# Исследование и разработка системы генерирования DTD для XML документов

**Леонов А. В. (al@tt.ru), Хуснутдинов Р. Р.**

**Институт физико-технической информатики**

## Аннотация

Описаны общие принципы построения системы генерирования DTD для XML документов, предложен ряд методов и алгоритмов для различных этапов генерирования DTD, разработана и реализована в качестве опытного образца система генерирования DTD для массива XML документов, которая может эффективно применяться как для решения прикладных задач, так и для дальнейших разработок.

## 1. Введение

На сегодняшний день язык XML (Extensible Markup Language) [1] фактически стал стандартом разметки данных для их хранения и обмена информацией между различными приложениями. XML активно используется в самых разных областях научной и коммерческой деятельности, и в обозримом будущем его популярность будет только расти. С ростом объема данных в формате XML существенную важность приобретают задачи хранения XML документов в базах данных и организации обмена XML документами между различными приложениями. Эти проблемы решаются наиболее эффективно в том случае, когда известна информация о внутренней структуре XML документа – DTD (Document Type Descriptor) или XML Schema.

XML документ может сопровождаться DTD, однако это условие не является обязательным (в отличие от стандарта SGML [2]). На сегодняшний день существует множество XML документов, которые не имеют DTD: это документы, полученные путем конвертации из HTML или других открытых форматов, а также конвертированные в XML из специализированных корпоративных систем хранения информации. Поскольку общее число XML документов пока еще сравнительно невелико (по отношению к общему объему электронных документов), очень вероятно, что большая часть XML документов, которые появятся в ближайшем будущем, также будут получены путем автоматической конвертации из хранилищ данных различных форматов. Скорее всего, в большинстве случаев эти массивы XML документов не будут иметь соответствующих им DTD.

Таким образом, одной из наиболее актуальных задач в области автоматической обработки XML документов сегодня является разработка методов и алгоритмов автоматического генерирования[1] DTD для массива XML документов. Эта задача является далеко не такой простой, как может показаться на первый взгляд. До сих пор нет доступных программных продуктов, которые позволяли бы генерировать DTD для массива XML документов с приемлемым качеством. Можно отметить несколько систем, которые содержат функцию автоматического или полуавтоматического построения DTD для XML документа – DDbE [3], Allora [4], а также некоторые XML редакторы – XML Spy [5], SAXON [6] и др. Однако, насколько мы можем судить по имеющейся в нашем распоряжении информации, все эти системы реализуют достаточно простые и прямолинейные подходы для генерирования DTD и малоэффективны для решения практических задач.

[1] В англоязычной литературе для обозначения процесса автоматического построения DTD для XML документа используется целый ряд терминов - “creating”, “building”, “generating”, “extraction”, “mining”.

На данный момент в открытой печати подробно описана лишь одна достаточно развитая система, предназначенная для автоматического генерирования DTD – система XTRACT, разработанная в Bell Laboratories в рамках проекта SERENDIP [7]. Принцип работы этой системы основан на общем алгоритме построения DTD для SGML документов, описанном в [8] – это обобщение исходных последовательностей вложенных элементов, упрощение получившихся выражений и выбор наилучшего из них. Также определенный интерес представляют работы [9] и [10], где предлагаются методы построения приблизительно эквивалентных регулярных выражений для достаточно длинных строк символов. Общие вопросы выделения логической структуры для массива слабоструктурированных данных рассмотрены в [11], [12], [13], но предлагаемые в этих работах схемы данных существенно отличаются от DTD и не могут применяться для генерирования регулярных выражений.

Целью данной работы является построение системы генерирования DTD для массива XML документов. Авторы рассматривают существующие подходы к решению этой задачи на примере архитектуры и принципов работы системы XTRACT, предлагают ряд новых методов и алгоритмов для различных этапов генерирования DTD, после чего разрабатывают и реализуют в качестве опытного образца собственную систему генерирования DTD для массива XML документов, которая может эффективно применяться как для решения прикладных задач, так и для дальнейших разработок.

## 2. Общие принципы построения системы генерирования DTD для массива XML документов

Основную сложность в задаче генерирования DTD для массива XML документов представляет задача построения DTD для элемента с содержимым из элементов. Действительно, DTD документа представляет собой простое объединение DTD отдельных элементов и атрибутов, каждый из которых никак не зависит от DTD других элементов и атрибутов. Построение DTD для атрибутов, элементов с содержимым типа #PCDATA и элементов со смешанным содержимым осуществляется относительно просто.

Таким образом, здесь и далее мы будем рассматривать задачу построения DTD для элемента с содержимым из элементов. Более строго эту задачу можно сформулировать следующим образом. Пусть имеется массив XML документов. Пусть элемент X встречается в этом массиве n раз, которым соответствуют n последовательностей вложенных в него элементов: $s_1$, $s_2$, … $s_n$. Задача состоит в том, чтобы для множества последовательностей I = {$s_1$, $s_2$… $s_n$} построить регулярное выражение (DTD элемента X), которое бы описывало все эти последовательности.

Автоматическое построение краткого и точного DTD для элемента с содержимым из элементов – достаточно сложная задача. Дело в том, что простые, "прямолинейные" подходы к автоматическому генерированию DTD дают слишком громоздкие и сложные для восприятия регулярные выражения, плохо отражающие внутреннюю структуру элемента и совершенно не похожие на то, что предложил бы в аналогичной ситуации человек. Поясним это на следующем примере.

**Пример 1.** Один из тривиальных подходов заключается в том, что для элемента строится регулярное выражение, которое точно соответствует всем встречающимся в массиве XML документов последовательностям вложенных в него элементов, и только им. Рассмотрим, например, XML документ, представляющий собой список литературы. Пусть этот документ состоит из последовательности элементов <paper>, каждый из которых, в свою очередь, содержит один вложенный элемент <title> и один или несколько вложенных элементов <author>. Для краткости обозначим <title> символом t, <author> – символом a.

Пусть элемент <paper> встречается в XML документе 5 раз, которым соответствуют следующие последовательности вложенных элементов: t, ta, taa, taaa, taaaa. Описанный выше тривиальный подход даст в этом случае следующее регулярное выражение, точно соответствующее всем этим последовательностям и только им: t|ta|taa|taaa|taaaa, которое может быть упрощено до t|t(a|a(a|a(a|aa))). Очевидно, что это регулярное выражение слишком громоздко и намного хуже отражает внутреннюю структуру элемента <paper>, чем, скажем, ta*, которое в данной ситуации, скорее всего, предложил бы человек.

Другими словами, для получения "хорошего" DTD, как правило, необходимо в той или иной степени обобщить встречающиеся в массиве XML документов последовательности вложенных элементов. На уровне синтаксиса DTD это означает построение регулярных выражений с использованием метасимволов * и +. Очевидно, что каждое регулярное выражение, полученное в результате обобщения, будет описывать не только тот набор последовательностей, которые реально присутствуют в анализируемом массиве XML документов, но и бесконечное число последовательностей, которые в данном массиве не встречаются (например, выражение ta* описывает, в том числе, последовательности taaaaa, taaaaaa и т. д.). Ясно также, что для каждого множества последовательностей можно предложить несколько обобщенных регулярных выражений, каждое из которых будет описывать все входящие в это множество последовательности.

Таким образом, при переходе от тривиальных методов построения DTD к методам с использованием обобщения (более близким к тому образу действий, которым в подобных ситуациях пользуется человек), возникает проблема выбора наилучшего DTD. Грубо говоря, необходимо определить DTD с наиболее приемлемой степенью обобщения. С одной стороны, DTD должен быть достаточно краток, "прозрачен", прост для восприятия; с другой стороны, DTD должен быть точен, то есть, он не должен описывать слишком много последовательностей, не встречающихся в анализируемом массиве XML документов. Приведем пример, поясняющий это утверждение.

**Пример 2.** Предположим, I = {ab, abab, ababab}. Для этого множества последовательностей можно предложить следующие DTD: (1) - (a|b)*, (2) - ab|abab|ababab, (3) - (ab)*, (4) - ab|ab(ab|abab) и т. д. Выражение (1) описывает произвольную последовательность символов a и b, выражение (2) – простая дизъюнкция (логическое ИЛИ) всех последовательностей из множества I (т. е. описанный выше тривиальный подход), выражение (4) получено из (2) путем факторизации (т. е. вынесения за скобки последовательности ab из двух последних членов дизъюнкции (2)). Недостаток выражения (1) заключается в том, что оно представляет собой слишком сильное обобщение, и совершенно не отражает внутренней структуры последовательностей в множестве I. С другой стороны, выражения (2) и (4) точно соответствуют последовательностям в множестве I, но не выделяют в них никаких общих внутренних структур, что позволило бы сделать DTD компактнее или проще для восприятия. Таким образом, ни одно из выражений (1), (2) или (4) нельзя признать "хорошим" DTD для множества I. Наиболее привлекательным из предложенных DTD является выражение (3), которое обобщает исходное множество последовательностей, сохраняя при этом наиболее значимую информацию об их внутренней структуре. Именно такое выражение, скорее всего, предложил бы в данной ситуации человек.

Задачу построения DTD для элемента с содержимым из элементов в общем случае можно разбить на три этапа: обобщение исходных последовательностей, факторизация полученных регулярных выражений и построение наилучшего DTD на основе полученного множества регулярных выражений. Наибольшую сложность представляет последний этап, так как "краткость" и "точность" DTD являются конкурирующими

критериями, причем критериями качественными. Для создания алгоритма построения наилучшего DTD необходимо найти способ количественного выражения "краткости" и "точности" и разработать метод нахождения оптимального соотношения между ними. Эта задача решается с использованием принципа MDL (Minimum Description Length).

