Department of Computer Science
Series of Publications A
Report A-1996-4

Generating Grammars for Structured Documents Using Grammatical Inference Methods

Helena Ahonen

University of Helsinki
Finland

Department of Computer Science
Series of Publications A
Report A-1996-4

Generating Grammars for Structured Documents Using Grammatical Inference Methods

Helena Ahonen

To be presented, with the permission of the Faculty of Science of the University of Helsinki, for public criticism in Auditorium, Teollisuuskatu 23, on November 29th, 1996, at 12 o'clock.

University of Helsinki
Finland

Contact information

Postal address:
Department of Computer Science
P.O.Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki
Finland

Email address: postmaster@cs.Helsinki.FI (Internet)

URL: http://www.cs.Helsinki.FI/

Telephone: +358 9 708 51

Telefax: +358 9 708 44441

ISSN 1238-8645
ISBN 951-45-7532-6
Computing Reviews (1991) Classification: I.2.6, I.7.2, F.4.3
Helsinki 1996
Helsinki University Printing House

Generating Grammars for Structured Documents Using Grammatical Inference Methods

Helena Ahonen

Department of Computer Science
P.O. Box 26, FIN-00014 University of Helsinki, Finland
Helena.Ahonen@cs.helsinki.fi, http://www.cs.helsinki.fi/~hahonen/

PhD Thesis, Series of Publications A, Report A-1996-4
Helsinki, November 1996, 107 pages
ISSN 1238-8645, ISBN 951-45-7532-6

Abstract

Dictionaries, user manuals, encyclopedias, and annual reports are typical examples of structured documents. Structured documents have an internal, usually hierarchical, organization that can be used, for instance, to help in retrieving information from the documents and in transforming documents into another form. The document structure is typically represented by a context-free or regular grammar. Many structured documents, however, lack the grammar: the structure of individual documents is known but the general structure of the document class is not available. Examples of this kind of documents include documents that have Standard Generalized Markup Language (SGML) tags but not a Document Type Definition (DTD).

In this thesis we present a technique for generating a grammar describing the structure of a given structured document instances. The technique is based on ideas from machine learning. It forms first finite-state automata describing the given instances completely. These automata are modified by considering certain context conditions; the modifications correspond to generalizing the underlying language. Finally, the automata are converted into regular expressions, which are then used to construct the grammar. Some refining operations are also presented that are necessary for generating a grammar for a large and complicated document. The technique has been implemented and it has been experimented using several document types.

Computing Reviews (1991) Categories and Subject Descriptors:
I.2.6 Artificial Intelligence: Learning
I.7.2 Text Processing: Document Preparation
F.4.3 Mathematical Logic and Formal Languages: Formal Languages

General Terms:
Learning, Algorithms, Theory, Experimentation

Additional Key Words and Phrases:
Document management, Grammar generation, SGML

Acknowledgements

First I would like to thank my supervisor, Professor Heikki Mannila for his endless patience and encouragement during my studies. Heikki really knows how to create an athmosphere where new ideas are freely born. With the help of his guidance and the gradually more demanding challenges he has given to me, I have been able to grow up as a scientist.

The Department of Computer Science at the University of Helsinki, headed by Professor Martti Tienari, has provided me excellent working conditions. Furthermore, I would like to direct my warmest thoughts to all the colleagues and friends, both at the department and elsewhere, who have made my life a lot easier with their care and friendship.

I am also deeply grateful to my parents, Anja and Eero, for their continuing interest on my studies and also for their financial aid that has helped me to concentrate on my work.

Finally, the financial support of the Academy of Finland, the Technology Development Center (TEKES), and the Helsinki Graduate School of Computer Science and Engineering is gratefully acknowledged.

Contents

Chapter 1

Introduction

In recent years, writing, storing, and retrieving documents in electronic form have become popular. Most of the non-fiction documents are somehow structured, i.e., they consist of parts that usually form a hierarchy. For instance, a scientific article consists of a title, an abstract, a text body, and references. The text body consists of a list of sections, every section consists of subsections or paragraphs, and so on. Other typical examples of structured documents are dictionaries, encyclopedias, user manuals, and annual reports. Recent surveys of the research concerning structured documents are [AFQ89a, AFQ89b, Qui89]. The interest in the area has led to the creation of several document standards, of which the best known are Open Document Architecture (ODA) and Standard Generalized Markup Language (SGML) [ODA89, SGM86, Jol89, Bar89, Bro89, Gol90a, vH94, BBKF96].

