XTRACT: Learning Document Type Descriptors from XML Document Collections

Minos Garofalakis
Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ 07974
minos@bell-labs.com

Aristides Gionis*
Dept. of Computer Science
Stanford University
Stanford, CA 94305
gionis@cs.stanford.edu

Rajeev Rastogi
Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ 07974
rastogi@bell-labs.com

S. Seshadri*
Strand Genomics
146, 5th Cross, RMV Ext.
Bangalore 560080, India
seshadri@strandgenomics.com

Kyuseok Shim†
SNU‡ and AITrc§
Kwanak P.O. Box 34
Seoul 151-742, Korea
shim@ee.snu.ac.kr

Abstract

XML is rapidly emerging as the new standard for data representation and exchange on the Web. Unlike HTML, tags in XML documents describe the semantics of the data and not how it is to be displayed. In addition, 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. Despite their importance, however, DTDs are *not mandatory*, and it is frequently possible that documents in XML databases will not have accompanying DTDs. 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.

*This work was done while the author was with Bell Laboratories.
†**CONTACT AUTHOR.** Phone: +82-2-880-7269 , Fax: +82-2-871-5974.
‡Seoul National University.
§Advanced Information Technology Research Center.

In many respects, XML data is an instance of *semistructured data* [Abi97]. 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 [SHT$^+$99], 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 [DFS99], 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, [GW97, FS97]). 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 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., [GMW99, Wid99]) 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. Note that, even though some simple structural descriptions or typings (e.g., [NAM98, GW97]) of the generated XML data can be made available, such structural information typically does not incorporate the expressive and semantic power of regular expressions and, as a consequence, is of limited use as a concise and meaningful data schema.

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. In contrast to simple structural models (e.g., [NAM98, GW97]), the DTD syntax incorporates the full specification power of regular expressions; thus, 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, users or domain experts looking for a meaningful description of their XML data can use the DTD description returned by such tools as a starting point from which more refined schemas can be generated. As another application, consider an employment web site that integrates information on job openings from thousands of different web sources 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. Further, a strategy that simply tries to consolidate the various DTDs based on simple heuristic rules could easily fail to produce a concise and meaningful DTD for the integrated collection, especially if there is sufficient variation in these per-source DTDs. Under such a scenario, 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 intuitive 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. Finally, 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 (*) [HU79].) 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 [BM82, Wan89].

- **Minimum Description Length (MDL) Principle.** In the final and most important step, the XTRACT system employs Rissanen's *Minimum Description Length* (MDL) principle [Ris78, Ris89] 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 [QR89]. 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 $\mathcal{NP}$-complete [Hoc82]. Fortunately, efficient approximation algorithms with guaranteed performance ratios have been proposed for the FLP in the literature [CG99], 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[1] DDbE (Data Descriptors by Example) DTD extraction tool, 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 and none of them have stabilized – therefore, we do not concentrate on these extensions in this paper.

Roadmap. The remainder of the paper is organized as follows. After discussing related work in Section 2, we present an overview of our approach to inferring DTDs in Section 3. Section 4 describes how the MDL principle is employed within XTRACT to compose a "good" DTD from an input set of candidate DTDs. In Sections 5 and 6,

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

we present generalization and factoring algorithms for producing candidate DTDs that are input to the MDL module of XTRACT. Section 7 discusses the results of our experiments with real-life and synthetic DTDs. Finally, we offer concluding remarks in Section 8.

2 Related Work

The XTRACT approach of generating intuitive, semantically-meaningful DTDs based on the information-theoretic MDL principle is novel and has not been previously explored in the database or machine-learning 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).

The problem of extracting a schema from semistructured data has been addressed in [NAM98, GW97, FS97]. Although, XML can be viewed as an instance of semistructured data, the kinds of schema considered in [NAM98, GW97, FS97] are very different from a DTD. The schema extracted by [NAM98, GW97, FS97] 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.

