) of the MDL cost for a DTD depend directly on its conciseness and preciseness, respectively. Part (A) is the number of bits required to describe the DTD and is thus a direct measure of its conciseness. Further, since a DTD that is more precise captures the structure of the input sequences more accurately, fewer bits are required to describe the sequences in $I$ in terms of a more precise DTD. As a result, Part (B) of the MDL cost captures a DTD's preciseness. The MDL cost for a DTD thus provides us with an elegant and principled mechanism (rooted in information theory) for quantifying (and combining) the conflicting concepts of conciseness and preciseness in a single unified framework.

Note that the actual encoding scheme used to specify a DTD as well as the data (with the help of the DTD) plays a critical role in determining the actual values for the two components of the MDL cost. We defer the details of the actual encoding scheme to Section 4. However, in the following example, we employ a simple encoding scheme (a coarser version of the scheme in Section 4) to illustrate how ranking DTDs based on their MDL cost closely matches our intuition of their "goodness".

**Example 3.2** Consider the input set $I$ and DTDs from Example 3.1. We compute the MDL cost of each DTD, which, as mentioned earlier, is the cost of encoding the DTD itself and the sequences in $I$ in terms of the DTD. We then rank the DTDs based on their MDL costs (DTDs with smaller costs are considered better). In our simple encoding scheme, we assume a cost of 1 unit for each character.

DTD (1), $(a \mid b)^*$, has a cost of 6 for encoding the DTD. In order to encode the sequence $abab$ using the DTD, we need one character to specify the number of repetitions of the the term $(a \mid b)$ that precedes the $^*$ (in this case, this number is 4), and 4 additional characters to specify which of $a$ or $b$ is chosen from each repetition. Thus, the total cost of encoding $abab$ using $(a \mid b)^*$ is 5 and the MDL cost of the DTD is $6 + 3 + 5 + 7 = 21$. Similarly, the MDL cost of DTD (2) can be shown to be 14 (to encode the DTD) + 3 (to encode the input sequences; we need one character to specify the position of the disjunct for each sequence) = 17. The cost of DTD (3) is 5 (to encode the DTD) + 3 (to encode the input sequences – note that we only need to specify the number of repetitions of the term $ab$ for each sequence) = 8. Finally, DTD (4) has a cost of 14 + 5 (1 character to encode sequence $ab$ and 2 characters for each of the other two input sequences) = 19. Thus, since DTD (3) has the least MDL cost, it would be considered the best DTD by the MDL principle – which matches our intuition. □

### 3.3 Overview of the XTRACT System

The architecture of the XTRACT system is illustrated in Figure 3(a). As shown in the figure, the system consists of three main components: the generalization module, the factoring module, and the MDL module. Input sequences in $I$ are processed by the three subsystems one after another, the output of one subsystem serving as input to the next. We denote the outputs of the generalization and factoring modules by $\mathcal{S}_{\mathcal{G}}$ and $\mathcal{S}_{\mathcal{F}}$, respectively. Observe that both $\mathcal{S}_{\mathcal{G}}$ and $\mathcal{S}_{\mathcal{F}}$ contain the initial input sequences in $I$. This is to ensure that the MDL module has a wide range of DTDs to choose from that includes the obvious DTD which is simply an *or* of all the input sequences in $I$. In the following, we provide a brief description of each subsystem; we defer a more detailed description of the algorithms employed by each subsystem to later sections.

**The Generalization Subsystem.** For each input sequence, the generalization module generates zero or more candidate

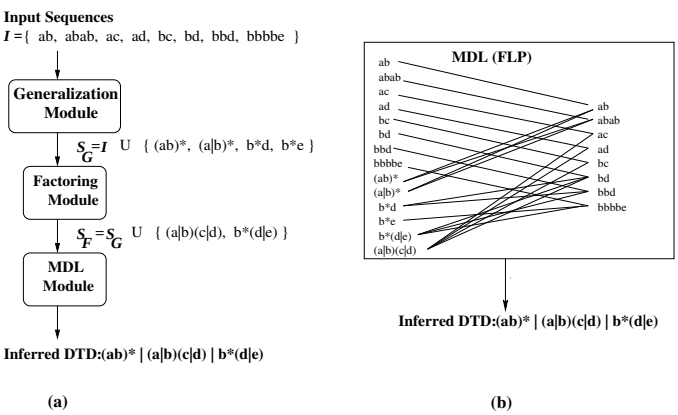


Figure 3: Architecture of the XTRACT System

DTDs that are derived by replacing patterns in the input sequence with regular expressions containing metacharacters like * and | (e.g., $(ab)^*$, $(a \mid b)^*$). Note that the initial input sequences do not contain metacharacters and so the candidate DTDs introduced by the generalization module are more general. For instance, in Figure 3(a), sequences $abab$ and $bbbe$ result in the more general candidate DTDs $(ab)^*$, $(a \mid b)^*$ and $b^*e$ being output by the generalization subsystem. Also, observe that each candidate DTD produced by the generalization module may cover only a subset of the input sequences. Thus, the final DTD output by the MDL module may be an *or* of multiple candidate DTDs.

Ideally, in the generalization phase, we should consider all DTDs that cover one or more input sequences as candidates, so that the MDL step can choose the best among them. However, the number of such DTDs can be enormous. For example, the sequence $ababaabb$ is covered by the following DTDs in addition to many more – $(a \mid b)^*$, $(a \mid b)^*a^*b^*$, $(ab)^*(a \mid b)^*$, $(ab)^*a^*b^*$. Therefore, in this paper, we outline several novel heuristics, inspired by real-life DTDs[2], for limiting the set of candidate DTDs $\mathcal{S}_\mathcal{G}$ output by the generalization module.