## 2.1. Принцип MDL

Принцип MDL [14], [15] в общей формулировке гласит следующее: наилучшей теорией для описания набора данных является та теория, для которой минимальна сумма:

A. длины теории, в битах;
B. длины данных, описанных (закодированных) с помощью этой теории, в битах.

Мы будем называть эту сумму MDL-стоимостью теории. Принцип MDL является общим и в каждой конкретной ситуации должен быть подходящим образом адаптирован.

В нашем случае теория – это DTD, а данные – это исходное множество последовательностей I. Покажем, что части (A) и (B), составляющие MDL-стоимость DTD, непосредственно соответствуют таким характеристикам DTD, как краткость и точность. Действительно, часть (A) – число битов, необходимых для описания (кодирования) DTD – это прямой количественный критерий краткости DTD. Далее отметим, что для описания множества последовательностей I в терминах DTD требуется тем меньше битов, чем более точен этот DTD, поскольку более точный DTD лучше отражает внутреннюю структуру последовательностей, входящих в I. Соответственно, часть (B) – это прямой количественный критерий точности DTD.

Таким образом, принцип MDL дает нам элегантный способ количественного выражения краткости и точности DTD и объединения этих конкурирующих характеристик в рамках единого критерия. Лучшим DTD из всех DTD, которые можно построить на основе полученного множества регулярных выражений, будет тот, для которого MDL-стоимость минимальна. Разумеется, для практического применения принципа MDL необходимо разработать способ вычисления частей (A) и (B) MDL-стоимости в нашем случае, то есть, схему кодирования DTD и описываемого с его помощью множества последовательностей I в последовательности битов. Строгое описание такой схемы кодирования будет дано в разделе "Модуль MDL". Ниже мы приведем простой пример (с использованием упрощенного варианта реальной схемы), показывающий, насколько близко рейтингование DTD на основе их MDL-стоимости соответствует нашему интуитивному представлению о качестве DTD.

**Пример 3.** Рассмотрим множество последовательностей I и DTD-кандидаты из примера 2. В нашей упрощенной схеме будем считать, что MDL-стоимость каждого символа - 1.

Итак, DTD (1) – (a|b)*. Стоимость этого DTD – 6. Для того, чтобы закодировать последовательность ab с помощью этого DTD, требуется 1 символ для определения числа повторений дизъюнкции (a|b) (это число 2), и 2 символа для определения того, какой из членов дизъюнкции – a или b – выбирается в каждом случае (это числа 1 и 2). Следовательно, стоимость кодирования последовательности ab с помощью DTD (1) – 3. Аналогичным образом, стоимость кодирования последовательности abab – 5, ababab – 7. Итого, полная MDL-стоимость DTD (1) – 6+3+5+7 = 21.

DTD (2) – ab|abab|ababab. Стоимость этого DTD – 14. Для того, чтобы закодировать каждую из последовательностей ab, abab и ababab с помощью этого DTD, требуется по 1 символу для определения того, какой из членов дизъюнкции – ab, abab или ababab –

выбирается в каждом случае (это числа 1, 2 и 3). Итого, полная MDL-стоимость DTD (2) – 14+1+1+1 = 17.

DTD (3) – (ab)*. Стоимость этого DTD – 5. Для того, чтобы закодировать каждую из последовательностей ab, abab или ababab с помощью этого DTD, требуется по 1 символу для определения числа повторений конъюнкции (ab) в каждом случае (это числа 1, 2 и 3). Итого, полная MDL-стоимость DTD (3) – 5+1+1+1 = 8.

Наконец, DTD (4) – ab|ab(ab|abab). Стоимость этого DTD – 14. Для того, чтобы закодировать последовательность ab с помощью этого DTD, требуется 1 символ (число 1, указывающее, что выбирается первый член главной дизъюнкции). Для того, чтобы закодировать последовательность abab с помощью этого DTD, нам требуется 2 символа (число 2, указывающее, что выбирается второй член главной дизъюнкции, и число 1, указывающее, что выбирается первый член вложенной дизъюнкции). Аналогичным образом, для кодирования последовательности ababab с помощью этого DTD также требуется 2 символа. Итого, полная MDL-стоимость DTD (4) – 14+1+2+2 = 19.

Таким образом, согласно принципу MDL, наилучшим DTD является DTD (3), что в точности соответствует нашему интуитивному представлению.

## 2.2. Архитектура системы XTRACT

Архитектура системы XTRACT представлена на рис. 1. Система состоит из трех модулей: модуля обобщения, модуля факторизации и модуля MDL. Входными данными для модуля обобщения служит множество последовательностей I. Результатом работы модуля обобщения является множество выражений $S_G$, которое включает регулярные выражения, полученные в результате обобщения последовательностей из набора I, а также все исходные последовательности из I. Множество $S_G$ служит входными данными для модуля факторизации. Результатом работы модуля факторизации является множество выражений $S_F$, который включает регулярные выражения, полученные в результате факторизации выражений из множества $S_G$, а также все выражения из множества $S_G$. Наконец, множество $S_F$ служит входными данными для модуля MDL, который выбирает из $S_F$ подмножество выражений, которое покрывает все исходные последовательности из множества I и MDL-стоимость которого минимальна. Итоговый DTD является дизъюнкцией (логическим ИЛИ) всех выражений из S.

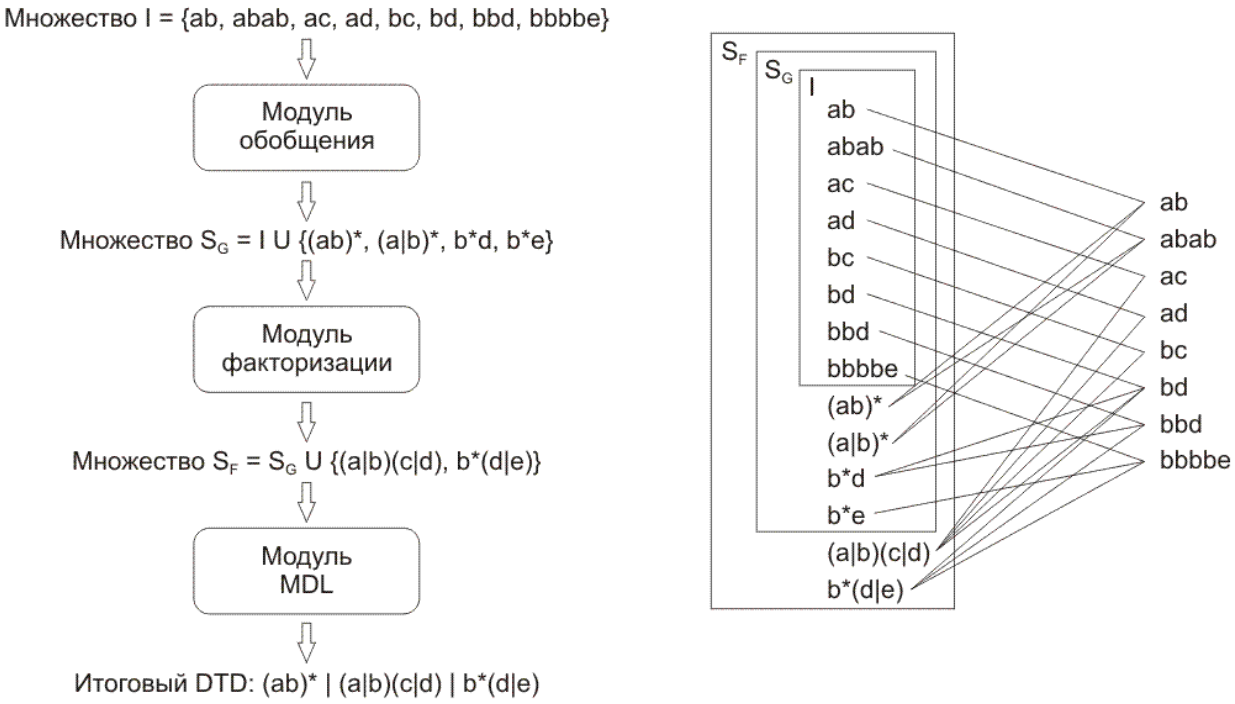


*Рис. 1. Архитектура системы XTRACT.*

Модуль обобщения обрабатывает все последовательности из множества I, генерируя для каждой из них (при возможности) обобщенные регулярные выражения с использованием метасимволов * и |. Например, для последовательностей abab и bbbe, рис. 1, модуль обобщения построит выражения (ab)*, (a|b)* и b*e. В идеале, на этапе обобщения нужно построить все возможные обобщенные выражения, оставляя задачу выбора лучших из них для модуля MDL. Однако, число таких выражений может быть исключительно велико, поэтому уже на шаге обобщения необходимо ограничить число выражений в множестве $S_G$ на основе эмпирически подобранных параметров.

Модуль факторизации обрабатывает все выражения из множества $S_G$, генерируя на основе нескольких выражений из $S_G$ (при возможности) новые регулярные выражения с использованием факторизации (т. е. вынесения за скобки повторяющихся последовательностей символов). Для этого используются соответствующим образом адаптированные алгоритмы факторизации булевых выражений из области логической оптимизации [16], [17]. Например, факторизация выражений b*d и b*e даст b*(d|e), выражения ac, ad, bc, bd будут факторизованы в (a|b)(c|d). Этап факторизации важен, так как позволяет получить более короткие выражения, которые, возможно, окажутся более предпочтительными на этапе построения наилучшего DTD. Модуль факторизации, как и модуль обобщения, строит не все возможные факторизованные выражения (число которых может быть огромно), а лишь некоторые из них на основе эмпирически подобранных алгоритмов.