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 *DFA*s (deterministic finite automata) [Gol67, Gol78, Ang78] (see also [Pit89] for a detailed survey of the topic). The above line of work is purely theoretical and it 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 [Bra93] which addressed the problem of approximating *roughly equivalent* regular expressions from a long enough string, and the work in [KMU95] where the *MDL* principle was used to infer a *pattern language* from positive examples. However, the problem tackled in [KMU95] is much simpler than ours since they assume that the set of simple patterns whose subset is to be computed is available. Furthermore, the patterns they consider are simple sequences that are permitted to contain single symbol wildcards. In our problem setting, unlike [KMU95], patterns are general regular expressions and are not known apriori. Ahonen et al. [AMN94, Aho96] propose an approach for automatically generating context-free grammars from structured text documents. Their method essentially produces a "union" finite-state automaton for all example documents and then simplifies/generalizes that automaton (and the corresponding regular expression) by merging states to guarantee a *(k,h)-contextuality* requirement (an extension that they propose to k-contextual regular languages). A potential problem with this approach is that the amount of generalization introduced depends critically on the values of the (k, h) parameters, and the result-

```
<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

ing regular expressions may need to be manipulated further in order to produce meaningful structural descriptions; to address this, Ahonen et al. [AMN94] suggest the use of interactive operations based on certain ad-hoc rules that allow users to interactively control the amount of generalization injected during the inference process (e.g., using frequency information from the example set). Young-Lai and Tompa [YLT00] propose a more systematic approach based on *stochastic* grammatical inference that, basically, takes frequency information directly into account during generalization. Briefly, their method makes use of frequencies of automaton paths in parameterized statistical tests in order to determine the states that should be merged. To ensure the validity of these tests, they introduce an additional statistical test to identify low-frequency components in the automaton and suggest different heuristics for dealing with such components. Unfortunately, the effectiveness of this method is, once again, critically dependent on the choice of values for the different statistical test parameters and “good” choices are not at all obvious; further, dealing with low-frequency paths needs to rely on ad-hoc rules. In contrast, our XTRACT approach does not rely on parameterized tests but solely on the solid, information-theoretic foundation of the MDL principle in order to infer accurate and meaningful DTD schemas.

3 Problem Formulation and Overview of our Approach

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. But first, we present a brief overview of XML and DTDs in the following subsection to make the subsequent discussion concrete.

3.1 Overview of XML and DTDs

An XML document, like an HTML document, consists of nested element structures starting with a root element. Subelements of an element can either be elements or simply character data. 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. The `title` element contains character data denoting the title of the article while the `name` element contains the name of the author of the article. The ordering of subelements within an element is significant in XML. Elements can also have zero or more attribute/value pairs that are stored within the element’s start tag. More details on the XML specification can be found in [BPSM].

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 2 illustrates a

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

Figure 2: An Example DTD

DTD that the XML document in Figure 1 conforms to. The DTD declaration syntax uses commas for sequencing, | for (exclusive) OR, parenthesis for grouping and the meta-characters ?,* ,$^+$ to denote, respectively, zero or one, zero or more, and one or more occurrences of the preceding term. As a special case, the DTD corresponding to an element can be ANY which allows an arbitrary XML fragment to be nested within the element. The DTD can also be used to specify the attributes for an element (using the `<!ATTLIST >` declaration) and to declare an attribute that refers to another element (via an IDREF field). We must point out that real-life DTDs can get fairly complex and can sometimes contain several regular expressions terms with multiple levels of nesting (e.g., $((ab)^*c)^*$). We present examples of real-life DTDs in Sections 5 and 7.

For brevity, in the remainder of the paper, we denote elements of an XML document by a single letter from the lower case alphabet. Also, we do not include explicit commas in element sequences and regular expressions since they can be inferred in a straightforward fashion.

3.2 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 XML documents, 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 rest of the paper, we concentrate on the problem of extracting a DTD for a single element. In this paper, 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 the DTD. It is straightforward to compute the sequence of subelements nested within each $< e >< /e >$ pair in the XML documents. 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. $\Box$

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 that the obtained regular expression, however, 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 lot 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 being 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 as and bs, 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 input sequences without losing too much information about the structure of the input 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, exact solutions, i.e., regular expressions whose language is *exactly* 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, there is a problem with devising a solution based on the above restrictions. The problem is that restrictions R1 and R2 conflict with each other. In our earlier example, restriction R1 would favor DTDs (1) and (3), while these DTDs would not be considered good according to criterion R2. The situation is exactly the reverse when we consider DTDs (2) and (4). Thus, in general, there is a tradeoff between a DTD's "conciseness" and it's "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 the two qualitative notions.

3.3 Using the MDL Principle to Define a Good DTD

We use the MDL principle [Ris78, Ris89] 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 [QR89, MRA95] to learning common patterns in sets of strings [KMU95].

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

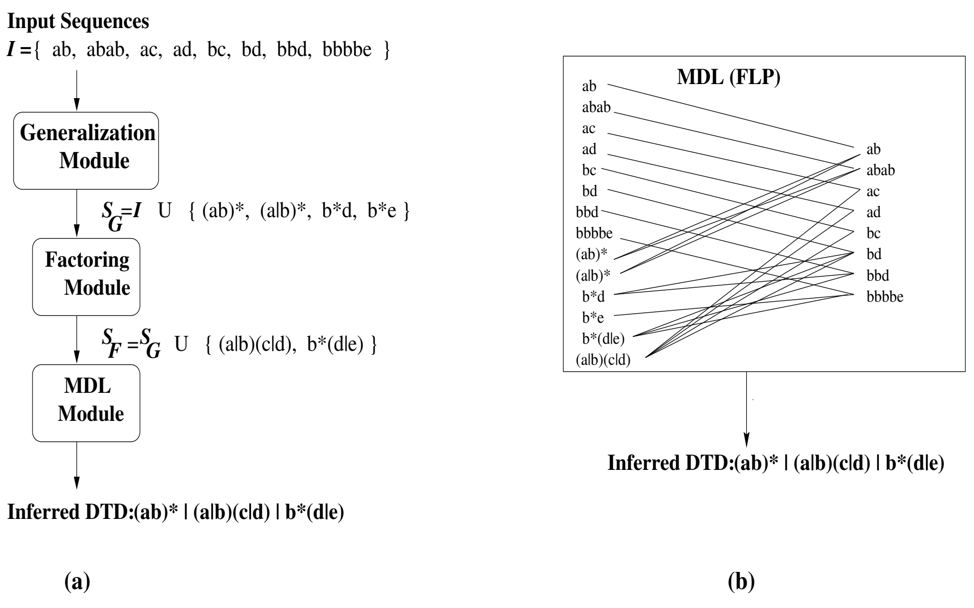
(B) the length of the data, in bits, when encoded with the help of the theory.

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, and in a manner that is consistent with our intuition. By favoring concise and precise DTDs, and penalizing those that are not, it ranks highly exactly those DTDs that would be deemed desirable by humans.

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 MDL 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 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



Input Sequences
I ={ ab, abab, ac, ad, bc, bd, bbd, bbbbe }
Generalization Module
S_G=I U { (ab)*, (a|b)*, b*d, b*e }
Factoring Module
S_F =S_G U { (a|b)(c|d), b*(d|e) }
MDL Module
Inferred DTD:(ab)* | (a|b)(c|d) | b*(d|e)
(a)
MDL (FLP)
ab
abab
ac
ad
bc
bd
bbd
bbbbe
(ab)*
(a|b)*
b*d
b*e
b*(d|e)
(a|b)(c|d)
Inferred DTD:(ab)* | (a|b)(c|d) | b*(d|e)
(b)

Figure 3: Architecture of the XTRACT System

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. □

From the above example, it follows that the MDL principle indeed provides an elegant mechanism for quantifying and resolving the tradeoff between the conciseness and preciseness properties of DTDs. Specifically,

1. Part (A) of the MDL cost includes the number of bits required to encode the DTD – this ensures that the inferred DTD is succinct.

2. Part (B) of the MDL cost includes the number of bits needed for encoding the input sequences using the DTD. Usually, expressing data in terms of a more general DTD (e.g., $(a \mid b)^*$ in Example 3.2) requires more bits than describing data in terms of a more specific DTD (e.g., $(ab)^*$ in Example 3.2). As a result, using the MDL principle ensures that the DTD we choose is a fairly tight characterization of the data.

The MDL principle, thus, enables us to choose a DTD that strikes the right balance between conciseness and preciseness.

3.4 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 [2]. 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 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 to be 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[3], 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 [BM82, Wan89] to meet our needs. However, even though every subset of candidate DTDs can, in principle, be factored, the number of these subsets can be large 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).

[2]Note that our result DTDs are not necessarily *deterministic*, in the sense that the DTD can, at some points, have more than one valid matches for elements in a conforming XML document. For XML parsers that have such determinism requirements, standard constructions for removing non-determinism [HU79] can be applied to the DTD output by XTRACT.

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

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 [Hoc82, CG99]. We adapt the algorithm from [CG99] 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 the 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 of

(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.

Thus, 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 the 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 Σ 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 in $\Sigma \cup \mathcal{M}$, be n. Then, the length of the DTD in bits is $n \log(\mid \Sigma \mid + \mid \mathcal{M} \mid)$. As an example, let Σ consist of the elements a and b. The length in bits of the DTD a^*b^* is $4 * \log(2+6) = 12$. 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). The 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 ϵ.

2. The encoding for the sequence b in terms of the DTD $a \mid b \mid c$ 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

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

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

(C) $seq(D_1 | \dots | D_m, s) = i \; seq(D_i, s)$ that is, D is the exclusive choice of regular expressions $D_1 \dots D_m$, 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) $seq(D^*, s_1 \dots s_k) = \begin{cases} k \; seq(D, s_1) \dots seq(D, s_k) & \text{if } k > 0 \\ 0 & \text{otherwise} \end{cases}$

In other words, the sequence $s = s_1 \dots s_k$ 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

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 the 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 [HU79] 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)* [Hoc82, CG99]. 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

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

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 i using the DTD corresponding to the facility j. If a DTD j does not cover a sequence i, then we set $d(j,i)$ to ∞. Thus, the set F computed by the FLP corresponds to our desired set $\mathcal{S}$ of candidate DTDs.

The FLP is $\mathcal{NP}$-hard; however, it can be reduced to the *set cover problem* and then approximated within a logarithmic factor as shown in [Hoc82]. In our implementation, we employ the randomized algorithm from [CG99] 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 [CG99] for our problem setting to be very good in practice. Furthermore, the time complexity of [CG99] 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. Below, we present examples of such regular expressions from real-life DTDs that appear in the Newspaper Association of America (NAA) Classified Advertizing Standards XML DTD[4].

a^*bc^*: DTDs of this form are generally used to specify tuples with set-valued attributes.