**The Factoring Subsystem.** The factoring component factors two or more candidate DTDs in $\mathcal{S}_\mathcal{G}$ into a new candidate DTD. The length of the new DTD is smaller than the sum of the sizes of the DTDs factored. For example, in Figure 3(a), candidate DTDs $b^*d$ and $b^*e$ representing the expression $b^*d \mid b^*e$, when factored, result in the DTD $b^*(d \mid e)$; similarly, the candidates $ac$, $ad$, $bc$ and $bd$ are factored into $(a \mid b)(c \mid d)$ (the pre-factored expression is $ac \mid ad \mid bc \mid bd$). Although factoring leaves the semantics of candidate DTDs unchanged, it is nevertheless an important step. The reason being that factoring reduces the size of the DTD and thus the cost of encoding the DTD, without seriously impacting the cost of encoding input sequences using the DTD. Thus, since the DTD encoding cost is a component of the MDL cost for a DTD, factoring can result in certain DTDs being chosen by the MDL module that may not have been considered earlier. We appropriately modify factoring algorithms for boolean functions in the logic optimization area [4, 24] to meet our needs. However, even though every subset of candidate DTDs can, in principle, be factored, the number of these subsets can be huge and only a few of them result in good factorizations. We propose novel heuristics to restrict our attention to subsets that can be factored effectively.

**The MDL Subsystem.** The MDL subsystem finally chooses from among the set of candidate DTDs $\mathcal{S}_\mathcal{F}$ generated by the previous two subsystems, a set of DTDs that cover all the input sequences in $I$ and the sum of whose MDL costs is minimum. The final DTD is then an *or* of the DTDs in the set. For the input sequences in Figure 3(a), we illustrate (using solid lines) in Figure 3(b), the input sequences (in the right column) covered by the candidate DTDs in $\mathcal{S}_\mathcal{F}$ (in the left column).

The above cost minimization problem naturally maps to the *Facility Location Problem* (FLP) for which polynomial-time approximation algorithms have been proposed in the literature [6, 14]. We adapt the algorithm from [6] for our purposes, and using it, the XTRACT system is able to infer the DTD shown at the bottom of Figure 3(b).

# 4 The MDL Subsystem

The MDL subsystem constitutes the core of the XTRACT system – it is responsible for choosing a set $\mathcal{S}$ of candidate DTDs from $\mathcal{S}_\mathcal{F}$ such that the final DTD $\mathcal{D}$ (which is an *or* of the DTDs in $\mathcal{S}$) (1) covers all sequences in $I$, and (2) has minimum MDL cost. Consequently, we describe this module first, and postpone the presentation of the generalization and factoring modules to Sections 5 and 6, respectively.

Recall that the MDL cost of a DTD that is used to explain a set of sequences, comprises (A) the length, in bits, needed to describe the DTD; and (B) the length of the sequences (in bits) when encoded in terms of the DTD. In the following subsection, we first present the encoding schemes for computing parts (A) and (B) of the MDL cost of a DTD. Subsequently, in Section 4.2, we present the algorithm for computing the set $\mathcal{S} \subseteq \mathcal{S}_\mathcal{F}$ of candidate DTDs whose *or* yields the final DTD $\mathcal{D}$ with minimum MDL cost. Note that the candidate DTDs in $\mathcal{S}_\mathcal{F}$ can be complex regular expressions (containing *, |, etc.) output by the generalization and factoring subsystems.

## 4.1 The Encoding Scheme

We begin by describing the procedure for estimating the number of bits required to encode the DTD itself (part (A) of the MDL cost). Let $\Sigma$ be the set of subelement symbols that appear in sequences in $I$. Let $\mathcal{M}$ be the set of metacharacters $|, ^*, +, ?, (, )$. Let the length of a DTD viewed as a string

[2]The DTDs are available at Robin Cover's SGML/XML web page (`www.oasis-open.org/cover/`).

**(A)** $seq(D, s) = \epsilon$ if $D = s$. In this case, the DTD $D$ is a sequence of symbols from the alphabet $\Sigma$ and does not contain any metacharacters.

**(B)** $seq(D_1 \ldots D_k, s_1 \ldots s_k) = seq(D_1, s_1) \ldots seq(D_k, s_k)$; that is, $D$ is the concatenation of regular expressions $D_1 \ldots D_k$ and the sequence $s$ can be written as the concatenation of the subsequences $s_1 \ldots s_k$, such that each subsequence $s_i$ matches the corresponding regular expression $D_i$.

**(C)** $seq(D_1|\ldots|D_m, s) = i\ seq(D_i, s)$; that is, $D$ is the exclusive choice of regular expressions $D_1 \ldots D_m$, and $i$ is the index of the regular expression that the sequence $s$ matches. Note that we need $\lceil \log m \rceil$ bits to encode the index $i$.

**(D)** $seq(D^*, s_1 \ldots s_k) = \begin{cases} k\ seq(D, s_1) \ldots seq(D, s_k) & \text{if } k > 0 \\ 0 & \text{otherwise} \end{cases}$

In other words, the sequence $s = s_1 \ldots s_k$ is produced from $D^*$ by instantiating the repetition operator $k$ times, and each subsequence $s_i$ matches the $i$-th instantiation. In this case, since there is no simple and inexpensive way to bound apriori the number of bits required for the index $k$, we first specify the number of bits required to encode $k$ in unary (that is, a sequence of $\lceil \log k \rceil$ 1s, followed by a 0) and then the index $k$ using $\lceil \log k \rceil$ bits. The 0 in the middle serves as the delimiter between the unary encoding of the length of the index and the actual index itself.