Модуль MDL выбирает из множества выражений $S_F$, полученного в результате работы модулей обобщения и факторизации, подмножество выражений S, которое покрывает все последовательности из набора I и MDL-стоимость которого минимальна. Для этого используются соответствующим образом адаптированные алгоритмы из [18], [19]. Итоговый DTD представляет собой дизъюнкцию (логическое ИЛИ) всех выражений подмножества S. На рис.1 справа показано, какие исходные последовательности из множества I (правая колонка) покрывает каждое выражение из множества $S_F$ (левая колонка).

## 2.3. Модуль обобщения

Алгоритмы обобщения, использованные в системе XTRACT, представлены на рис. 2. Процедура GENERALIZE генерирует несколько обобщенных регулярных выражений для каждой последовательности из исходного множества I и добавляет их в множество $S_G$ (которое изначально совпадает с I). Для этого последовательно вызываются процедуры DISCOVERSEQPATTERN и DISCOVERORPATTERN с различными параметрами.

**процедура** GENERALIZE(I)
**начало**
1. **для каждой** последовательности s в I
2. добавить s в $S_G$
3. **для** r := 2,3,4
4. s' := DISCOVERSEQPATTERN(s,r)
5. **для** d := 0,1*|s'|,0,5*|s'|,|s'|
6. s'' := DISCOVERORPATTERN(s',d)
7. добавить s'' в $S_G$

**конец**

**процедура** DISCOVERSEQPATTERN(s,r)
**начало**
1. **повторять**
2. пусть x – подпоследовательность в s, которая последовательно
3. повторяется максимальное число раз (≥ r)
4. заменить все последовательные повторения x в s
5. новым подстановочным символом $A_i = (x)*$
6. **до тех пор, пока** (s не содержит ≥ r последовательных повторений
7. любой подпоследовательности x)
8. **вернуть** s

**конец**

**процедура** DISCOVERORPATTERN(s,d)
**начало**
1. $s_1, s_2, \ldots, s_n$ := PARTITION(s,d)
2. **для каждой** подпоследовательности $s_j$ из $s_1, s_2, \ldots, s_n$
3. пусть набор различных символов в $s_j$ – это $a_1, a_2, \ldots, a_m$
4. **если** (m>1)
5. заменить подпоследовательность $s_j$ в последовательности s
6. новым подстановочным символом $A_i = (a_1|a_2|\ldots|a_m)*$
7. **вернуть** s

**конец**

**процедура** PARTITION(s,d)
**начало**
1. i := start := end := 1
2. $s_i$ = s[start, end]
3. **пока** (end < |s|)
4. **пока** (end < |s| **и** любой символ из $s_i$ встречается справа от $s_i$
5. на расстоянии от этого символа, меньше или равном d)
6. end := end+1;
7. $s_i$ := s[start, end];
8. **если** (end < |s|)
9. i := i+1;

```
10.                 start := end+1;
11.                 end := end+1;
12.                 s_i := s[start, end];
13.     вернуть s_1, s_2, …, s_i
конец
```

*Рис. 2. Алгоритм обобщения.*

### *2.3.1. Генерирование выражений вида (x)**

Процедура DISCOVERSEQPATTERN получает на входе последовательность s и параметр r>1. В том случае, если последовательность s содержит хотя бы одну подпоследовательность вида xx…x (где x – один символ или последовательность символов) с числом повторений ≥ r, процедура DISCOVERSEQPATTERN возвращает регулярное выражение, которое получено из последовательности s путем замены этой подпоследовательности на регулярное выражение (x)*. Если таких подпоследовательностей несколько, выбирается наиболее длинная из них и процедура повторяется снова.

Важно отметить, что в последовательность s подставляется не само регулярное выражение (x)*, а заменяющий ее подстановочный символ $A_i$. При этом соответствие между регулярными выражениями и подстановочными символами взаимно однозначное во всей системе XTRACT. Благодаря этому, упрощается описание процедур, так как они всегда оперируют простыми последовательностями символов (без метасимволов).

В частности, процедура DISCOVERSEQPATTERN может повторяться несколько раз, и на каждой последующей итерации подстановочные символы обрабатываются точно так же, как обычные символы. Например, если на вход процедуры DISCOVERSEQPATTERN поступает последовательность s = abababcababc и параметр r=2, то после первой итерации получается последовательность $A_1$cababc, где $A_1$ означает (ab)*, после второй – $A_1cA_1c$, после третьей – $A_2$, где $A_2$ означает ((ab)*c)*.

Процедура DISCOVERSEQPATTERN вызывается из процедуры GENERALIZE с тремя различными значениями параметра r, что позволяет генерировать регулярные выражения с различной степенью обобщения. Например, для последовательности aabbb процедура DISCOVERSEQPATTERN с r=2 даст выражение a*b*, а с r=3 – выражение aab*. На этапе выбора по принципу MDL более предпочтительным может оказаться как то, так и другое выражение – в зависимости от того, какое из них точнее описывает последовательности из исходного множества I.

### *2.3.2. Генерирование выражений вида $(a_1|a_2|…|a_m)$**

Процедура DISCOVERORPATTERN заменят локальные скопления символов $a_1, a_2, …, a_m$ в последовательности s на регулярные выражения вида $(a_1|a_2|…|a_m)$*. Идея состоит в том, что если в последовательности s есть подпоследовательность, которая представляет собой частое повторение символов из набора $\{a_1, a_2, …, a_m\}$, то эта подпоследовательность с большой вероятностью описывается регулярным выражением вида $(a_1|a_2|…|a_m)$*.

Для выявления таких локальных скоплений используется процедура PARTITION. Эта процедура разбивает последовательность s на подпоследовательности $s_1, s_2, …, s_n$ таким образом, чтобы расстояние между одинаковыми символами в разных подпоследовательностях оказалось не меньше d (где d – входной параметр процедуры DISCOVERORPATTERN). Этот принцип разбиения проиллюстрирован на рис. 3. Процедура DISCOVERORPATTERN затем просто заменят каждую подпоследовательность $s_i$ на

регулярное выражения вида $(a_1|a_2|\ldots|a_m)^*$, где $a_1$, $a_2$, …, $a_m$ – различные символы в подпоследовательности $s_i$.

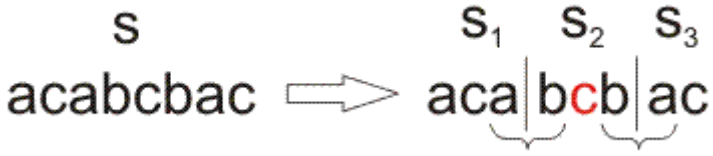


*Рис. 3. Принцип разбиения на подпоследовательности (для d=2).*

Опишем более подробно процедуру PARTITION. Введем следующие обозначения. Для произвольной последовательности s, выражение s[i,j] означает подпоследовательность s, которая начинается i-м символом s и заканчивается j-м символом s. Процедура PARTITION строит подпоследовательности последовательности s по порядку: $s_1$, $s_2$ и так далее. Предположим, построены подпоследовательности $s_1 \ldots s_j$. Тогда подпоследовательность $s_{j+1}$ начинается с символа, следующего за последним символом подпоследовательности $s_j$, и продлевается вправо до тех пор, пока не будет выполнено условие "ни один символ из $s_{j+1}$ не встречается справа от $s_{j+1}$ на расстоянии от этого символа, меньше или равном d". По построению, такой символ не может встретиться слева от $s_j$.

Отметим, что на вход процедуры DISCOVERORPATTERN поступают выражения, которые получились в результате выполнения процедуры DISCOVERSEQPATTERN. Это позволяет строить регулярные выражения более сложной формы, чем при независимом выполнении этих процедур или при их выполнении в обратном порядке. Например, для входной последовательности s = abcbca процедура DISCOVERSEQPATTERN с параметром r=2 даст $s'=aA_1a$, где $A_1=(bc)^*$, а обработка последовательности s' процедурой DISCOVERORPATTERN с параметром d=|s'| даст $s''=A_2$, где $A_2=(a|A_1)^*=(a|(bc)^*)^*$. Выполнение же процедуры DISCOVERORPATTERN сразу для последовательности s с параметром d=|s| дало бы более простое выражение (a|b|c|)*.

Отметим также, что процедура DISCOVERORPATTERN вызывается с различными значениями параметра d, которые выражаются в долях от длины входной последовательности. Это позволяет получать регулярные выражения с разной степенью обобщения. Например, для входной последовательности abcbac процедура DISCOVERORPATTERN вернет:

- $aA_1ac$ для d=2, где $A_1=(b|c)^*$
- $aA_2$ для d=3, где $A_2=(a|b|c)^*$
- $A_2$ для d=4, где $A_2=(a|b|c)^*$

## 2.4. Модуль факторизации

На вход модуля факторизации поступает множество регулярных выражений $S_G$, полученное в результате работы модуля обобщения. Модуль факторизации добавляет к этому множеству новые выражения, полученные в результате факторизации (вынесения за скобки повторяющхся последовательностей символов) двух или более выражений из множества $S_G$. Поскольку факторизованные выражения короче, чем простая дизъюнкция соответствующих выражений из множества $S_G$, они могут оказаться более предпочтительными на этапе построения наилучшего DTD.