```
<!ELEMENT account-info (account-number, sub-account-number*)> <!--
Specification for account identification information -->
```

$(abc)^*$: This type of DTD is used to represent a set (or a list) of ordered tuples.

```
<!ELEMENT days-and-hours (date, time)+> <!-- provide times/dates
when job fairs will be held -->
```

$(a|b|c)^*$: The DTD of the form $(a|b|c)*$ is frequently used to represent a multiset containing the elements a, b and c. This DTD is very useful since the elements in the multiset are allowed to appear multiple times and in any order in the document. For example, the following DTD specifies that the support information for an ad can consist of an arbitrary number of audio or video clips, photos, and further these can appear in any order.

```
<!ELEMENT support-info (audio-clip | file-id | graphic | logo |
new-list | photo | video-clip | zz-generic-tag)*> <!-- support
information for ad content -->
```

$((ab)^*c)^*$: This type of DTD permits nesting relationships among sets (or lists).

```
<!ELEMENT transfer-info (transfer-number, (from-to, company-id)+,
contact-info)*> <!-- provides parent information through the multi-
level aggregation process. may be repeated -->
```

Although our XTRACT algorithms can infer regular expressions that are more complex than the above, there are certain complex patterns (e.g., patterns containing the optional symbol (?) nested within Kleene stars) that are not explicitly detected by XTRACT's generalization subsystem. Such complex expressions are probably less likely to occur in practice; for example, in the representative set of real-life DTDs used in our experiments (Section 7) there was only one such instance, namely $(ab?c^*d?)*$. Further, we believe that our XTRACT generalization framework can be appropriately extended with more sophisticated sequence-analysis heuristics to effectively deal with such "difficult" scenarios.

[4]These can be accessed at `http://www.naa.org/technology/clsstdtf/Adex010.dtd`.

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), but however, our MDL step will ensure that such over-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 be considered by the MDL step.

The essence of procedure GENERALIZE are the procedures DISCOVERSEQPATTERN and DISCOVERORPATTERN which are repeatedly called with different values for their input parameters r and d. These parameter values basically control the amount of generalization injected by XTRACT in the discovered candidate patterns, and using several different combinations allows the generalization subsystem to produce a richer collection of candidates for the final MDL-based pattern selection (that will weed out overly general or overly restrictive candidates). The specific parameter values used in the pseudo-code of Figure 5 are ones that we found to perform reasonably well in practice without overloading the set of potential candidates. We discuss the details of our two generalization procedures and the exact roles of their input 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 the input to them is always a 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 $s = ababab cababc$ 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 symbol corresponding to $((ab)^*c)^*$. Thus, the resulting candidate DTD returned by procedure DISCOVERSEQPATTERN can contain Kleene stars nested within other stars. 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

```
procedure GENERALIZE($I$)
begin
1.  for each sequence $s$ in $I$
2.      add $s$ to $\mathcal{S}_\mathcal{G}$
3.      for $r := 2, 3, 4$
4.          $s' :=$ DISCOVERSEQPATTERN($s, r$)
5.          for $d := 0.1 \cdot |s'|, 0.5 \cdot |s'|, |s'|$
6.              $s'' :=$ DISCOVERORPATTERN($s', d$)
7.              add $s''$ to $\mathcal{S}_\mathcal{G}$
end