Figure 4: The Encoding Scheme

in $\Sigma \cup \mathcal{M}$, be $n$. Then, the length of the DTD in bits is $n\lceil \log(|\ \Sigma\ |\ +\ |\ \mathcal{M}\ |) \rceil$. As an example, let $\Sigma$ consist of the elements $a$ and $b$. The length in bits of the DTD $a^*b^*$ is $4 * \lceil \log(2 + 6) \rceil = 12$. Similarly, the length in bits of the DTD $(ab|abb)(aa|ab^*)$ is $16 * 3 = 48$.

We next describe the scheme for encoding a sequence using a DTD (part (B) of the MDL cost). Our encoding scheme constructs a sequence of integral indices (which forms the encoding) for expressing a sequence in terms of a DTD. The following simple examples illustrate the basic building blocks on which our encoding scheme for more complex DTDs is built:

1. The encoding for the sequence $a$ in terms of the DTD $a$ is the empty string $\epsilon$.
2. The encoding for the sequence $b$ in terms of the DTD $a\ |\ b\ |\ c$ is the integral index 1 (denotes that $b$ is at position 1, counting from 0, in the above DTD).
3. The encoding for the sequence $bbb$ in terms of the DTD $b^*$ is the integral index 3 (denotes 3 repetitions of $b$).

We now generalize the encoding scheme for arbitrary DTDs and arbitrary sequences. Let us denote the sequence of integral indices for a sequence $s$ when encoded in terms of a DTD $D$ by $seq(D, s)$. We define $seq(D, s)$ recursively in terms of component DTDs within $D$ as shown in Figure 4. Thus, $seq(D, s)$ can be computed using a recursive procedure based on the encoding scheme of Figure 4. Note that we have not provided the definitions of the encodings for operators $^+$ and ? since these can be defined in a similar fashion to $^*$ (for $^+$, $k$ is always greater than 0, while for ?, $k$ can only assume values 1 or 0). We now illustrate our encoding scheme using the following example.

**Example 4.1** Consider the DTD $(ab|c)^*(de|fg^*)$ and the sequence $abccabfggg$ to be encoded in terms of the DTD. Below, we list how steps (A), (B), (C) and (D) in Figure 4 are recursively applied to derive the encoding $seq((ab|c)^*(de|fg^*), abccabfggg)$.

1. **Apply Step (B):** $seq((ab|c)^*, abccab)\ seq((de|fg^*), fggg)$
2. **Apply Step (D):** 4 $seq(ab|c, ab)\ seq(ab|c, c)\ seq(ab|c, c)\ seq(ab|c, ab)\ seq((de|fg^*), fggg)$
3. **Apply Step (C):** 4 0 $seq(ab, ab)$ 1 $seq(c, c)$ 1 $seq(c, c)$ 0 $seq(ab, ab)$ 1 $seq(fg^*, fggg)$
4. **Apply Step (A):** 4 0 1 1 0 1 $seq(fg^*, fggg)$
5. **Apply Steps (A), (B) and (D):** 4 0 1 1 0 1 3

In order to derive the final bit sequence corresponding to the above indices, we need to include in the encoding the unary representation for the number of bits required to encode the indices 4 and 3. Thus, we obtain the following bit encoding for the sequence (we have inserted blanks in between the encoding for successive indices for clarity).

$seq((ab|c)^*(de|fg^*), abccabfggg) = 1110100\ 0\ 1\ 1\ 0\ 1\ 11011$

□

In steps (B), (C) and (D), we need to be able to determine if a sequence $s$ matches a DTD $D$. Since a DTD is a regular expression, well-established techniques for finding out if a sequence is covered by a regular expression can be used for this purpose [15] and have a complexity of $O(|D| \cdot |s|)$ ($|s|$ denotes the length of sequence $s$). These methods involve constructing a non-deterministic finite automaton for $D$ and can also be used to decompose the sequence $s$ into subsequences such that each subsequence matches the corresponding sub-part of the DTD $D$, thus enabling us to come up with the encoding.

Note that there may be multiple ways of partitioning the sequence $s$ such that each subsequence matches the corresponding sub-part of the DTD $D$. In such a case, we can extend the above procedure to enumerate every decomposition of $s$ that match sub-parts of $D$, and then select from among the decompositions the one that results in the minimum length encoding of $s$ in terms of $D$. The complexity of considering all possible decompositions, however, is much higher and therefore not included in our XTRACT implementation.

### 4.2 Computing the DTD with Minimum MDL Cost

We now turn our attention to the problem of computing the final DTD $\mathcal{D}$ (which is an *or* of a subset $\mathcal{S}$ of candidate DTDs in $\mathcal{S}_{\mathcal{F}}$) that covers all the input sequences in $I$ and whose MDL cost for encoding sequences in $I$ is minimum. The above minimization problem maps naturally to the *Facility*

*Location Problem* (FLP) [6, 14]. The FLP is formulated as follows: Let $C$ be a set of clients and $J$ be a set of facilities such that each facility "serves" every client. There is a cost $c(j)$ of "choosing" a facility $j \in J$ and a cost $d(j,i)$ of serving client $i \in C$ by facility $j \in J$. The problem definition asks to choose a subset of facilities $F \subset J$ such that the sum of costs of the chosen facilities plus the sum of costs of serving every client by its closest chosen facility is minimized, that is

$$\min_{F \subset J} \{ \sum_{j \in F} c(j) + \sum_{i \in C} \min_{j \in F} d(j,i) \ \}. \quad (1)$$

The problem of inferring the minimum MDL cost DTD can be reduced to FLP as follows: Let $C$ be the set $I$ of input sequences and $J$ be the set of candidate DTDs in $\mathcal{S}_\mathcal{F}$. The cost of choosing a facility is the length of the corresponding candidate DTD. The cost of serving client $i$ from facility $j$, $d(j,i)$, is the length of the encoding of the sequence corresponding to client $i$ using the DTD corresponding to facility $j$. If a DTD $j$ does not cover a sequence $i$, then we set $d(j,i)$ to $\infty$. Thus, the set $F$ computed by the FLP corresponds to our desired set $\mathcal{S}$ of candidate DTDs.