The common way to describe the structure of a set of similar documents is to use context-free grammars [GT87, BR84, CIV86, FQA88, QV86]. It is typical to use regular expressions on the right-hand sides of the productions of the grammar. For example, the following might describe the simplified structure of a dictionary entry:

$$\text{Entry} \rightarrow \text{Headword Sense}^*.$$

The meaning of this production is that an entry consists of a headword followed by zero or more senses. A more complicated example is

$$\begin{aligned}\text{Entry} \rightarrow\ & \text{Headword [Inflection]}\\ & (\text{Sense\_Number Description}\\ & \quad [\text{Parallel\_form} \mid \text{Preferred\_form}]\ \text{Example}^*)^*,\end{aligned}$$

which states that an entry consists of a headword followed by an optional inflection part and zero or more groups, each group consisting of a sense number, a description, a further optional part which is either a parallel form or a preferred form, and a sequence of zero or more examples.

Grammars can be used to facilitate transformations and queries that have structural conditions. The grammar also provides general knowledge of the text. It can be fairly complicated, however, to find a grammar that describes the structure of a given large text. A dictionary, for instance, may contain thousands of entries, and their structures may vary considerably.

Although it is difficult to construct a grammar for a set of documents, it is rather easy to name the parts of a document instance and construct simple productions from them. For instance, the structure of the following dictionary entry:

> *delta: 1.* a letter in the Greek alphabet. *2. (geographic)* land at the mouth of a river.

can be described with the simple productions:

> Entry → Headword Sense Sense
> Sense → Sense_number Description
> Sense → Sense_number Technical_field Description

Since the simple productions are based on some specific parts of the text, they are overly restrictive and hence, they cannot be used as the grammar describing the structure of the whole set of documents. Thus, one should be able to generalize the productions in some meaningful way.

For the generalization, we use techniques from machine learning [Mug90, Nat91] and formulate our problem as a grammatical inference problem [AS83]. The basic ideas of grammatical inference are presented in Chapter 4. The automatic generalization method we have developed proceeds as follows.

1. The document instances, e.g. tagged SGML documents, are parsed to form a set of simple productions.

2. The simple productions are transformed to a set of finite automata, one for each structural element. These automata accept exactly the right-hand sides of the given productions for the corresponding element.

3. Each automaton is modified in isolation, so that it accepts a larger language. This language is the smallest one that includes the original right-hand sides and has an additional property called (k, h)-contextuality. This property states roughly that in the structure of

the document what can follow a certain element is completely determined by the k preceding elements at the same level. Steps 1 and 2 are based on the synthesis of finite automata presented in [Ang82, Mug90], specifically (k, h)-contextuality is a modification of k-reversibility [Ang82] and k-contextuality [Mug90].

4. The automata are disambiguated until their language is unambiguous [BKW94].

5. The resulting automata are transformed to unambiguous regular expressions, which form the right-hand sides of the productions for the corresponding elements.

The rest of this thesis is organized as follows. First, in Chapter 2, we give the basic definitions and notations used throughout. Chapter 3 introduces the concepts of structured documents and SGML, and the problem of automatic generation of grammars. We also discuss the characteristics of various applications that can utilize generation of grammars. Chapter 4 describes grammar generating as a grammatical inference problem and presents some general solutions. Chapter 5 describes detailly our method of generalizing the right-hand sides of productions. We have implemented the method and experimented with several document types. Some results are presented and evaluated in Chapter 6. Finally, Chapter 7 contains some concluding remarks.

Chapter 2

Basic definitions

In this chapter we present the basic definitions and notations used in this thesis. The definitions are mostly based on [Sal69, HU79, Ang82, Woo87].

2.1 Sets, strings and languages

If S is any finite set, $|S|$ denotes the cardinality of S. An *alphabet* is a finite nonempty set of symbols. A *string* over an alphabet Σ consists of zero or more symbols of Σ. Symbol ϵ denotes the empty string that consists of zero symbols. The set of all finite strings over an alphabet Σ is denoted by Σ^*. Clearly, the set Σ^* is infinite. The length of a string w is denoted by $|w|$.