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

procedure DISCOVERORPATTERN($s$, $d$)
begin
1.  $s_1, s_2, \ldots, s_n :=$ PARTITION($s, d$)
2.      for each subsequence $s_j$ in $s_1, s_2, \ldots, s_n$
3.          let the set of distinct symbols in $s_j$ be $a_1, a_2, \ldots, a_m$
4.          if $(m > 1)$
5.              replace subsequence $s_j$ in sequence $s$ by a new auxiliary symbol $A_i = (a_1|\cdots|a_m)^*$
6.  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 the right of $s_i$ within a distance $l \leq d$)
5.          $end := end + l$; $s_i := s[start, end]$
6.      if $(end < |s|)$
7.          $i := i + 1; start := end + 1; end := end + 1; s_i := s[start, end]$
8.  return $s_1, s_2, \cdots, s_i$
end
```

Figure 5: The Generalization Algorithm

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 $(a_1|a_2|\dots|a_m)^*$ based on the locality of these symbols within a sequence s. It finds out such locality by first partitioning (performed by procedure PARTITION) the input sequence s into the smallest possible subsequences $s_1, s_2, \dots, s_n$, 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 $(a_1|a_2|\dots|a_m)^*$ where $a_1, \dots, a_m$ are the distinct symbols in the subsequence s_i. The intuition here is that if s_i contains frequent repetitions of the symbols $a_1, a_2, \dots, a_m$ in close proximity, then it is very likely that s_i originated from a regular expression of the form $(a_1|a_2|\dots|a_m)^*$. As an illustration, on the input sequence *abcbac*, procedure DISCOVEROR-PATTERN 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 the subsequences 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 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 that are an *OR* of a subset of the candidate DTDs in $\mathcal{S}_G$. For example, for candidate DTDs *ac*, *ad*, *bc* and *bd* in $\mathcal{S}_G$, 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. Thus, the DTD (in factored form) for the element `article` would then be as follows:

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

The set of candidate DTDs output by the factorization module, $\mathcal{S}_{\mathcal{F}}$, in addition to the factored forms generated from candidates in $\mathcal{S}_{\mathcal{G}}$, 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 pretty large. Therefore, in the following subsection, we propose a heuristic for selecting sets of candidates in $\mathcal{S}_{\mathcal{G}}$ that when factored yield good factored DTDs. We then present a brief description of the factoring algorithm itself, which is an adaptation of factoring algorithms for boolean expressions from the logic optimization literature.

Note that each candidate DTD in $\mathcal{S}_{\mathcal{G}}$ is a sequence of symbols, some of which can be auxiliary symbols. Recall that auxiliary symbols translate to regular expressions on symbols in Σ, and there is a one-to-one correspondence between auxiliary symbols and the expressions that they represent.

6.1 Selecting Subsets of $\mathcal{S}_{\mathcal{G}}$ to Factor

In this section, we describe how we choose subsets of $\mathcal{S}_{\mathcal{G}}$ that lead to good factorizations. Intuitively, a subset S of $\mathcal{S}_{\mathcal{G}}$ is a good candidate for factoring if the factored form of S is much smaller than S itself. In addition, even though $\mathcal{S}_{\mathcal{G}}$ may contain multiple generalizations that are derived from the same input sequence, it is highly unlikely that the final DTD will contain two generalizations of the same input sequence. Thus, factoring candidate DTDs in $\mathcal{S}_{\mathcal{G}}$ that cover similar sets of input sequences does not lead to factors that can improve the quality of the final DTD.

We thus conclude that if a subset S of $\mathcal{S}_{\mathcal{G}}$ to yield good factored forms it must satisfy the following two properties:

1. Every DTD in S has a common prefix or suffix with a number of other DTDs in S. Further, as more DTDs in S share common prefixes or suffixes, or as the length of the common prefixes/suffixes increases, the quality of the generated factored form can be expected to improve.

2. The *overlap* between every pair of DTDs D, D' in S is minimal, that is, the intersection of the input sequences covered by D and D' is small. This is important because, as mentioned above, a factored DTD adds little value (from an MDL cost perspective) over the candidate DTDs from which it was derived if it cannot be used to encode a significantly larger number of input sequences compared to the sequences covered by each individual DTD.

Definitions. In order to state properties (1) and (2) for a set S of DTDs more formally, we need to first define the following notation. For a DTD D, let $\mathit{cover}(D)$ denote the input sequences in I that are covered by D

(note that auxiliary symbols are expanded completely when $cover$ for a DTD is computed). Then, $overlap(D, D')$ is defined as the fraction of the input sequences covered by D and D' that are common to D and D', that is, $overlap(D, D') = \frac{|cover(D) \cap cover(D')|}{|cover(D) \cup cover(D')|}$. Thus, for a sufficiently small value of the (user-specified) parameter δ, by ensuring that $overlap(D, D') < \delta$ for every pair of DTDs D and D' in S, we can ensure that S satisfies Property (2) mentioned above.

In order to characterize Property (1) more rigorously, we introduce the function $score(D, S)$ which attempts to capture the degree of similarity between prefixes/suffixes of DTD D and those of DTDs in the set S of DTDs. Intuitively, a DTD with a high $score$ with respect to set S is a good candidate to be factored with other DTDs in set S. For a DTD D, let $pref(D)$ and $suf(D)$ denote the set of prefixes and suffixes of D, respectively. Let $psup(p, S)$ denote the support of prefix p in set S of DTDs, that is, the number of DTDs in S for which p is a prefix. Similarly, let $ssup(s, S)$ denote number of DTDs in S for which s is a suffix. Then $score(D, S)$ is defined as follows.

$$score(D, S) = \max(\{|p| \cdot psup(p, S) : p \in pref(D)\} \cup \{|s| * ssup(s, S) : s \in suf(D)\})$$

Thus, the prefix/suffix p/s of D, for which the product of p/s's length and its support in S is maximum, determines the score of D with respect to S. The intuition here is that if DTD D has a long prefix or suffix that occurs frequently in set S, then this prefix can be factored out thus resulting in good factored forms. The function $score$ is thus a good measure of how well D would factor with other DTDs in S.

Algorithm. Procedure FACTORSUBSETS, shown in Figure 6, first selects subsets S of $\mathcal{S}_\mathcal{G}$ to factor that satisfy properties (1) and (2) mentioned earlier. Each of these subsets S is then factored by invoking procedure FACTOR (in Step 15) described in the next subsection. Assuming that the factoring algorithm returns $F_1 \mid F_2 \mid \cdots F_m$, each of the F_i is added to $\mathcal{S}_\mathcal{F}$ that is then input to the MDL module.

We now discuss how procedure FACTORSUBSETS computes the set S of candidate DTDs to factor. First, k seed DTDs for the sets S to be factored are chosen in the for loop spanning Steps 4–7. These seed DTDs have a high $score$ value with respect to $\mathcal{S}_\mathcal{G}$ and overlap minimally with each other. Thus, we ensure that each seed DTD not only factors well with other DTDs in $\mathcal{S}_\mathcal{G}$, but is also significantly different from other seeds. In Steps 9–14, each seed DTD is used to construct a new set S of DTDs to be factored (thus, only k sets of DTDs are generated). After initializing S to a seed DTD D, in each subsequent iteration, the next DTD D' that is added to S is chosen greedily – it is the one whose score with respect to DTDs in S is maximum and whose overlap with DTDs already in S is less than δ.

Complexity Results. The time complexity of selecting the sets S to factor in the FACTORSUBSETS procedure can be shown to be $O(N^2 \cdot (N + L))$, where $N = |I|$ and L is the maximum length of an input sequence in I. The reason for this is that the initial computation of $score(D, \mathcal{S}_\mathcal{G})$ for every DTD D in $\mathcal{S}_\mathcal{G}$ requires us to compute the support of every prefix and suffix of D in $\mathcal{S}_\mathcal{G}$. Since $\mathcal{S}_\mathcal{G}$ contains $O(N)$ DTDs, and each DTD can have at most $2L$ prefixes/suffixes, there are at most $O(N \cdot L)$ distinct prefixes and suffixes. The supports for these can be computed in $O(N \cdot L)$ steps by storing them in a trie structure. Thus, the time complexity of computing the scores for all the DTDs in $\mathcal{S}_\mathcal{G}$ (in Steps 1–2) is $O(N \cdot L)$.

Computing the overlap between a pair of DTDs requires $O(N)$ time to compute the intersection and union of the input sequences they cover. Thus, the worst-case time complexity to compute the overlap between all pairs of DTDs in $\mathcal{S}_\mathcal{G}$ is $O(N^3)$. Assuming that we precompute the overlapping DTD pairs in $\mathcal{S}_\mathcal{G}$, SeedSet can be computed in $O(N)$