В идеале, на этапе факторизации нужно строить факторизованные выражения для всех подмножеств регулярных выражений из множества $S_G$. Однако, число таких подмножеств может оказаться очень велико, поэтому авторы системы XTRACT используют эмпирические алгоритмы для выбора наиболее "перспективных" для факторизации подмножеств. Таким образом, модуль факторизации состоит из двух суб-модулей: первый

из них выбирает "перспективные" для факторизации подмножества из множества $S_G$, а второй строит факторизованные формы для этих подмножеств.

Напомним, что регулярные выражения, полученные в результате работы модуля обобщения, могут содержать подстановочные символы. Прежде чем множество выражений $S_G$ поступает на вход модуля факторизации, все подстановочные символы заменяются на оригинальные выражения.

### *2.4.1. Выбор подмножеств множества $S_G$ для факторизации*

Алгоритм выбора "перспективных" для факторизации подмножеств множества $S_G$ основан на двух идеях. С одной стороны, подмножество S множества $S_G$ является хорошим кандидатом для факторизации, если факторизованная форма выражений из S существенно короче, чем простая дизъюнкция этих выражений. С другой стороны, факторизация выражений, которые описывают одинаковые или близкие наборы последовательностей из исходного множества I, вряд ли позволит улучшить финальный DTD. Хотя множество $S_G$ и может содержать несколько регулярных выражений, полученных в результате обобщения одной входной последовательности из I, в финальный DTD, скорее всего, войдет только одно из них.

Таким образом, "перспективное" для факторизации подмножество S множества $S_G$ должно удовлетворять двум требованиям:

1. Каждое выражение в S должно иметь общий префикс или суффикс с несколькими другими выражениями в S. При этом, чем больше выражений в S имеют общие префиксы или суффиксы и чем больше длина общих префиксов/суффиксов, тем лучшего результата можно ожидать от факторизации подмножества S.
2. Пересечение между подмножествами последовательностей из I, которые описываются регулярными выражениями D и D', должно быть достаточно мало.

Для того, чтобы более строго определить критерий (2), введем следующие обозначения:

- *покрытие(D)* – подмножество последовательностей из множества I, которые описываются регулярным выражением D;
- *перекрытие(D,D')* = |*покрытие(D)* ∩ *покрытие(D')*| / |*покрытие(D)* ∪ *покрытие(D')*| - отношение числа последовательностей, описываемых одновременно обеими выражениями D и D', к общему числу последовательностей, описываемых выражениями D и D'.

Таким образом, строгая формулировка критерия (2) выглядит следующим образом: подмножества S должны строится так, чтобы для любой пары D и D' из подмножества S *перекрытие(D,D')* < δ, где δ – достаточно малая величина, заданная пользователем.

Для того, чтобы более строго определить критерий (1), введем следующие обозначения:

- *pref(D)* – множество префиксов D (т. е., выражений, полученных отсечением правой части регулярного выражения D);
- *suf(D)* – множество суффиксов D (т. е., выражений, полученных отсечением левой части регулярного выражения D);
- *psup(p,S)* – число выражений в подмножестве S, для которых p – префикс;
- *ssup(s,S)* – число выражений в подмножестве S, для которых s – суффикс;
- *рейтинг(D,S)* = max [{|p|**psup(p,S)*: p принадлежит *pref(D)*}∪{|s|**ssup(s,S)*: s принадлежит *suf(D)*}].

Функция *рейтинг(D,S)* - это просто максимальное произведение длины префикса/суффикса выражения D на число выражений в подмножестве S с таким же

префиксом/суффиксом. Функция *рейтинг(D,S*) вводится для того, чтобы в численном виде выразить степень схожести между префиксами/суффиксами выражения D и префиксами/суффиксами остальных выражений подмножества S.

Таким образом, строгая формулировка критерия (1) выглядит следующим образом: подмножества S должны строится так, чтобы для каждого D из S значения *рейтинг(D,S)* и *рейтинг(D,$S_G$)* были как можно больше.

Выбор подмножеств S из множества $S_G$, удовлетворяющих критериям (1) и (2), осуществляется процедурой FACTORSUBSETS, рис. 4. Эта процедура выбирает k таких подмножеств (где k – параметр, задаваемый пользователем), каждое из которых затем подвергается факторизации путем вызова процедуры FACTOR. Процедура FACTOR в общем случае возвращает дизъюнкцию вида $F_1 | F_2 | \ldots | F_m$ для каждого подмножества S, соответственно, каждое из выражений $F_i$ добавляется к множеству $S_F$.

**процедура** FACTORSUBSETS($S_G$)
**начало**
1. **для каждого** выражения D из $S_G$
2. вычислить *рейтинг(D, $S_G$)*
3. $S_F$ := S' := $S_G$; SeedSet := Ø
4. **для** i := от 1 до k
5. пусть D – выражение из S' с максимальным значением *рейтинг(D,S')*
6. SeedSet := SeedSet ∪ D
7. S' := S' – {D': *перекрытие(D,D')* ≥ δ}
8. **для каждого** выражения D в множестве SeedSet
9. S := {D}
10. S' := $S_G$ – {D': *перекрытие(D,D')* ≥ δ}
11. **пока** S' не станет пустым
12. пусть D' – выражение из S' с макс. значением *рейтинг(D',S)*
13. S := S ∪ D'
14. S' = S' – {D'': *перекрытие(D',D'')* ≥ δ}
15. F := FACTOR(S) /* примечание: $F = F_1 | F_2 | \ldots | F_m$ */
16. $S_F$ := $S_F$ ∪ {$F_1$, $F_2$, …$F_m$}

**конец**

*Рис. 4. Процедура FACTORSUBSETS.*

Процедура FACTORSUBSETS работает следующим образом. Сначала (шаги 4-7) из множества $S_G$ выбирается подмножество SeedSet из k выражений, которые имеют наибольшие *рейтинги* по отношению к $S_G$ и минимальные *перекрытия* друг с другом. Затем (шаги 9-14) каждое из этих выражений используется для построения подмножества S для факторизации (таким образом, генерируется всего k подмножеств). Это построение осуществляется итерационно: на первом шаге S состоит только из начального выражения, каждое следующее выражение добавляется в S из $S_G$ по правилу: "*рейтинг* выражения по отношению к S максимален и его *перекрытие* с выражениями из S меньше δ".

### *2.4.2. Факторизация подмножества регулярных выражений*

Алгоритмы факторизации множества регулярных выражений представляют собой соответствующим образом адаптированные алгоритмы факторизации логических выражений из булевой алгебры [17]. Псевдо-код, описывающий процедуру факторизации, представлен на рис. 5.

```
процедура FACTOR(S)    /* Прим.: S – множество выражений, который нужно
факторизовать */
начало
1.      DivisorSet := FINDALLDIVISORS(S)
2.      если (DivisorSet = Ø)
3.              вернуть дизъюнкцию (логическое ИЛИ) выражений из S
4.      DivisorList := Ø
5.      для каждого делителя V из множества DivisorSet
6.              Q, R := DIVIDE(S,V)
7.              добавить (V, Q, R) в множество DivisorList
8.      найти самый короткий триплет (Vi, Qi, Ri) в множестве DivisorList
9.      вернуть (FACTOR(Vi)FACTOR(Qi) | FACTOR(Ri))
конец

процедура FINDALLDIVISORS(S)
начало
1.      DivisorSet := Ø
2.      для каждой последовательности символов s, такой, что s – суффикс
3.      для двух или более выражений в наборе S
4.              DivisorSet := DivisorSet ∪ {{p:ps принадлежит S}}
5.      вернуть DivisorSet
конец

процедура DIVIDE(S,V)
начало
1.      для каждой последовательности символов p из множества V
2.              qp := {s: ps принадлежит S}
3.      Q := пересечение всех qp для p, принадлежащих V
4.      R := S – V º Q
                /* Примечание:  V º Q – это множество последовательностей,
                полученных путем присоединения каждой последовательности из Q
                справа к каждой последовательности из V */
5.      вернуть Q, R
конец
```

*Рис. 5. Алгоритм факторизации.*

Ключевым элементом алгоритма факторизации является процедура DIVIDE(S,V). Эта процедура осуществляет деление множества S на множество V и возвращает частное Q и остаток R. Деление в данном случае означает поиск таких множеств Q и R, что $S = V º Q ∪ R$, где $V º Q$ – это множество выражений, полученных в результате присоединения каждого выражения из Q справа к каждому выражению из V. Например, для множества S = {b, c, ab, ac, df, dg, ef, eg} и делителя V = {d, e} процедура DIVIDE(S,V) вернет частное Q = {f, g} и остаток R ={b, c, ab, ac}.

Поясним основные принципы работы алгоритма факторизации на примере обработки множества S = {b, c, ab, ac, df, dg, ef, eg}.

1. Строится набор потенциальных делителей для S. Каждый потенциальный делитель – это просто множество префиксов, имеющих общие суффиксы в S. Например, для рассматриваемого S потенциальные делители – это {d, e} (целых два общих

суффикса – f и g) и {1, a} (тоже два общих суффикса – b и c). "1" – специальный символ, удовлетворяющий правилу: "1s = s1 = s для любой последовательности s".