If u and v are strings over an alphabet Σ, then their *catenation* uv (or $u \cdot v$) is also a string over Σ. The string u is a *prefix* of the string v if and only if there exists a string w such that $uw = v$. Respectively, the string u is a *suffix* of the string v if and only if there exists a string w such that $wu = v$.

For a string u and an integer i, the notation u^i denotes the string obtained by catenating i copies of the word u. Similarly, Σ^i denotes the set of all strings over Σ with the length i. The set Σ^0 contains the empty string ϵ.

A *language* is any subset of Σ^*. The *catenation* of two languages L_1 and L_2 is defined to be the language

$$L_1 L_2 = \{uv \mid u \in L_1, v \in L_2\}.$$

Additionally, $L^i = \{u_1 \ldots u_i \mid u_j \in L, 1 \leq j \leq i\}$. The *closure* L^* of a language L is defined to be the language

$$L^* = \sum_{i=0}^{\infty} L^i,$$

where L^0 contains the empty string ϵ only.

If L is a language, we define the set $Pr(L)$ of prefixes of elements of L by

$$Pr(L) = \{u \mid \text{for some } v, uv \in L\}.$$

Also, for a language L, we denote by $T_L(w)$ the set of strings that can follow w in a member of L, i.e.,

$$T_L(w) = \{v \mid wv \in L\}.$$

2.2 Finite automata and regular expressions

Consider the auxiliary alphabet $\Sigma' = \{\,|\,, *, \emptyset, (,)\}$ and any alphabet Σ such that Σ and Σ' are disjoint. The regular expressions over the alphabet Σ are defined as follows:

1. $\emptyset$ is a regular expression.
2. ϵ is a regular expression.
3. For each symbol $a \in \Sigma$, a is a regular expression.
4. If E_1 and E_2 are regular expressions, then $(E_1|E_2)$, (E_1E_2), and (E_1^*) are regular expressions.
5. Nothing is a regular expression over Σ, unless its being so follows from a finite number of applications of the 1, 2, 3, and 4.

Each regular expression E over an alphabet Σ denotes a language $L(E)$ over Σ:

1. $L(\emptyset) = \emptyset$.
2. $L(\epsilon) = \{\epsilon\}$.
3. For all symbols $a \in \Sigma$, $L(a) = \{a\}$. In this case, we often denote $L(a) = a$, since there is no possibility for misunderstanding.
4. For all regular expressions E_1 and E_2 over Σ, $L(E_1 \mid E_2) = L(E_1) \cup L(E_2)$, $L(E_1E_2) = L(E_1)L(E_2)$, and $L(E_1^*) = (L(E_1))^*$.

A language L is *regular* if and only if there is a regular expression E such that $L = L(E)$. A string u is an *instance* of a regular expression E if u belongs to the regular language defined by E.

A *finite automaton* is a quintuple $(Q, \Sigma, \delta, I, F)$, where Q is a finite non-empty set of *states*, Σ is the set of *input symbols*, $\delta\colon Q \times \Sigma \to \mathcal{P}(Q)$ is the *transition function*, $I \subseteq Q$ is the set of *initial states* and $F \subseteq Q$ is the set of *final states*.

The transition function δ is defined as a mapping from $Q \times \Sigma$ to the set of all the subsets of Q. It can be extended to a function $\hat{\delta} : Q \times \Sigma^* \to Q$, where $\hat{\delta}(q, \epsilon) = \{q\}$, and $\hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a)$, with $q \in Q$, $x \in \Sigma^*$, and $a \in \Sigma$. From now on, the function δ is to be interpreted as the function $\hat{\delta}$ above whenever the second argument is a string. Additionally, we will abbreviate the set $\{\delta(q_0, a) \mid q_0 \in I\}$ with $\delta(I, a)$.

For an automaton $M = (Q, \Sigma, \delta, I, F)$, the language accepted by M is the set $\{u \mid \delta(I, u) \cap F \neq \emptyset\}$, and it is denoted by $L(M)$.

An automaton $M' = (Q', \Sigma, \delta', I', F')$ is a *subautomaton* of automaton M if and only if Q' and F' are subsets of Q and F, respectively, and for every $q' \in Q'$ and $b \in \Sigma$, $\delta'(q', b) \subseteq \delta(q', b)$.