The FLP is NP-hard; however, it can be reduced to the *set cover problem* and then approximated within a logarithmic factor as shown in [14]. In our implementation, we employ the randomized algorithm from [6] which approximates the FLP within a constant factor if the distance function is a metric. Even though our distance function is not a metric, we have found the FLP approximations produced by [6] for our problem setting to be very good in practice. Furthermore, the time complexity of [6] for computing the approximate solution is $O(N^2 \cdot \log N)$, where $N = |I|$.

# 5 The Generalization Subsystem

The quality of the DTD computed by the MDL module is very dependent on the set of candidate DTDs $\mathcal{S}_\mathcal{F}$ input to it. In case $\mathcal{S}_\mathcal{F}$ were to contain only input sequences in $I$, then the final DTD output by the MDL subsystem would simply be the *or* of all the sequences in $I$. However, as we observed earlier, this is not a desirable DTD since it is neither concise nor intuitive. Thus, in order to infer meaningful DTDs, it is crucial that the candidate DTDs in $\mathcal{S}_\mathcal{F}$ be *general*. The goal of the generalization component is to achieve this objective by inferring a set $\mathcal{S}_\mathcal{G}$ of general DTDs which are then input to the factorization step. As we mentioned before, the factorization step infers additional factored DTDs and generates $\mathcal{S}_\mathcal{F}$ which is a superset of $\mathcal{S}_\mathcal{G}$.

The generalization component in XTRACT infers a number of regular expressions which we have found to frequently appear in real-life DTDs. Some of the most common regular expressions from real-life DTDs are $a^*bc^*$, $(abc)^*$, $(a|b|c)^*$, and $((ab)^*c)^*$. These are examples that appear in the Newspaper Association of America (NAA) Classified Advertizing Standards XML DTD[3]. (A detailed description of the NAA data can be found in the full version of this paper [9].)

Although our algorithms can infer regular expressions that are more complex than the above, we do not infer certain complex expressions such as $(ab?c^*d?)^*$ that are less likely to occur in practice. We defer further discussion of this topic to Section 7.

We now discuss our generalization algorithm which is outlined in Figure 5. Procedure GENERALIZE infers several DTDs for each input sequence in $I$ independently and adds them to the set $\mathcal{S}_\mathcal{G}$. Therefore, it may over-generalize in some cases (since we are inferring DTDs based on a single sequence); however, our MDL step will ensure that such overly general DTDs are not chosen as part of the final inferred DTD, if there are better alternatives. Recall that the generalization step is merely trying to provide several alternate candidates to the MDL step. In particular, $\mathcal{S}_\mathcal{G} \supseteq$ I, and therefore, the DTD corresponding to the *or*'s of the input will always be considered during the MDL step.

The essence of procedure GENERALIZE lies in the two procedures DISCOVERSEQPATTERN and DISCOVEROR-PATTERN, which are repeatedly called with various parameter values. We discuss the details of these procedures and the roles of their parameters next.

## 5.1 Discovering Sequencing Patterns

Procedure DISCOVERSEQPATTERN, shown in Figure 5, takes as input an input sequence $s$ and returns a candidate DTD that is derived from $s$ by replacing sequencing patterns of the form $xx \cdots x$, for a subsequence $x$ in $s$, with the regular expression $(x)^*$. In addition to $s$, the procedure also accepts as input a threshold parameter $r > 1$ which is the minimum number of contiguous repetitions of subsequence $x$ in $s$ required for the repetitions to be replaced with $(x)^*$. In case there are multiple subsequences $x$ with the maximum number of repetitions in Step 2, the longest among them is chosen, and subsequent ties are resolved arbitrarily.

Note that instead of introducing the regular expression term $(x)^*$ into the sequence $s$, we choose to introduce an *auxiliary* symbol that serves as a representative for the term. The auxiliary symbols enable us to keep the description of our algorithms simple and clean, since their input is always a simple sequence of symbols. We ensure that there is a one-to-one correspondence between auxiliary symbols and regular expression terms throughout the XTRACT system; thus, if the auxiliary symbol, $A$ denotes $(bc)^*$ in one candidate DTD, then it represents $(bc)^*$ in every other candidate DTD. Also, observe that procedure DISCOVERSEQPATTERN may perform several iterations and thus new sequencing patterns may contain auxiliary symbols corresponding to patterns replaced in previous iterations. For example, invoking procedure DISCOVERSEQPATTERN with the input sequence $s = abababcabab$ and $r = 2$ yields the sequence $A_1cA_1c$ after the first iteration, where $A_1$ is an auxiliary symbol for the term $(ab)^*$. After the second iteration, the procedure returns the candidate DTD $A_2$, where $A_2$ is the auxiliary sym-