```
procedure FACTORSUBSETS($\mathcal{S_G}$)
begin
1.  for each DTD $D$ in $\mathcal{S_G}$
2.          Compute $score(D,\mathcal{S_G})$
3.  $\mathcal{S_F}$:= $S'$ := $\mathcal{S_G}$; SeedSet := $\emptyset$
4.  for $i$ := 1 to $k$
5.          let $D$ be the DTD in $S'$ with the maximum value for $score(D,\mathcal{S_G})$
6.          SeedSet := SeedSet $\cup D$
7.          $S' := S' - \{D' : overlap(D, D') \geq \delta\}$
8.  for each DTD $D$ in SeedSet
9.          $S := \{D\}$
10.         $S' := \mathcal{S_G} - \{D' : overlap(D, D') \geq \delta\}$
11.         while ($S'$ is not empty)
12.                 let $D'$ be the DTD in $S'$ with the maximum value for $score(D', S)$
13.                 $S := S \cup D'$
14.                 $S' := S' - \{D'' : overlap(D', D'') \geq \delta\}$
15.         $F$ := FACTOR($S$)
16.         $\mathcal{S_F}$:= $\mathcal{S_F} \cup \{F_1, \ldots, F_m\}$        /* $F = F_1 \mid \cdots \mid F_m$ */
end
```

Figure 6: Choosing Subsets Of $\mathcal{S_G}$ For Factoring

steps (since the number of seeds, k, is a constant). Furthermore, the time complexity of computing each set S of DTDs to be factored can be shown to be $O(N^2 \cdot L)$ since the while loop (Steps 11–14) performs at most $O(N)$ iterations and the cost of recomputing the scores for DTDs in S' (with respect to S) in each iteration is $O(N \cdot L)$ (as before, this can be achieved by maintaining a trie structure for prefixes and suffixes of DTDs in S).

6.2 Algorithm For Factoring a Set of DTDs

In this section, we show how the factored form for a set S of DTDs can be derived – the expression we factor is actually the *OR* of the DTDs in S. Algorithms for computing the optimum factored form, that is, the one with the minimum number of literals have been proposed earlier in [Law64]. However, the complexity of these exact techniques are impractical for all but the smallest expressions. Fortunately, however, there exist heuristic factoring algorithms for boolean functions [Wan89] which work very well in practice. Factored forms of boolean functions are very useful in VLSI design, since in most design styles such as complex-gate CMOS design, the implementation of a function directly corresponds to a factored form, and thus factored forms are useful in estimating area and delay in a multi-level logic synthesis and optimization system.

We adapt the technique for factoring boolean expressions from [Wan89] for our purposes. There is a close correspondence between the semantics of DTDs and those of boolean expressions. The sequencing operator in DTDs is similar to a logical AND in boolean algebra, while the *OR* operator (|) is like a logical OR. However, there exist

```
procedure FACTOR($S$)          /* $S$ is the set of sequences to be factored */
begin
1. DivisorSet := FINDALLDIVISORS($S$)
2. if (DivisorSet = $\phi$)
3.         return OR of sequences in $S$
4. DivisorList := $\phi$
5. for each divisor $V$ in DivisorSet
6.         $Q, R$ := DIVIDE($S, V$)
7.         add ($V, Q, R$) to DivisorList
8. find the most compact triplet $(V_i, Q_i, R_i)$ in DivisorList
9. return (FACTOR($V_i$))(FACTOR($Q_i$)) | FACTOR($R_i$)
end

procedure FINDALLDIVISORS($S$)
begin
1. DivisorSet := $\phi$
2. for each distinct sequence $s$ such that $s$ is a suffix for at least two elements in $S$
3.         DivisorSet := DivisorSet $\cup$ $\{\{p : ps \in S\}\}$
4. return DivisorSet
end