2. Из набора потенциальных делителей выбирается "наилучший" делитель V. Для этого сначала вычисляются частное Q и остаток R для каждого из потенциальных делителей, и выбирается тот делитель V, для которого триплет (V, Q, R) имеет наименьший размер. В нашем примере, будет выбран делитель V = {d, e}, так как триплет (V = {d, e}, Q = {f, g}, R = {b, c, ab, ac}) короче, чем (V = {1, a}, Q = {b, c}, R = {df, dg, ef, eg}).
3. Шаги 1 и 2 повторяются для каждого из полученных множеств V, Q и R до тех пор, пока не останется делителей. Иначе говоря, процедура FACTOR выполняется рекурсивно: FACTOR(S) = FACTOR(V)FACTOR(Q) | FACTOR(R). В нашем примере, множества V = {d, e} и Q = {f, g} уже не могут быть факторизованы, так как составляющие их выражения не имеют общих суффиксов. Таким образом, FACTOR (V) = (d|e) и FACTOR (Q) = (f|g). Множество R = {b, c, ab, ac} может быть факторизовано, так как имеет делитель $V_1$ = {1, a}. Повторяя шаги 1 и 2 для R, получим FACTOR (R) = (1|a)(b|c). В итоге факторизация множества S даст следующее выражение: (d|e)(f|g)|(1|a)(b|c).
4. Исключается специальный символ "1". Для этого выражения вида (1|s) заменяются на s?. Таким образом, мы получаем окончательную факторизованную форму подмножества S: (d|e)(f|g)|a?(b|c).

Для того, чтобы стал ясен смысл строк 2-3 процедуры DIVIDE, приведем еще один пример. Пусть S = {ax, bx, ay, by, cy}. Процедура FINDALLDIVISORS(S) найдет два делителя: $V_1$ = {a, b} (общие префиксы для x) и $V_2$ = {a, b, c} (общие префиксы для y). Выполнение процедуры DIVIDE(S,$V_1$) даст $q_a$={x, y}, $q_b$={x, y} и, соответственно, Q = $q_a \cap q_b$={x, y}. Выполнение процедуры DIVIDE(S,$V_2$) даст $q_a$={x, y}, $q_b$={x, y}, $q_c$={y} и, соответственно, Q = $q_a \cap q_b \cap q_c$ ={y}. Таким образом, факторизация S даст (a|b)(x|y)|cy.

## 2.5. Модуль MDL

Итак, результатом работы модуля факторизации является множество регулярных выражений $S_F$. Напомним, что $S_F$ – это объединение множества последовательностей из набора I, множества выражений $S_G$, полученных путем обобщения исходных последовательностей из набора I, и множества выражений, полученных путем факторизации двух и более выражений из $S_G$. Модуль MDL осуществляет выбор такого подмножества выражений S из множества $S_F$, что финальный DTD (дизъюнкция выражений из S) покрывает все последовательности из исходного множества I и при этом его MDL-стоимость минимальна. Напомним, что MDL-стоимость DTD складывается из (A) числа битов, необходимых для того, чтобы закодировать данный DTD, и (B) числа битов, необходимых для того, чтобы закодировать все последовательности из I на основе данного DTD. Соответственно, сначала мы опишем схему кодирования, которая применяется в системе XTRACT для вычисления значений (A) и (B), а затем – алгоритм выбора подмножества S из множества $S_F$.

### *2.5.1. Схема кодирования XTRACT*

Для вычисления числа битов (A), необходимых для того, чтобы закодировать данный DTD, в системе XTRACT применяется следующая схема. Пусть Σ – множество символов, которые встречаются в последовательностях набора I. Пусть M – множество метасимволов |, *, +, ?, ), (. Тогда DTD будет строкой из элементов множества Σ ∪ M. Пусть длина DTD – n. Тогда число битов, необходимых для кодирования данного DTD, вычисляется по формуле: $n*]\log_2(|\Sigma \cup M|)[$. Здесь $|\Sigma \cup M|$ - число элементов в Σ ∪ M, $]x[$ – ближайшее к $x$ целое число, большее или равное $x$.

Смысл этой формулы очевиден. Для того, чтобы закодировать $|\Sigma \cup M|$ различных элементов в двоичной форме, необходимо минимум $]\log_2(|\Sigma \cup M|)[$ двоичных разрядов. Два элемента можно закодировать с использованием одного разряда (0 и 1), три или четыре элемента – с использованием двух разрядов (00, 01, 10, 11) и т. д. Соответственно, длина DTD в двоичной кодировке вычисляется как произведение числа элементов DTD на число разрядов, необходимых для кодирования каждого элемента.

Пусть, например, $\Sigma = \{a, b\}$. Тогда длина DTD a*b* в битах будет $4*]\log_2(2+6)[ = 4*3 = 12$, длина DTD (ab|abb)(aa|ab*) – 16*3 = 48, и т. д.

Число битов (Б), необходимых для того, чтобы закодировать исходный набор последовательностей I на основе данного DTD, вычисляется как сумма чисел битов, необходимых для кодирования каждой последовательности из I. Суть алгоритма кодирования отдельной последовательности на основе данного DTD состоит в том, что сначала последовательность кодируется строкой индексов (чисел 0, 1, 2, 3…), после чего каждый индекс по определенному правилу кодируется последовательностью битов.

Принцип кодирования последовательности строкой индексов основан на следующих соображениях:

- последовательность *a* кодируется на основе DTD *a* пустой строкой ε
- последовательность *b* кодируется на основе DTD *a*|*b*|*c* индексом 1 (который обозначает позицию *b* в дизъюнкции *a*|*b*|*c*, начиная с 0)
- последовательность *ccc* кодируется на основе DTD *c** индексом 3 (который обозначает число повторений символа *c*)

Более формально, алгоритм кодирования последовательности s на основе данного DTD D описывается следующим образом.

Обозначим последовательность индексов, которая кодирует последовательность s на основе регулярного выражения D, как seq(D,s). Для вычисления seq(D,s) рекурсивно применяются правила 1-4:

1. seq(D,s) = ε если D = s (где ε – пустая строка). Это правило применяется в том случае, когда регулярное выражение D не содержит метасимволов, то есть является просто последовательностью символов из множества Σ.
2. $seq(D_0 \ldots D_k, s_0 \ldots s_k) = seq(D_0, s_0) \ldots seq(D_k, s_k)$. Это правило применяется в том случае, когда регулярное выражение D можно представить в виде объединения выражений $D_0 \ldots D_k$, а последовательность s – в виде объединения подпоследовательностей $s_0 \ldots s_k$ таким образом, что каждая подпоследовательность $s_i$ соответствует выражению $D_i$.
3. $seq(D_0|\ldots|D_m, s) = i\ seq(D_i, s)$. Это правило применяется в том случае, когда регулярное выражение D представляет собой дизъюнкцию выражений $D_0 \ldots D_m$, и последовательность s соответствует выражению $D_i$.
4. $seq(D^*, s_1 \ldots s_k) =$
    a. $k\ seq(D, s_1) \ldots seq(D, s_k)$, если $k > 0$
    b. 0, если $k = 0$

    Это правило применяется в том случае, когда последовательность s можно представить в виде объединения подпоследовательностей $s_1 \ldots s_k$ таким образом, что каждая подпоследовательность $s_i$ соответствует регулярному выражению D. Отметим, что кодирование операторов "+" и "?" осуществляется по этому же правилу: для "+" k всегда больше 0, для "?" k принимает только значения 0 и 1.

После того, как построена последовательность индексов, каждый индекс представляется в виде последовательности битов. Индексы 0 и 1 представляются битами 0 и 1 соответственно. Каждый индекс k > 1 представляется в виде последовательности битов по следующему правилу. Сначала вычисляется число двоичных разрядов, необходимых для представления данного индекса в двоичной форме: $]\log_2(k+1)[$. Затем строится последовательность длиной $2*]\log_2(k+1)[+1$, где первые $]\log_2(k+1)[$ символов – это единицы, число которых обозначает число двоичных разрядов, необходимых для представления данного индекса в двоичной форме; потом идет "0" в качестве разделителя; и затем – последовательность из $]\log_2(k+1)[$ нулей и единиц, представляющих уже сам индекс k в двоичной форме. Таким образом, 0↔0, 1↔1, 2↔11010, 3↔11011, 4↔1110100, 5↔1110101, 6↔1110110, 7↔1110111 и т. д.

Приведем пример, поясняющий, каким образом в системе XTRACT последовательность s кодируется на основе данного DTD сначала в виде последовательности индексов, а затем в виде последовательности битов.

**Пример 4.** Пусть DTD = (ab|c)*(de|fg*), s = abccabfggg. Тогда:
seq ((ab|c)*(de|fg*), abccabfggg) = (по правилу 2)
seq ((ab|c)*, abccab) seq ((de|fg*), fggg) = (по пр. 4)
4 seq ((ab|c), ab) seq ((ab|c), c) seq ((ab|c), c) seq ((ab|c), ab) seq ((de|fg*), fggg) = (по пр. 3)
4 0 seq (ab, ab) 1 seq (c, c) seq 1 (c, c) 0 seq (ab, ab) 1 seq (fg*, fggg) = (по пр. 1)
4 0 1 1 0 1 seq (fg*, fggg) = (по пр. 2, 1 и 4)
4 0 1 1 0 1 3 = (по правилу перевода индексов в последовательности битов)
1110100 0 1 1 0 1 11011 (пробелы поставлены для удобства читателя).