[3] www.naa.org/technology/clsstdtf/Adex010.dtd

```
procedure GENERALIZE(I)
begin
1.  for each sequence s in I
2.      add s to S_G
3.      for r := 2, 3, 4
4.          s' := DISCOVERSEQPATTERN(s, r)
5.          for d := 0.1 · |s'|, 0.5 · |s'|, |s'|
6.              s'' := DISCOVERORPATTERN(s', d)
7.              add s'' to S_G
end

procedure DISCOVERSEQPATTERN(s, r)
begin
1.  repeat
2.      let x be a subsequence of s with the maximum
3.        number (≥ r) of contiguous repetitions in s
4.      replace all (≥ r) contiguous occurrences of
5.        x in s with a new auxiliary symbol A_i = (x)*
6.  until (s no longer contains ≥ r contiguous
7.        occurrences of any subsequence x)
8.  return s
end

procedure DISCOVERORPATTERN(s, d)
begin
1.  s_1, s_2, ..., s_n := PARTITION(s, d)
2.      for each subsequence s_j in s_1, s_2, ..., s_n
3.          let the set of distinct symbols in s_j
4.            be a_1, a_2, ..., a_m
5.          if (m > 1)
6.              replace subsequence s_j in sequence s by a
7.              new auxiliary symbol A_i = (a_1|···|a_m)*
8.  return s
end

procedure PARTITION(s, d)
begin
1.  i := start := end := 1
2.  s_i = s[start, end]
3.  while (end < |s|)
4.      while (end < |s| and a symbol in s_i occurs to
5.              the right of s_i within a distance d)
6.          end := end + 1;
7.          s_i := s[start, end];
8.      if (end < |s|)
9.          i := i + 1;
10.         start := end + 1;
11.         end := end + 1;
12.         s_i := s[start, end];
13. return s_1, s_2, ···, s_i
end
```

Figure 5: The Generalization Algorithm

bol corresponding to $((ab)^*c)^*$. Thus, the resulting candidate DTD returned by procedure DISCOVERSEQPATTERN can contain $^*$'s nested within other $^*$'s. Finally, we have chosen to invoke DISCOVERSEQPATTERN (from GENERALIZE) with three different values for the parameter $r$ to control the eagerness with which we generalize. For example, for the sequence $aabbb$, DISCOVERSEQPATTERN with $r = 2$ would infer $a^*b^*$, while with $r = 3$, it would infer $aab^*$. In the MDL step, if many other sequences are covered by $aab^*$, then a DTD of $aab^*$ may be preferred to a DTD of $a^*b^*$, since it more accurately describes sequences in $I$.

The time complexity of the procedure is dominated by the first step that involves finding the subsequence $x$ with the maximum number of contiguous repetitions. Since $s$ contains at most $O(|s|^2)$ possible subsequences and computing the number of repetitions for each subsequence takes $O(|s|)$ steps, the complexity of the first step is $O(|s|^3)$ per iteration, in the worst case.

### 5.2 Discovering Or Patterns

Procedure DISCOVERORPATTERN infers patterns of the form $(a_1|a_2|\ldots|a_m)^*$ based on the locality of these symbols within a sequence $s$. It discovers such localities by first partitioning (using procedure PARTITION) the input sequence $s$ into the smallest possible subsequences $s_1, s_2, \ldots, s_n$, such that for any occurrence of a symbol $a$ in a subsequence $s_i$, there does not exist another occurrence of $a$ in some other subsequence $s_j$ within a distance $d$ (which is a parameter to DISCOVERORPATTERN). Each subsequence $s_i$ in $s$ is then replaced by the pattern $(a_1|a_2|\ldots|a_m)^*$ where $a_1, \ldots, a_m$ are the distinct symbols in the subsequence $s_i$. The intuition here is that if $s_i$ contains frequent repetitions of the symbols $a_1, a_2, \ldots, a_m$ in close proximity, then it is very likely that $s_i$ originated from a regular expression of the form $(a_1|a_2|\ldots|a_m)^*$. As an illustration, on the input sequence $abcbac$, procedure DISCOVERORPATTERN returns

- $aA_1ac$ for $d = 2$, where $A_1 = (b \mid c)^*$,
- $aA_2$ for $d = 3$, where $A_2 = (a \mid b \mid c)^*$, and
- $A_2$ for $d = 4$, where $A_2 = (a \mid b \mid c)^*$.

A critical component for discovering *or* patterns is procedure PARTITION, which we now discuss in more detail. Before that, we define the following notation for sequences. For a sequence $s$, $s[i, j]$ denotes the subsequence of $s$ starting at the $i^{th}$ symbol and ending at the $j^{th}$ symbol of $s$. Procedure PARTITION constructs subsequences of $s$ in the order $s_1$, $s_2$, and so on. Assuming that $s_1$ through $s_j$ have been generated, it constructs $s_{j+1}$ by starting $s_{j+1}$ immediately after $s_j$ ends and expanding the subsequence $s_{j+1}$ to the right as long as required to ensure that there is no symbol in $s_{j+1}$ that occurs within a distance $d$ to the right of $s_{j+1}$. By construction, there cannot exist such a symbol to the left of $s_{j+1}$. Note that the condition of whether a symbol in $s_i$ occurs within a distance $d$ outside $s_i$ can be checked in $O(|s|)$ time if we keep track of the next occurrence outside $s_i$

of every symbol in $s_i$ – this can be achieved by initially constructing for every symbol, the locations of its occurrences in $s$ sorted order. Therefore, the time complexity of procedures PARTITION and DISCOVERORPATTERN can be easily shown to be $O(|s|^2)$.

Note that procedure GENERALIZE invokes DISCOVERORPATTERN on the DTDs that result from calls to DISCOVERSEQPATTERN and therefore it is possible to infer more complex DTDs of the form $(a|(bc)^*)^*$ in addition to DTDs like $(a|b|c)^*$. For instance, for the input sequence $s = abcbca$, procedure DISCOVERSEQPATTERN invoked with $r = 2$ would return $s' = aA_1a$, where $A_1 = (bc)^*$, which when input to DISCOVERORPATTERN returns $s'' = A_2$ for $d = |s'|$, where $A_2 = (a|A_1)^*$. Further, observe that DISCOVERORPATTERN is invoked with various values of $d$ (expressed as a fraction of the length of the input sequence) to control the degree of generalization. Small values of $d$ lead to conservative generalizations while larger values result in more liberal generalizations.