procedure DIVIDE($S, V$)
begin
1. for each sequence $p$ in $V$
2.         $q_p := \{s : ps \in S\}$
3. $Q := \bigcap_{p \in V} q_p$
4. $R := S - V \circ Q$
   /* $V \circ Q$ is the set of sequences resulting from concatenating
      every sequence in $Q$ to the end of every sequence in $V$ */
5. return $Q, R$
end
```

Figure 7: Factoring Algorithm

certain fundamental differences between DTDs and boolean expressions. First, while the logical AND operator in boolean logic is commutative, the sequencing operator in DTDs is not (the ordering of symbols in a sequence matters!). Second, in boolean logic, the expression $a \mid ab$ is equivalent to a; however, the equivalent DTD for $a \mid ab$ is $ab?$. Our factoring algorithm makes appropriate modifications to the algorithm from [Wan89] to handle the above-mentioned differences between the semantics of DTDs and boolean algebra. The details of our factoring procedure (termed FACTOR) can be found in the pseudo-code depicted in Figure 7. The following example illustrates the key steps of our factoring algorithm.

Example 6.1 Consider the set $S = \{b, c, ab, ac, df, dg, ef, eg\}$ of input sequences corresponding to the expression $b|c|ab|ac|df|dg|ef|eg$ whose factored form is $a?(b|c)|(d|e)(f|g)$. Before we present the steps that FACTOR performs to derive the factored form, we introduce the DIVIDE operation that constitutes the core of our algorithm. For sets of sequences S, V, DIVIDE(S, V) returns a quotient Q and remainder R such that $S = V \circ Q \cup R$ (here, $V \circ Q$ is the set of sequences resulting from concatenating every sequence in Q to the end of every sequence in V). Thus, for the above set S and $V = \{d, e\}$, DIVIDE(S, V) returns the quotient $Q = \{f, g\}$ and remainder $R = \{b, c, ab, ac\}$. We are now in a position to describe the steps executed by FACTOR to generate the factored form.

1. **Compute set of potential divisors for S.** These are simply sets of prefixes that have a common suffix in S. Thus, potential divisors for S include $\{d, e\}$ (both f and g are common suffixes) and $\{1, a\}$ (both b and c are common suffixes). The symbol "1" is special and denotes the identity symbol with respect to the sequencing operator, that is, $1s = s1 = s$ for every sequence s.

2. **Choose divisor V from set of potential divisors.** This is carried out by first dividing S by each potential divisor V to obtain a quotient Q and remainder R, and then selecting the V for which the triplet (V, Q, R) has the smallest size. In our case, $V = \{d, e\}$ results in a smaller quotient and remainder ($Q = \{f, g\}$, $R = \{b, c, ab, ac\}$) than $\{1, a\}$ ($Q = \{b, c\}$, $R = \{df, dg, ef, eg\}$) and is thus chosen.

3. **Recursively factor V, Q and R.** The final factored form is FACTOR(V)FACTOR(Q)|FACTOR(R), where $V = \{d, e\}$, $Q = \{f, g\}$ and $R = \{b, c, ab, ac\}$. Here, V and Q cannot be factored further since they have no divisors. Thus, FACTOR(V) = $(d|e)$ and FACTOR(Q) = $(f|g)$. However, R can be factored more since $\{1, a\}$ is a divisor. Thus, repeating the above steps on R, we obtain FACTOR(R) = $(1|a)(b|c)$. Thus, the final factored form is $(1|a)(b|c)|(d|e)(f|g)$.

4. **Simplify final expression by eliminating "1".** The term $(1|a)$ in the final expression can be further simplified to $a?$. Thus, we obtain the desired factored form for S.

$\square$

7 Experimental Study

In order 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 as well as real-life DTDs. We also compared the DTDs produced by XTRACT with those generated by the IBM alphaworks DTD extraction tool, DDbE[5] (Data Description by Example),

[5]The DDbE tool and a detailed description of it is available at `http://www.alphaworks.ibm.com/`.

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 the 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 δ is set to 0 in the procedure FACTORSUBSETS. Finally, in the MDL step, we employ the algorithm from [CG99] to compute an approximation to the FLP.

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 which permit the user to control the structure of the content models and the types used for attribute declarations. Some of the important parameters of DDbE that we used in our experiments, along with their default values, are presented in Table 1.

Parameter	Meaning	Default
c	Maximum number of consecutive identical tokens not replaced by a list	1
d	Maximum depth of factorization	2

Table 1: Description of Parameters Used by DDbE

The parameter c specifies the maximum number of consecutive identical tokens that should not be replaced by a list. For example, the default value of this parameter is 1 and thus all sequences containing two or more repetitions of the same symbol are replaced with a positive list. That is, aa is substituted by $a+$. The parameter d determines the number of applications of factoring. For a set of input sequences that conform to the DTD of $a(b|c|d)(e|f|g)h$, for increasing values of the parameter d, DDbE returns the DTDs in Table 2.

Parameter Value (d)	DTD Obtained
1	$(acg|ace|adf|abg|abe|acf|adg|ade|abf)h$
2	$a(cg|ce|df|bg|be|cf|dg|de|bf)h$
3	$a((c|b|d)g|(d|c|b)f|(c|b|d)e)h$
4	$a((c|b|d)g|(d|c|b)f|(c|b|d)e)h$

Table 2: DTDs generated by DDbE for Increasing Values of Parameter d

As shown in the table, for $d = 1$, factorization is performed once in which the rightmost symbol h is factored out. When the value of d becomes 2, the leftmost symbol a is also factored out. A further increase in the value of d to 3 causes factorization to be performed on the middle portion of the expression and the common expression $(b|c|d)$ to be extracted. However, note that subsequent increases in the value of d (beyond 3) do not result in further changes to the DTD. This seems to be a limitation of DDbE's factoring algorithm since examining the DTD for $d = 3$, we can easily notice that e, f and g have a common factor of $(b|c|d)$ with different placement of the symbols within the parenthesis. However, the current version of DDbE cannot factorize this further.

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 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 where n is randomly chosen number 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) $(a_1|a_2|a_3|a_4|\ldots|a_{n_i})^*$ (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 3, 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. Note that we use letters from the alphabet as subelement symbols.

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 3: Synthetic DTD Data Set

The ten synthetic DTDs vary in complexity with later DTDs being more complex than the 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[6]. We examined this real-life DTD data and collected six representative DTDs that are shown in Table 4. Of the DTDs shown in the table, the last three DTDs are quite interesting. 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.

No.	Original DTD	Simplified DTD
1	< !ENTITY % included-elements ”audio-clip|blind-box-reply|graphic|linkpi-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$*$, geographic-area?, city?, state-province?, postal-code?, country?” >	$a^*b?c?d?$
5	< !ELEMENT transfer-info(transfer-number, (from-to, company-id)$^+$, contact-info)$*$ >	$(a(bc)^+d)^*$