Отметим, что при выполнении действий по правилам 2, 3 и 4 необходимо решать следующие две задачи: (1) – определять, соответствуют ли друг другу данные последовательность s и DTD D, и (2) – разбивать последовательность s на подпоследовательности таким образом, чтобы каждая подпоследовательность описывалась соответствующей частью DTD D. В системе XTRACT для решения этих задач используются алгоритмы из [20], адаптированные соответствующим образом. При этом в системе XTRACT не реализовано рассмотрение и сравнение разных вариантов декомпозиции, хотя в общем случае возможны несколько альтернативных вариантов разбиения последовательности s и DTD D на соответствующие друг другу части.

Очевидно, что схема кодирования должна подразумевать возможность однозначного восстановления исходной последовательности, то есть выполнения декодирования: "последовательность битов" → "последовательность индексов" + DTD → "исходная последовательность". Нетрудно заметить, что предложенная в XTRACT схема кодирования допускает различные варианты восстановления последовательности индексов из последовательности битов (например, последовательность битов 11011 может означать как 11011 ↔ 3, так и 1 1 0 1 1 ↔ 1 1 0 1 1). Эту неоднозначность можно легко устранить простой модернизацией метода кодирования, которую мы опишем в главе 3.

### *2.5.2. Построение DTD с минимальной MDL-стоимостью*

Задача выбора из множества регулярных выражений $S_F$ такого подмножества S, которое покрывает все последовательности из исходного набора I и при этом имеет минимальную MDL-стоимость, может быть строго сформулирована следующим образом:

$$S = \min_{\substack{\text{по всем} \\ F \subset J}} \left( \sum_{\substack{\text{по всем} \\ j \in F}} c(j) + \sum_{\substack{\text{по всем} \\ i \in C}} \min_{\substack{\text{по всем} \\ j \in F}} [d(j,i)] \right)$$

Здесь введены следующие обозначения:

- J – множество регулярных выражений $S_F$;
- F – подмножество регулярных выражений из $S_F$;
- *j* – регулярное выражение из $S_F$;
- C – исходный набор последовательностей I;
- *i* – последовательность из I;
- *c(j)* – длина последовательности битов, кодирующей регулярное выражение *j*;
- *d(j,i)* – длина последовательности битов, кодирующей последовательность *i* на основе регулярного выражения *j* (если регулярное выражение *j* не покрывает последовательность *i*, то *d(j,i)* полагается равной ∞).

Алгоритмы решения такого рода задач достаточно хорошо исследованы в литературе [18], [19], мы не будем останавливаться на них подробно. Отметим лишь, что в системе XTRACT используется соответствующим образом адаптированный алгоритм из [18], временная сложность которого составляет $O(N^2 \cdot \log_2 N)$, где $N = | I |$. Итоговый DTD представляет собой дизъюнкцию всех регулярных выражений из S.

## 3. Модернизация существующих методов и алгоритмов

Описанные выше методы и алгоритмы построения DTD были использованы нами для построения собственной системы генерирования DTD для массива XML документов. При этом мы внесли в них ряд изменений и усовершенствований. Наиболее важные из этих изменений мы коротко опишем ниже. Описание усовершенствований технического характера (оптимизация структуры конкретных процедур и т. п.) можно найти в [21].

Во-первых, в модуле обобщения мы изменили процедуру DISCOVERSEQPATTERN. В системе XTRACT эта процедура заменяет подпоследовательность xxx…x на регулярное выражение (x)* . Очевидно, что такая замена некорректна: если строго следовать синтаксису DTD, то подпоследовательность xxx…x должна заменяться на регулярное выражение (x)+. Преобразование выражений вида (x)+ в выражения вида (x)* в нашей системе осуществляется на этапе факторизации: точно так же, как дизъюнкции вида (a|1) заменяются на a?, дизъюнкции вида (a+|1) заменяются на a*.

Во-вторых, в модуль обобщения мы добавили новую процедуру DISCOVERPLUSPATTERN. Дело в том, что в реальных DTD часто встречаются выражения вида $(a_1 \cdot a_2 \cdot \ldots a_n \cdot)+$, где "·" означает "?", "+" или "*" (так называемые "+"-выражения), но система XTRACT практически не способна распознавать их. Например, обработка последовательности abcabbaccabc в системе XTRACT даст регулярное выражение (a|b*|c*)*, а не более точное (ab*c*)+. Для генерирования "+"-выражений мы разработали алгоритм, который описан в разделе 3.1.

В-третьих, мы устранили неоднозначность в схеме кодирования системы XTRACT, описанную в пункте 2.5.2. Для того, чтобы обеспечить взаимно однозначное соответствие между последовательностью индексов и последовательностью битов, мы осуществляем кодирование индексов 0 и 1 по общему правилу – то есть, 0 кодируется в виде последовательности битов 100, 1 – в виде последовательности битов 101. Например, последовательность индексов 1 4 0 1 7 кодируется в нашей системе в виде

последовательности битов 101 1110100 100 101 1110111. Такой способ кодирования позволяет однозначно восстановить исходную последовательность индексов.

В-четвертых, в нашей системе мы реализовали возможность эмпирической настройки схемы кодирования. Пользователь может задавать весовые коэффициенты для метасимволов при вычислении MDL-стоимости DTD, а также устанавливать весовой коэффициент MDL-стоимости самого DTD (A) по отношению к MDL-стоимости кодирования (B). Дело в том, что при использовании схемы кодирования, предложенной в XTRACT, для небольших XML документов возникает тенденция к построению слишком громоздких и общих DTD. В нашей системе пользователь имеет возможность настраивать параметры схемы кодирования для получения DTD желаемого вида.

Забегая вперед, отметим: наши эксперименты показали, что для небольших XML документов качество DTD заметно улучшается при введении весового коэффициента 2 для метасимвола "|" (т. е. увеличении его стоимости в два раза), коэффициента 0,5 для метасимвола "?" (т. е. уменьшении его стоимости в два раза) и коэффициента 2 для MDL-стоимости DTD (т. е. при вычислении итоговой MDL стоимости DTD по формуле 2A+B, где A – стоимость самого DTD, а B – стоимость кодирования).

### 3.1. Генерирование "+"-выражений в системе DTDXtract

Очевидно, что для произвольной последовательности s, которая содержит n различных символов, можно построить n! различных "+"-выражений, каждое из которых будет соответствовать s. Например, для s=abaab (n=2) это будут "+"-выражения (a+b)+ и (b*a*)+, первое из которых позволяет восстановить s за 2 повторения (ab|aab), а второе – за три (a|baa|b). В системе DTDXtract мы ограничились построением одного "+"-выражения для каждой последовательности из I – такого, для которого число повторений минимально.

Принцип работы нашего алгоритма построения "+"-выражений мы опишем на примере последовательности abccddbaccbccddbac (n=4). Алгоритм состоит из двух этапов: сначала мы строим шаблон "+"-выражения, то есть конструкцию вида $(a_1 \cdot a_2 \cdot \ldots a_n \cdot)+$, после чего в этом шаблоне подставляем вместо точек соответствующие метасимволы (?, * или +).

Прежде всего, из исходной последовательности s мы исключаем все повторяющиеся символы (т. е. все подпоследовательности типа xx…x заменяем на x). Очевидно, что учитывать повторяющиеся символы для конструирования шаблона "+"-выражения не требуется. В нашем примере последовательность abccddbaccbccddbac будет приведена к виду abcdbacbcdbac. Затем мы выбираем из получившейся последовательности все комбинации длины n и n-1, которые не содержат повторяющихся символов. В нашем примере это будут комбинации: abcd, bcd, cdba, dbac, bac, acb, bcd, cdba, dbac. После этого все комбинаци длины n-1 мы дополняем до комбинаций длины n, приписывая справа "недостающий" символ (bac → bacd и т. д.). Таким образом, в нашем примере мы получим следующее множество комбинаций K: abcd, bcda, cdba, dbac, bacd, acbd, bcda, cdba, dbac.

Введем понятия "циклического сдвига" и "циклической группы". Циклическим сдвигом комбинации $k = a_1a_2\ldots a_i$ будем называть операцию разбиения комбинации k на две произвольные части и их перестановки (например, abcd → ab cd → cdab). Циклической группой комбинации k будем называть множество всех комбинаций, которые могут быть получены из k операцией циклического сдвига. Циклическую группу будем обозначать $\{k\}_ц$. Например, $\{abcd\}_ц$ = {abcd, bcda, cdab, dabc}.

На следующем этапе мы последовательно выбираем из множества комбинаций K все подмножества комбинаций, принадлежащих к определенной циклической группе. В

нашем примере эта процедура будет выполнена за три шага:

- abcd, bcda, cdba, dbac, bacd, acbd, bcda, cdba, dbac: подчеркнутые комбинации принадлежат группе $\{abcd\}_ц$
- cdba, dbac, bacd, acbd, cdba, dbac: подчеркнутые комбинации принадлежат группе $\{cdba\}_ц$
- acbd: принадлежит группе $\{acbd\}_ц$

Затем мы выбираем ту циклическую группу, комбинации из которой входят в множество K наибольшее число раз (при равном результате для нескольких циклических групп произвольным образом выбирается одна из них). В нашем примере будет выбрана циклическая группа $\{cdba\}_ц$. Наконец, из данной циклической группы мы выбираем ту комбинацию, которая начинается с того же символа, что и исходная последовательность s. Именно на основе этой комбинации мы и строим шаблон "+"-выражения для s. В нашем примере из циклической группы $\{cdba\}_ц$ будет выбрана комбинация acdb, которой соответствует шаблон "+"-выражения (a·c·d·b·)+.