# 6 The Factoring Subsystem

In a nutshell, the factoring step derives factored forms for expressions consisting of an *or* of a subset of the candidate DTDs in $\mathcal{S}_\mathcal{G}$. For example, for candidate DTDs $ac$, $ad$, $bc$ and $bd$ in $\mathcal{S}_\mathcal{G}$, the factoring step would generate the factored form $(a \mid b)(c \mid d)$. Note that since the final DTD is an *or* of candidate DTDs in $\mathcal{S}_\mathcal{F}$, factored forms are candidates, too. Further, a factored candidate DTD, because of its smaller size, has a lower MDL cost, and is thus more likely to be chosen in the MDL step. Thus, since factored forms (due to their compactness) are more desirable (see restriction **R1** in Section 3), factoring can result in better-quality DTDs. In this section, we describe the algorithms used by the factoring module to derive factored forms of the candidate DTDs in $\mathcal{S}_\mathcal{G}$ produced by the generalization step.

Factored DTDs are common in real life, when there are several choices to be made. For example, in the DTD in Figure 2, an article may be categorized based on whether it appeared in a workshop, conference, or journal; it may also be classified according to its area as belonging to either computer science, physics, chemistry, etc. As a consequence, the DTD (in factored form) for the element `article` could be as follows:

```
<!ELEMENT article(title, author*,
(workshop | conference | journal),
(computer science | physics | chemistry | ...))
```

In addition to the factored forms generated from candidates in $\mathcal{S}_\mathcal{G}$, the set of candidate DTDs output by the factorization module, $\mathcal{S}_\mathcal{F}$, also contains all the DTDs in $\mathcal{S}_\mathcal{G}$. Ideally, factored forms for every subset of $\mathcal{S}_\mathcal{G}$ should be added to $\mathcal{S}_\mathcal{F}$ to be considered by the MDL module. However, this is clearly impractical, since $\mathcal{S}_\mathcal{G}$ could be very large. We have devised a heuristic strategy for selecting sets of candidates in $\mathcal{S}_\mathcal{G}$ that when factored yield "good" factored DTDs. Intuitively, our heuristic greedily selects DTDs from $\mathcal{S}_\mathcal{G}$ that (1) share common prefixes or suffixes, and (2) have minimal overlap with other selected DTDs. Due to space constraints, we omit the detailed description of our heuristic as well as the factoring algorithm itself (which is an adaptation of factoring algorithms for boolean expressions from the logic optimization literature [24]). The details can be found in the full version of this paper [9].

# 7 Experimental Results

To determine the effectiveness of XTRACT's methodology for inferring the DTD of a database of XML documents, we conducted a study with both synthetic and real-life DTDs. We also compared the DTDs produced by XTRACT with those generated by the IBM Alphaworks DTD extraction tool, DDbE[4], for XML data. Our results indicate that XTRACT outperforms DDbE over a wide range of DTDs, and accurately finds almost every original DTD while DDbE fails to do so for most DTDs. Thus, our results clearly demonstrate the effectiveness of XTRACT's approach that employs generalization and factorization to derive a range of general and concise candidate DTDs, and then uses the MDL principle as the basis to select amongst them.

## 7.1 Algorithms

In the following, we describe the two DTD extraction algorithms that we considered in our experimental study.

**XTRACT:** Our implementation of XTRACT includes all three modules as described in Sections 4, 5, and 6. In the generalization step, we discover both sequencing and *or* patterns using procedure GENERALIZE. In the factoring step, $k = \frac{N}{10}$ subsets are chosen for factoring and the parameter $\delta$ is set to 0 in the procedure FACTORSUBSETS. Finally, in the MDL step, we employ the algorithm from [6] to compute an approximation to the FLP solution.

**DDbE:** We used Version 1.0 of the DDbE DTD extraction tool in our experiments. DDbE is a Java component library for inferring a DTD from a data set consisting of well-formed XML instances. DDbE offers parameters that allow the user to control the structure of the content models and the types used for attribute declarations. Two important parameters of DDbE are (1) the maximum number of consecutive identical tokens that should not be replaced by a list, and (2) the number of applications of factoring. For our experiments we used the default values of these parameters, which are 1 and 2, respectively [9].

## 7.2 Data Sets

In order to evaluate the quality of DTDs retrieved by XTRACT, we used both synthetic as well as real-life DTD schemas. For each DTD for a single element, we generated an XML file containing 1000 instantiations of the element. These 1000 instantiations were generated by randomly sampling from the DTD for the element. Thus, the initial set of input sequences $I$ to both XTRACT and DDbE contained

[4] `www.alphaworks.ibm.com/formula/xml/`

| No. | Original DTD |
|---|---|
| 1 | $abcde\|efgh\|ij\|klm$ |
| 2 | $(a\|b\|c\|d\|f)^*gh$ |
| 3 | $(a\|b\|c\|d)^*\|e$ |
| 4 | $(abcde)^*f$ |
| 5 | $(ab)^*\|cdef\|(ghi)^*$ |
| 6 | $abcdef(g\|h\|i\|j)(k\|l\|m\|n\|o)$ |
| 7 | $(a\|b\|c)d^*e^*(fgh)^*$ |
| 8 | $(a\|b)(cdefg)^*hijklmnopq(r\|s)^*$ |
| 9 | $(abcd)^*\|(e\|f\|g)^*\|h\|(ijklm)^*$ |
| 10 | $a^*\|(b\|c\|d\|e\|f)^*\|gh\|(i\|j\|k)^*\|(lmn)^*$ |

Table 1: Synthetic DTD Data Set

somewhere between 500 and 1000 sequences (after the elimination of duplicates) conforming to the original DTD.