6	< !ELEMENT real-estate-services(real-estate-service.type, real-estate-service.location?, r-e.response-modes$*$, r-e.comment?)$*$ >	$(ab^?c^*d?)^*$

Table 4: Real-life DTD Data Set

7.3 Quality of Inferred DTDs

Synthetic DTD Data Set. The DTDs inferred by XTRACT and DDbE for the synthetic data set are presented in Table 5. As shown in the table, XTRACT infers each of the original DTDs correctly. In contrast, DDbE computes the accurate DTD for only 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. For instance, one of the input sequences covered by DTD 2 is gh which is not covered by the DTD inferred by DDbE. Thus, while XTRACT infers a DTD that covers all the input sequences, the DTD returned by DDbE may not cover every input sequence. DTD 4 exemplifies the two typical behaviors of DDbE – (1) sequence f that is not frequently repeated is appended to both the front and the back of the final DTD, and (2) symbols that are repeated frequently are all *OR*'d together and encapsulated by the metacharacter $^+$. For example, DDbE incorrectly identifies the term $(abcde)^*$ to be $(a|b|c|d|e)^*$ which is much more general. Thus, the DDbE tool has a tendency to over-generalize when the original DTDs contain regular expressions with Kleene stars. This same trend to over-generalize can be seen in DTDs 8–10 also. On the other hand, as is evident from Table 3, this is not the case for XTRACT which correctly infers every one of the original DTDs even for the more complex DTDs 8–10

[6]This can be accessed at `http://www.naa.org/technology/clsstdtf/Adex010.dtd`.

No.	Original DTD	DTD Inferred by XTRACT	DTD Inferred by DDbE
1	$abcde\vert efgh\vert ij\vert klm$	$abcde\vert efgh\vert ij\vert klm$	$abcde\vert efgh\vert ij\vert klm$
2	$(a\vert b\vert c\vert d\vert f)^*gh$	$(a\vert b\vert c\vert d\vert f)^*gh$	$gh(a\vert b\vert c\vert d\vert f)^+gh$
3	$(a\vert b\vert c\vert d)^*\vert e$	$(a\vert b\vert c\vert d)^*\vert e$	$(e(a\vert c\vert d\vert b)^+e)$
4	$(abcde)^*f$	$(abcde)^*f$	$(f(a\vert e\vert d\vert c\vert b)^+f)$
5	$(ab)^*\vert cdef\vert (ghi)^*$	$(ab)^*\vert cdef\vert (ghi)^*$	$cdef(a\vert b\vert g\vert i\vert h)^+cdef$
6	$abcdef(g\vert h\vert i\vert j)(k\vert l\vert m\vert n\vert o)$	$abcdef(g\vert h\vert i\vert j)(k\vert l\vert m\vert n\vert o)$	$abcdef(j(o\vert l\vert m\vert n\vert k)\vert g(o\vert l\vert n\vert m\vert k)\vert$ $h(m\vert l\vert n\vert k\vert o)\vert i(o\vert l\vert n\vert m\vert k))$
7	$(a\vert b\vert c)d^*e^*(fgh)^*$	$(a\vert b\vert c)d^*e^*(fgh)^*$	$((c\vert b\vert a)d^+e^+\vert ad^+\vert bd^+\vert c(e^+\vert d^+)?\vert$ $ad^*\vert be^*))(f\vert h\vert g)^+((a\vert b\vert c)d^+e^+\vert$ $c(e^+\vert d^+)?\vert a(e^+\vert d^+)?\vert b(e^+\vert d^+)?)$
8	$(a\vert b)(cdefg)^*$ $hijklmnopq(r\vert s)^*$	$(a\vert b)(cdefg)^*$ $hijklmnopq(r\vert s)^*$	$((((a\vert b)hijabcdefg)\vert b\vert a)$ $(c\vert g\vert f\vert e\vert d\vert s\vert r)^+((b\vert a)?hijkamnopq))$
9	$(abcd)^*\vert (e\vert f\vert g)^*\vert h\vert (ijklm)^*$	$(abcd)^*\vert (ijklm)^*\vert h\vert (e\vert f\vert g)^*$	$h(a\vert d\vert c\vert b\vert e\vert g\vert f\vert i\vert m\vert l\vert k\vert j)^+h$
10	$a^*\vert (b\vert c\vert d\vert e\vert f)^*\vert gh\vert (i\vert j\vert k)^*\vert$ $(lmn)^*$	$a^*\vert (b\vert c\vert d\vert e\vert f)^*\vert gh\vert (i\vert j\vert k)^*\vert$ $(lmn)^*$	$(a^+\vert gh)(e\vert f\vert d\vert i\vert j\vert l\vert n\vert m\vert k\vert c\vert b)^+$ $(a^+\vert gh)$

Table 5: DTDs generated by XTRACT and DDbE for Synthetic Data Set

that contain various combinations of sequencing and OR patterns. This clearly demonstrates the effectiveness of our generalization module in discovering these patterns and our MDL module in selecting these general candidate DTDs as the final DTDs.

Also, as discussed earlier, DDbE is not very good at factoring DTDs. For instance, unlike XTRACT, DDbE is unable to derive the final factored form for DTD 6. Finally, DDbE infers an extremely complex DTD for the simple DTD 7. The results for the synthetic data set clearly demonstrate the superiority of XTRACT's approach (based on the combination of generalization, factoring and the MDL principle) compared to DDbE's for the problem of inferring DTDs.

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

Table 6: DTDs generated by XTRACT and DDbE for Real-life Data Set

Real-life DTD Data Set. The DTDs generated by the two algorithms for the real-life data set are shown in Table 6. 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 it's factorization step takes into account the identity element "1" and simplifies expressions 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 Kleene stars. 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 6.) The reason for XTRACT's failure is that our generalization subsystem does not explicitly detect patterns containing the optional symbol (?) nested within Kleene stars. Working with other real-life DTDs, we also found that XTRACT can have some difficulties in inferring a concise and meaningful schema for very complicated DTD patterns with multiple levels of operator nesting. Finding such "difficult" patterns requires that a more sophisticated analysis of symbol occurrences within and across sequences be performed in XTRACT's generalization engine, and we plan to pursue this further as part of our future work.

7.4 Comparisons with *Fred* [Sha95]

Another recent approach towards automatic generation of DTDs from sample tagged documents, is the Grammar Builder Engine (GB-Engine) developed at the Online Computer Library Center (OCLC), Inc. OCLC's GB-Engine is embedded in a number of systems and *Fred* is currently the most popular of these systems. Automatic DTD creation is one of the services offered in *Fred*.

Despite our efforts, we have not been able to run the *Fred* system on our collection of XML documents, so all of our comments here are based on the development in the original *Fred* paper [Sha95]. In short, *Fred* is comparable with the Generalization module of XTRACT. *Fred* first deduces the structural types of a specific element by syntactic analysis of the document, and then it combines these types to deduce the resulting DTD. Combining element types is achieved by applying a sequence of *generalization* and *reduction* rules. An example of a generalization rule is replacing type aaa with type $a+$. Reduction rules include removing empty parentheses, collapsing ORs and ANDs, combining identical bases (e.g., $(ab)^*|a?b?$ becomes $(a?b?)^*$), eliminating redundancies (mostly up to a syntactic level), and so on. The user selects which of these rules are applied, but the order of application is predefined by the system.

Fred does not perform any kind of factoring. Factoring is present in both the DDbE and XTRACT systems and, from our experience, it turns out to be very useful in improving the conciseness of the resulting DTDs. In addition, compared to XTRACT, the *Fred* system lacks all the advantages of the MDL module, which we believe it is the most distinctive feature of our system and one of the most important contributions of our paper. In particular, the order in which the various generalization and reduction rules are applied by *Fred* is somehow arbitrary, and while a particular rule order is good for one element a different order might be better for another. XTRACT is addressing exactly this problem by proposing a well-motivated, information-theoretic measure of goodness for DTDs and by using this measure to select the best DTD among many candidates generated in the previous modules. Our experience with the XTRACT system has verified that the MDL module indeed selects the most intuitive DTDs. Due to the lack of the MDL module, we believe that DTDs generated by *Fred* could be arbitrarily complex or arbitrarily general.