Оценим стоимость нахождения такой комбинации. Очевидно, максимальная стоимость построения множества комбинаций K равна O(|s| - (n-1)), где |s| - длина последовательности, n - количество различных символов. Пусть число встретившихся циклических групп равно N (хотя теоретически для n различных символов возможно (n-1)! циклических групп, в реальных последовательностях число встретившихся циклических групп N ~ n). Таким образом, стоимость генерирования шаблона "+"-выражения равна O((|s| - (n - 1)) * N). В случае, когда |s| >> n, стоимость равна O(|s|*n). В случае, когда |s| ~ n, стоимость равна O(n).

Построение "+"-выражения на основе полученного шаблона выполняется по следующему алгоритму. Сначала исходная последовательность s разбивается на минимальное число подпоследовательностей, каждая из которых соответствует одному повторению шаблона "+"-выражения $(a_1 \cdot a_2 \cdot \ldots a_n \cdot)+$. В нашем примере: abccddbaccbccddbac → a__b|_ccddb|acc_b|_ccddb|ac__. Затем для каждого символа вычисляется число его повторений в каждой такой подпоследовательности. В нашем примере: a – (1, 0, 1, 0, 1), c – (0, 2, 2, 2, 1), d – (0, 2, 0, 2, 0), b – (1, 1, 1, 1, 0). Затем в шаблон "+"-выражения вместо точек подставляются соответствующие метасимволы по правилу:

- для случая, когда числа повторений – только 1: подставляется пустая строка;
- для случая, когда числа повторений – только 0 и 1: подставляется "?";
- для случая, когда числа повторений – 0, 1 и N>1: подставляется "*";
- для случая, когда числа повторений – только 1 и N>1: подставляется "+".

В нашем примере: для a и b – "?", для c и d – "*". Таким образом, выполнение процедуры DISCOVERPLUSPATTERN для последовательности abccddbaccbccddbac даст регулярное выражение (a?c*d*b?)+, которое наряду с другими будет добавлено в множество $S_G$.

## 4. Архитектура системы DTDXtract

Методы и алгоритмы генерирования DTD, описанные в предыдущих главах, были реализованы нами в системе DTDXtract. Эта система позволяет пользователю генерировать DTD для массива XML документов, сохраненных в РСУБД. Наша система состоит из четырех основных подсистем: модуля записи XML документов в РСУБД, реляционной базы данных, модуля генерирования DTD и интерфейса пользователя, рис. 6.

Модуль генерирования DTD включает в себя модуль генерирования DTD для атрибутов, модуль генерирования DTD для элементов со смешанным содержимым и модуль

генерирования DTD для элементов с содержимым из элементов. Последний состоит из модулей обобщения, факторизации и MDL, каждый из которых выполняет те же функции, что и в системе XTRACT, и построен по тем же принципам.

Описание архитектуры и алгоритмов работы модуля записи XML документов в РСУБД выходит за рамки данной статьи. Методы сохранения XML документов в РСУБД без использования DTD подробно исследованы в предыдущей работе авторов [22].

Интерфейс пользователя позволяет выбирать XML документы для обработки, а также задавать различные рабочие параметры модуля записи XML документов в РСУБД и модулей генерирования DTD, рис. 7.

В качестве РСУБД использовалась MySQL (версия 4.0.5a-beta-max) [23], установленная на рабочей станции SUN spark (512 Мб RAM) под ОС Solaris 2.7. Графический интерфейс был реализован на Java в среде Java 2 Standard Edition (версия 1.4.1_01) [24] на рабочей станции Pentium IV 2,4 ГГц (1024 Мб RAM) под ОС Windows 2000 (SP4).

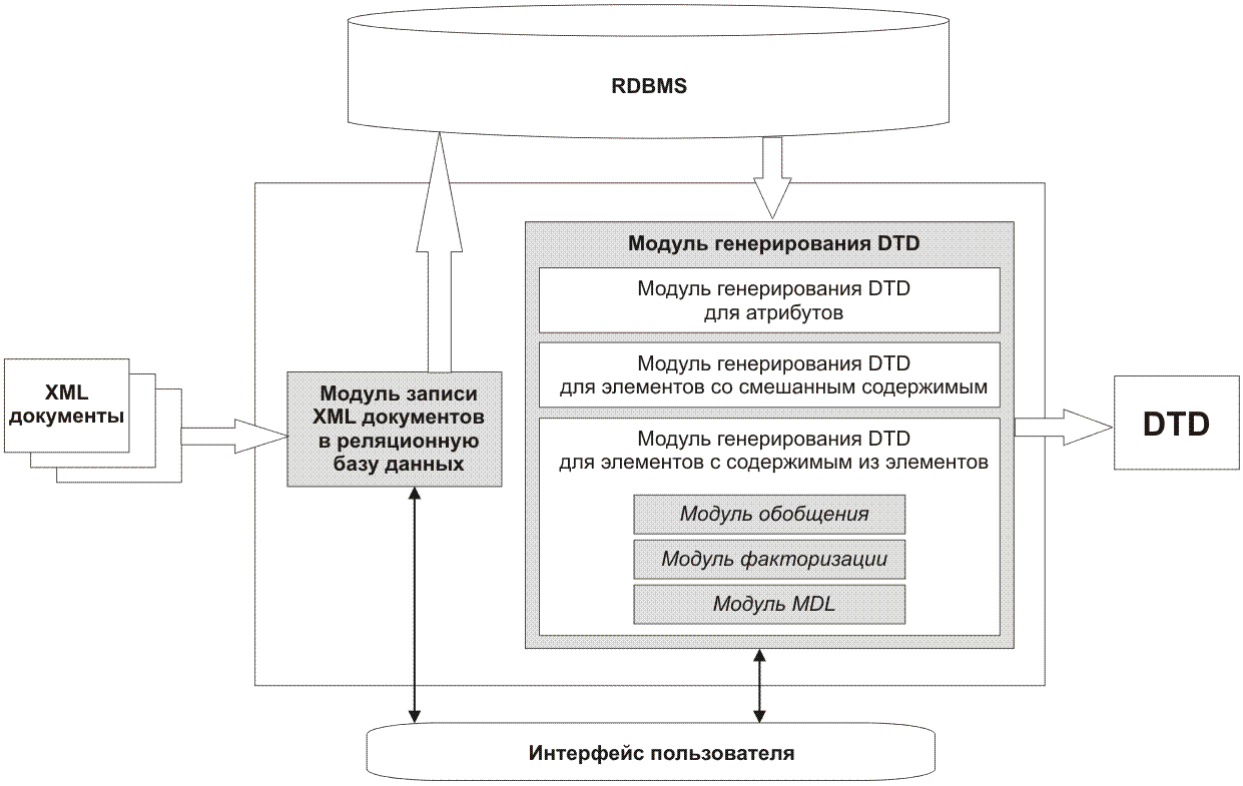


*Рис. 6. Архитектура системы DTDXtract.*

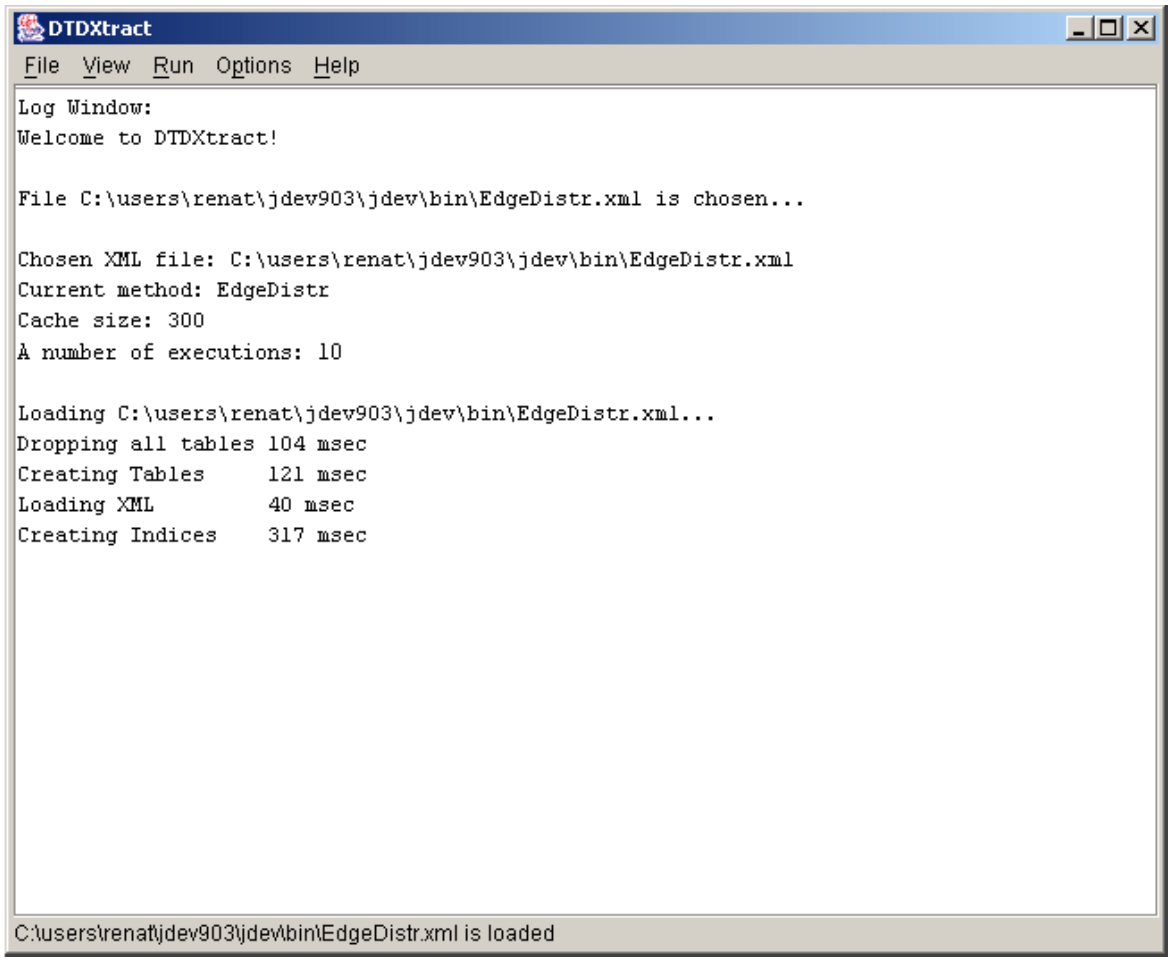


*Рис. 7. Интерфейс пользователя системы DTDXtract.*

Генерирование DTD для массива XML документов в системе DTDXtract осуществляется по следующему алгоритму:

1. Записать выбранные пользователем XML документы в РСУБД.
2. Сгенерировать DTD атрибутов для всех элементов.
3. Сгенерировать DTD элементов со смешанным содержимым.
4. Сгенерировать DTD элементов с содержимым из элементов.
5. Объединить DTD отдельных элементов и атрибутов в итоговый DTD.

Модуль генерирования DTD для атрибутов работает по следующему алгоритму:

1. Найти все атрибуты для данного экземпляра элемента.
2. Повторить шаг 1 для всех экземпляров данного элемента в массиве XML документов.
3. Для тех атрибутов, которые встретились во всех экземплярах данного элемента, сгенерировать объявления вида: Def = Attribute_name CDATA (#REQUIRED). Для всех остальных атрибутов сгенерировать объявления вида: Def = Attribute_name CDATA (#IMPLIED).
4. Сгенерировать обявление списка атрибутов (Attribute List Declaration, ALD) для данного элемента: <!ATTLIST Element_name $Def_1$ $Def_2$ … $Def_i$>
5. Повторить шаги 1-4 для всех элементов в массиве XML документов.

Модуль генерирования DTD для элементов со смешанным содержимым работает по следующему алгоритму:

1. Найти все дочерние элементы для данного экземпляра элемента со смешанным содержимым.

2. Повторить шаг 1 для всех экземпляров данного элемента в массиве XML документов.
3. Для тех элементов, у которых не было найдено ни одного дочернего элемента, сгенерировать объявление типа элемента (Element Type Declaration, ETD) вида: <!ELEMENT Element_name (#PCDATA)>.
4. Для тех элементов, у которых были найдены дочерние элементы $Child_1$, $Child_2$ … $Child_i$ сгенерировать объявление типа элемента (Element Type Declaration, ETD) вида: <!ELEMENT Element_name (#PCDATA | $Child_1$ | $Child_2$ |… |$Child_i$)*>.
5. Повторить шаги 1-4 для всех элементов со смешанным содержимым в массиве XML документов.

Модуль генерирования DTD для элементов с содержимым из элементов работает по следующему алгоритму:

1. Для данного экземпляра элемента с содержимым из элементов построить последовательность его дочерних элементов $s_i = c_1c_2…c_j$.
2. Повторить шаг 1 для всех экземпляров данного элемента в массиве XML документов (1…k).
3. Из множества последовательностей $s_1$, $s_2$ … $s_k$ исключить повторяющиеся последовательности.
4. Для полученного множества $I = \{s_1, s_2 … s_n\}$ сгенерировать DTD путем последовательного выполнения модулей обобщения, факторизации и MDL.
5. Повторить шаги 1-4 для всех элементов с содержимым из элементов в массиве XML документов.

## 5. Тестирование системы DTDXtract

Тестирование системы DTDXtract проводилось на массивах XML документов разного объема. В одних случаях массив XML документов имел сопровождающий DTD, и сгенерированный DTD сравнивался с оригинальным. В других случаях массив XML документов не имел сопровождающего DTD, и качество сгенерированного DTD оценивалось субъективно. Тестирование показало высокую эффективность системы DTDXtract и подтвердило ее способность генерировать краткие и точные DTD.

Результаты сравнительного тестирования систем DTDXtract, XTRACT и DDbE представлены в таблице 1. Видно, что процедура распознавания "+"-выражений, реализованная нами в системе DTDXtract, позволила повысить качество итоговых DTD по сравнению с системами XTRACT и DDbE.

| Оригинальный DTD | Система DDbE | Система XTRACT | Система DTDXtract |
|---|---|---|---|
| a\|b\|c\|d\|e | a\|b\|c\|d\|e | a\|b\|c\|d\|e | a\|b\|c\|d\|e |
| (a\|b\|c\|d\|e)* | (a\|b\|c\|d\|e)* | (a\|b\|c\|d\|e)* | (a\|b\|c\|d\|e)* |
| a*b?c?d? | - | a*b?c?d? | a*b?c?d? |
| (a(bc)+d)* | (a\|b\|c\|d)+ | (a(bc)*d)* | (a(bc)+d)* |
| (ab?c*d?)* | (a\|b\|c\|d)+ | - | (ab?c*d?)* |

*Таблица 1. Результаты сравнительного тестирования систем DTDXtract, XTRACT и DDbE (DTD представлены в упрощенном виде, т. е. реальные имена элементов заменены символами).*

## 6. Заключение

Автоматическое генерирование DTD для массива XML документов является одной из актуальных задач, решение которой необходимо для дальнейшего развития современных электронных информационных систем. Однако, построение эффективной системы генерирования DTD представляет собой достаточно сложную проблему. На данный момент на рынке фактически не существует доступных программных продуктов, которые обеспечивали бы приемлемое решение этой задачи. В литературе достаточно подробно описана лишь одна система генерирования DTD для массива XML документов – система XTRACT, созданная в качестве экспериментального образца в Bell Laboratories.

На примере архитектуры и принципов работы системы XTRACT авторы рассмотрели и описали общие принципы построения системы генерирования DTD для XML документов. Предложив ряд новых методов и алгоритмов для различных этапов генерирования DTD, авторы разработали собственную систему генерирования DTD для массива XML документов – DTDXtract. В настоящее время эта система реализована в качестве опытного образца и действует в экспериментальном режиме. Тестирование, в том числе сравнительное с системой XTRACT, показало высокую эффективность работы системы DTDXtract и возможность ее применения для решения прикладных задач.

Методы и алгоритмы, описанные в данной работе, являются основой для построения любой системы генерирования DTD для массива XML документов. Авторы надеются, что предложенная ими система станет платформой для дальнейших исследований и разработок в этом направлении. В качестве основного направления своей дальнейшей работы в этой области авторы рассматривают разработку комплексной системы для сохранения массива XML документов без соответствующего им DTD в РСУБД, автоматического генерирования DTD для этого массива документов и автоматического построения оптимальной реляционной схемы для его хранения в РСУБД.

## Литература


[1] Extensible Markup Language (XML) 1.0 (Third Edition): http://www.w3.org/TR/REC-xml

[2] Overview of SGML Resources: http://www.w3.org/MarkUp/SGML/

[3] AlphaWorks: Data Descriptors by Example: http://www.alphaworks.ibm.com/tech/DDbE

[4] Allora: http://www.hitsw.com/products_services/xml_platform/allora_dsheet.html

[5] XML Spy: http://www.xmlspy.com/

[6] SAXON: http://sourceforge.net/projects/saxon

[7] Minos Garofalakis, Aristides Gionis, Rajeev Rastogi, S. Seshadri, Kyuseok Shim. XTRACT: Learning Document Type Descriptors from XML Document Collections. *Data Mining and Knowledge Discovery, 7, 23-56, 2003.*

[8] Keith E. Shafer. Creating DTDs via the GB-Engine and Fred. *1995.*

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

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

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

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

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

[14] J. Rissanen. Modeling by Shortest Data Description. *Automatica, 14: 465-471, 1978.*

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

[16] R. K. Brayton and C. McMullen. The Decompozition and Factorization of Boolean Expressions. *Proc. Of the Intl. Symp. On Circuits and Systems, 1982.*

[17] A. R. R. Wang. Algorithms for Multi-Level Logic Optimization. *PhD Thesis, Univ. of California, Berkeley, 1989.*

[18] M. Charikar and S. Guha. Improved Combinatorial Algorithms for the Facility Location and K-median Problem. *Proc. of the Ann. Symp. on Foundations of Computer Science (FOCS), 1999.*

[19] D. S. Hochbaum. Heuristics for the Fixed Cost Median Problem. *Mathematical Programming, 22: 148-162, 1982.*

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

[21] Хуснутдинов Р. Р. Сохранение XML данных в RDBMS и генерирование DTD на их основе. *Диссертационная работа на соискание степени магистра прикладных математики и физики, МФТИ, 2004.*

[22] Леонов А. В., Хуснутдинов Р. Р. Построение оптимальной реляционной схемы для хранения XML документов в РСУБД без использования DTD / XML Schema. *"Программирование", 6, 2004 (находится в печати).*

[23] MySQL: Open Source Relational Database Management System: http://www.mysql.com.

[24] Java 2 Standard Edition: http://java.sun.com/j2se/