**Synthetic DTD Data Set.** We used a synthetic data generator to generate the synthetic data sets. Each DTD is randomly chosen to have one of the following two forms: $A_1|A_2|A_3|\cdots|A_n$ and $A_1A_2A_3\cdots A_n$. Thus, a DTD has $n$ building blocks with $n$ randomly chosen between 1 and $mb$, where $mb$ is an input parameter to the generator that specifies the maximum number of building blocks in a DTD. Each building block $A_i$ further consists of $n_i$ symbols, where $n_i$ is randomly chosen to be between 1 and $ms$ (the parameter $ms$ specifies the maximum number of symbols that can be contained in a building block). Each building block $A_i$ has one of the following four forms, each of which has an equal probability of occurrence: (1) $(a_1|a_2|a_3|\ldots|a_{n_i})$, (2) $a_1a_2a_3\ldots a_{n_i}$, (3) $(a_1|a_2|a_3|a_4|\ldots|a_{n_i})^*$; and (4) $(a_1a_2a_3a_4\ldots a_{n_i})^*$. Here, the $a_i$'s denote subelement symbols. Thus, our synthetic data generator essentially generates DTDs containing one level of nesting of regular expression terms.

In Table 1, we show the synthetic DTDs that we considered in our experiments. Note that, in the figure, we only include the regular expression corresponding to the DTD. The DTDs were produced using our generator with the input parameters $mb$ and $ms$ both set to 5. (We use letters from the alphabet as subelement symbols.)

The ten synthetic DTDs vary in complexity with later DTDs being more complex than earlier ones. For instance, DTD 1 does not contain any metacharacters, while DTDs 2 through 5 contain simple sequencing and *or* patterns. DTD 6 represents a DTD in factored form, while in DTDs 7 through 10 factors are combined with sequencing and *or* patterns.

**Real-life DTD Data Set.** We obtained our real-life DTDs from the Newspaper Association of America (NAA) Classified Advertising Standards XML DTD produced by the NAA Classified Advertising Standards Task Force[5]. We examined this real-life DTD data and collected six representative DTDs that are shown in Table 2. Of the DTDs shown in the table, the last three DTDs are quite interesting.

[5] www.naa.org/technology/clsstdtf/Adex010.dtd

| No | Original DTD | Simplified DTD |
|---|---|---|
| 1 | `<!ENTITY included-elements (audio-clip\|blind-box-reply\| link\|pi-char\|video-clip)>` | $a\|b\|c\|d\|e$ |
| 2 | `<!ELEMENT communications-contacts (phone\|fax\|email\|pager\|web-page)*>` | $(a\|b\|c\|d\|e)^*$ |
| 3 | `<!ELEMENT employment-services (employment-service.type, employment-service.location* (e.zz-generic-tag)*)>` | $ab^*c^*$ |
| 4 | `<!ENTITY location (addr*,area?, city?,state?,zip-code?,country?)>` | $a^*b?c?d?$ |
| 5 | `<!ELEMENT transfer-info (transfer-number,(from-to,co-id)+, contact-info)*>` | $(a(bc)^+d)^*$ |
| 6 | `<!ELEMENT real-estate-services (r-e.type,r-e.location?, r-e.response-modes*,r-e.comment?)*>` | $(ab?c^*d?)^*$ |

Table 2: Real-life DTD Data Set

DTD 4 contains the metacharacter ? in conjunction with the metacharacter *, while DTDs 5 and 6 contain two regular expressions with *'s, one nested within the other.

### 7.3 Quality of Inferred DTDs

**Synthetic DTD Data Set.** For the synthetic data set, XTRACT infers *each* of the original DTDs correctly. In contrast, DDbE computes the accurate DTD only for DTD 1, which is the simplest DTD containing no metacharacters. Even for the simple DTDs 2–5, not only is DDbE unable to correctly deduce the original DTD, but it also infers a DTD that does not cover the set of input sequences. In addition, DDbE is not very good at factoring DTDs and, unlike XTRACT, it is unable to derive the final factored form for DTD 6. Finally, DDbE infers an extremely complex DTD for the simple DTD 7 and overly general DTDs for the more complex DTDs 8–10. (The exact DTDs inferred by DDbE can be found in the full version of this paper [9].) Our results for synthetic data clearly demonstrate the superiority of XTRACT's approach (based on the combination of generalization, factoring, and the MDL principle) compared to that of DDbE for the DTD inference problem.

**Real-life DTD Data Set.** The DTDs generated by the two algorithms for the real-life data set are shown in Table 3. Of the six DTDs, XTRACT is able to infer the first five correctly. In contrast, DDbE is able to derive the accurate DTD only for DTDs 1 and 2, and an approximate DTD for DTD 3. Basically, with an additional factoring step, DDbE could obtain the original DTD for DTD 3. Note, however, that DDbE is unable to infer the simple DTD 4 that contains the metacharacter ?. In contrast, XTRACT is able to deduce this DTD because its factorization step takes into account the identity element "1" and simplifies expressions

| No | Simplified DTD | DTD Obtained by XTRACT | DTD obtained by DDbE |
|---|---|---|---|
| 1 | $a\|b\|c\|d\|e$ | $a\|b\|c\|d\|e$ | $a\|b\|c\|d\|e$ |
| 2 | $(a\|b\|c\|d\|e)^*$ | $(a\|b\|c\|d\|e)^*$ | $(a\|b\|c\|d\|e)^*$ |
| 3 | $(ab^*c^*)$ | $ab^*c^*$ | $(ab^+c^*)\|(ac^*)$ |
| 4 | $a^*b?c?d?$ | $a^*b?c?d?$ | $(a^+b(c\|(c?d))?)\|$ $((b\|a^+)?cd)\|((a^+\|b)?d)\|$ $((a^+\|b)?c)\|a^+\|b)$ |
| 5 | $(a(bc)^+d)*$ | $(a(bc)^*d)^*$ | $(a\|b\|c\|d)^+$ |
| 6 | $(ab?c^*d?)*$ | – | $(a\|b\|c\|d)+$ |