8 Conclusions

In this paper, we presented the architecture of the XTRACT system for inferring a DTD for a database of XML documents. The DTD plays the role of a schema and thus contains valuable information about the structure of the XML documents that it describes. However, since DTDs are *not mandatory*, in a number of cases, documents in an XML database may not have an accompanying DTD. Thus, the DTD inference problem is important, especially given the critical role that the DTD plays in the storage as well as the formulation, optimization and processing of queries on the underlying data.

The problem of deriving the DTD for a set of documents is complicated by the fact that the DTD syntax incorporates the full expressive power of regular expressions. Specifically, as we showed, naive approaches that do not "generalize" beyond the input element sequences fail to deduce concise and semantically meaningful DTDs. Instead, XTRACT applies sophisticated algorithms in three steps to compute a DTD that is more along the lines that a human would infer. In the first *generalization* step, patterns within the input sequences are detected and more "general" regular expressions are substituted for them. These "generalized" candidate DTDs are then processed by the *factorization* step that factors common expressions within the DTDs to make them more succinct. The first two steps thus produce a range of candidate DTDs that vary in their conciseness and precision. In the third and final step, XTRACT employs the MDL principle to select from amongst the candidates the DTD that strikes the right balance between conciseness and preciseness – that is, a DTD that is concise, but at the same time, is not too general. The MDL principle maps naturally to the *Facility Location Problem (FLP)*, which we solved using an efficient approximation algorithm recently proposed in the literature.

We compared the quality of the DTDs inferred by XTRACT with those returned by the IBM alphaworks DDbE (Data Descriptors by Example) DTD extraction tool on synthetic as well as real-life DTDs. In our experiments, XTRACT outperformed DDbE by a wide margin, and for most DTDs it was able to accurately infer the DTD while 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 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. While we are encouraged by XTRACT's performance, we are continuing to further enhance our algorithms to infer even more complex DTDs (than those considered in this paper).

Acknowledgments: The work of Kyuseok Shim was partially supported by the Korea Science and Engineering Foundation (KOSEF) through the Advanced Information Technology Research Center (AITrc).

References

[Abi97] S. Abiteboul. Querying semi-structured data. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 1–18, 1997.

[Aho96] Helena Ahonen. *Generating grammars for structured documents using grammatical inference methods*. PhD thesis, University of Helsinki, 1996.

[AMN94] Helena Ahonen, Heikki Mannila, and Erja Nikunen. Forming grammars for structured documents: an application of grammatical inference. In *Proceedings of the 2nd Intl. Colloquium on Grammatical Inference and Applications*, pages 153–167, 1994.

[Ang78] Dana Angluin. On the complexity of minimum inference of regular sets. *Information and Control*, 39(3):337–350, 1978.

[BM82] R. K. Brayton and C. McMullen. The decomposition and factorization of boolean expressions. In *International Symposium on Circuits and Systems*, pages 49–54, 1982.

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

[Bra93] Alvis Brazma. Efficient identification of regular expressions from representative examples. *COLT*, pages 236–242, 1993.

[CG99] M. Charikar and S. Guha. Improved combinatorial algorithms for the facility location and k-median problems. In *40th Annual Symposium on Foundations of Computer Science*, 1999.

[DFS99] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with stored. In *Proc. of the ACM SIGMOD Conference on Management of Data*, June 1999.

[FS97] M. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *Proceedings of the International Conference on Database Theory (ICDT)*, 1997.

[GMW99] R. Goldman, J. McHugh, and J. Widom. From semistructured data to XML: Migrating the lore data model and query language. In *Proceedings of the 2nd International Workshop on the Web and Databases (WebDB '99)*, pages 25–30, June 1999.

[Gol67] E. Mark Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[Gol78] E. Mark Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, 1978.

[GW97] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, Athens, Greece, August 1997.

[Hoc82] D. S. Hochbaum. Heuristics for the fixed cost median problem. *Mathematical Programming*, 22:148–162, 1982.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automaton Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts, 1979.

[KMU95] P. Kilpeläinen, H. Mannila, and E. Ukkonen. MDL learning of unions of simple pattern languages from positive examples. In *Second European Conference on Computational Learning Theory, EuroCOLT*, pages 252–260, Barcelona, 1995.

[Law64] E. Lawler. An approach to multilevel boolean minimization. *Journal of the ACM*, July 1964.

[MRA95] Manish Mehta, Jorma Rissanen, and Rakesh Agrawal. MDL-based decision tree pruning. In *Int'l Conference on Knowledge Discovery in Databases and Data Mining (KDD-95)*, Montreal, Canada, August 1995.

[NAM98] S. Nestorov, S. Abiteboul, and R. Motwani. Extracting schema from semistructured data. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 295–306, 1998.

[Pit89] L. Pitt. Inductive inference, DFAs, and computational complexity. *Analogical and Inductive Inference*, pages 18–44, 1989.

[QR89] J. Ross Quinlan and Ronald L. Rivest. Inferring decision trees using the minimum description length principle. *Information and Computation*, 80:227–248, 1989.

[Ris78] J. Rissanen. Modeling by shortest data description. *Automatica*, 14:465–471, 1978.

[Ris89] J. Rissanen. *Stochastic Complexity in Statistical Inquiry*. World Scientific Publ. Co., 1989.

[Sha95] Keith E. Shafer. Creating dtds via the gb-engine and fred. In *Proceedings of the SGML'95 Conference*, Boston, MA, December 1995. `http://www.oclc.org/fred/docs/sgml95.html`.

[SHT$^+$99] 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 Int'l Conf. on Very Large Data Bases*, Edinburgh, Scotland, 1999.

[Wan89] A. R. R. Wang. *Algorithms for Multi-level Logic Optimization*. PhD thesis, The University of California, Berkeley, 1989.

[Wid99] J. Widom. Data management for XML: research directions. *IEEE Data Engineering Bulletin*, 22(3), 1999.

[YLT00] Matthew Young-Lai and Frank WM. Tompa. Stochastic grammatical inference of text database structure. *Machine Learning*, 40(2):111–137, August 2000.