Table 3: DTDs generated for Real-life Data Set

of the form $1|a$ to $a?$. DTD 5 represents an interesting case where XTRACT is able to mine a DTD containing regular expressions containing nested $^*$'s. This is due to our generalization module that iteratively looks for sequencing patterns. On the other hand, DDbE simply over-generalizes DTD 5 by *or*-ing all the symbols in it and enclosing them within the metacharacter $^+$. Finally, neither XTRACT nor DDbE is able to correctly infer DTD 6. (The approximate DTD derived by XTRACT for DTD 6 is rather complex and, therefore, we chose to omit it from Table 3.) The reason for XTRACT's failure is that our generalization subsystem does not detect patterns containing the optional symbol ?. Finding such patterns requires a more sophisticated analysis of symbol occurrences within and across sequences, and we plan to pursue this further as part of our future work.

## 8 Conclusions

We have presented the architecture of the XTRACT system for inferring a DTD for a database of XML documents. The problem of automated DTD derivation is complicated by the fact that the DTD syntax incorporates the full expressive power of regular expressions. Specifically, as we have shown, naive approaches that do not "generalize" beyond the input element sequences fail to deduce concise and semantically meaningful DTDs. Instead, XTRACT applies sophisticated algorithms to compute a DTD that is more along the lines of what a human expert would infer. We compared the quality of the DTDs inferred by XTRACT with those returned by the IBM Alphaworks DDbE tool on synthetic and real-life DTDs. In our experiments, XTRACT outperformed DDbE by a wide margin; for most of our test cases, XTRACT was able to accurately infer the DTD whereas DDbE completely failed to do so. A number of the DTDs which were correctly identified by XTRACT were fairly complex and contained factors, metacharacters and nested regular expression terms. Thus, our results clearly demonstrate the effectiveness of the XTRACT approach that employs generalization and factorization to derive a range of general and concise candidate DTDs, and then uses the MDL principle as the basis to select amongst them.

## References


[1] S. Abiteboul. Querying semi-structured data. In *Proc. of the Intl. Conf. on Database Theory (ICDT)*, 1997.

[2] D. Angluin. On the complexity of minimum inference of regular sets. *Information and Control*, 39(3):337–350, 1978.

[3] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible markup language (XML). (`www.w3.org/TR/REC-xml`)

[4] R. K. Brayton and C. McMullen. The decomposition and factorization of boolean expressions. In *Proc of the Intl. Symp. on Circuits and Systems*, 1982.

[5] A. Brazma. Efficient identification of regular expressions from representative examples. In *Proc. of the Ann. Conf. on Computational Learning Theory (COLT)*, 1993.

[6] M. Charikar and S. Guha. Improved combinatorial algorithms for the facility location and k-median problems. In *Proc. of the Ann. Symp. on Foundations of Computer Science (FOCS)*, 1999.

[7] A. Deutsch, M. Fernandez, and D. Suciu. Storing semi-structured data with stored. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 1999.

[8] M. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *Proc. of the Intl. Conf. on Database Theory (ICDT)*, 1997.

[9] M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: A System for Extracting Document Type Descriptors from XML Documents. Bell Labs Tech. Memorandum, 1999.

[10] E. Mark Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[11] E. Mark Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, 1978.

[12] R. Goldman, J. McHugh, and J. Widom. From semistructured data to XML: Migrating the lore data model and query language. In *Proc. of the Intl. Workshop on the Web and Databases (WebDB)*, 1999.

[13] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proc. of the Intl. Conf. on Very Large Data Bases (VLDB)*, 1997.

[14] D. S. Hochbaum. Heuristics for the fixed cost median problem. *Mathematical Programming*, 22:148–162, 1982.

[15] J. E. Hopcroft and J. D. Ullman. *Introduction to Automaton Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts, 1979.

[16] P. Kilpeläinen, H. Mannila, and E. Ukkonen. MDL learning of unions of simple pattern languages from positive examples. In *Proc. of the European Conf. on Computational Learning Theory (EuroCOLT)*, 1995.

[17] M. Mehta, J. Rissanen, and R. Agrawal. MDL-based decision tree pruning. In *Proc. of the Intl. Conf. on Knowledge Discovery and Data Mining (KDD)*, 1995.

[18] S. Nestorov, S. Abiteboul, and R. Motwani. Extracting schema from semistructured data. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 1998.

[19] L. Pitt. "Inductive inference, DFAs, and computational complexity". *Analogical and Inductive Inference*, pp. 18–44, 1989.

[20] J. R. Quinlan and R. L. Rivest. Inferring Decision Trees Using the Minimum Description Length Principle. *Information and Computation*, 80:227–248, 1989.

[21] J. Rissanen. Modeling by shortest data description. *Automatica*, 14:465–471, 1978.

[22] J. Rissanen. *Stochastic Complexity in Statistical Inquiry*. World Scientific Publ. Co., 1989.

[23] J. Shanmugasundaram, G. He, K. Tufte, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proc. of the Intl. Conf. on Very Large Data Bases (VLDB)*, 1999.

[24] A. R. R. Wang. *Algorithms for Multi-level Logic Optimization*. PhD thesis, Univ. of California, Berkeley, 1989.

[25] J. Widom. Data management for XML: research directions. *IEEE Data Engineering Bulletin*, 1991.