A state q of M is called *useful* if and only if there exist strings u and v such that $q \in \delta(I, u)$, and $\delta(q, v)$ contains some element of F.

Let $M = (Q, \Sigma, \delta, I, F)$ and $M' = (Q', \Sigma, \delta', I', F')$ be automata. Automaton M is *isomorphic* to M' if and only if there exists a bijection h of Q onto Q' such that $h(I) = I'$, $h(F) = F'$, and for every $q \in Q$ and $b \in \Sigma$, $h(\delta(q, b)) = \delta'(h(q), b)$. Isomorphic automata are the same up to renaming of the states.

The automaton is *deterministic* if and only if there is at most one initial state, and for each state $q \in Q$ and symbol $a \in \Sigma$ there is at most one element in $\delta(q, a)$.

A finite automaton can be seen as a *directed graph* in which the vertices correspond to the states of the finite automaton. If there is a transition from state q to state p on an input symbol a, then there is an arc labelled a from state q to state p. States q and p are called the *source* and *goal* of the arc, respectively.

A *path* in a finite automaton $M = (Q, \Sigma, \delta, I, F)$ is a sequence $(p_1, p_2, \ldots, p_n)$ of states $p_i \in Q$, such that $\delta(p_1, a_1) = p_2$, $\delta(p_2, a_2) = p_3$, ..., $\delta(p_{n-1}, a_{n-1}) = p_n$ are transitions in M with $a_1, \ldots, a_{n-1} \in \Sigma$. The *length* of a path is the number of transitions on the path. A *cycle* is a path of length at least one that begins and ends at the same state.

2.3 Context-free grammars

A *context-free grammar* G is specified by a quadruple (N, Σ, P, I), where N is the nonterminal alphabet, Σ is the terminal alphabet, $P \subseteq N \times (N \cup \Sigma)^*$

is a finite set of *productions*, and I is a special nonterminal called the *initial symbol*.

Alphabets N and Σ are disjoint. Each production is of the form $A \to \beta$, where A is a nonterminal and β is a string over $N \cup \Sigma$.

If $A \to \beta$ is a production of P and α and γ are any strings in $(N \cup \Sigma)^*$, we say that $\alpha A \gamma$ *directly derives* $\alpha\beta\gamma$, and denote this by $\alpha A \gamma \Rightarrow \alpha\beta\gamma$. Suppose that $\alpha_1, \alpha_2, \ldots, \alpha_m$ are strings in $(N \cup \Sigma)^*$, $m \geq 1$, and $\alpha_1 \Rightarrow \alpha_2, \alpha_2 \Rightarrow \alpha_3, \ldots, \alpha_{m-1} \Rightarrow \alpha_m$. Then we say that α_1 *derives* α_m in grammar G, and denote this by $\alpha_1 \Rightarrow^* \alpha_m$. The language generated by G is denoted by $L(G)$, and it is defined as $\{w \mid w \in \Sigma^* \text{ and } I \Rightarrow^* w\}$.

A language $L \subseteq \Sigma^*$ is said to be a *context-free language* if there is a context-free grammar G with $L = L(G)$. Two context-free grammars G_1 and G_2 are *equivalent* if $L(G_1) = L(G_2)$, that is, they specify the same language.

A tree is a *derivation (or parse) tree* for a context-free grammar $G = (N, \Sigma, P, I)$ if:

1. Every node in the tree has a *label*, which is a symbol of $N \cup \Sigma \cup \{\epsilon\}$.
2. The label of the root is I.
3. If a node is interior and has a label A, then A must be in N.
4. If a node n has a label A and nodes $n_1, n_2, \ldots, n_k$ are the children of n, in order from the left, with labels $X_1, X_2, \ldots, X_k$, respectively, then $A \to X_1 X_2 \cdots X_k$ must be a production in P.
5. If a node n has a label ϵ, then n is a leaf and the only child of its parent.

In a context-free grammar $G = (N, \Sigma, P, I)$, each nonterminal $A \in N$ has a finite set of productions, and each right-hand side of a production is a string over the alphabet $N \cup \Sigma$. This can be extended by allowing the right-hand sides to be regular expressions over $N \cup \Sigma$, which corresponds to the situation where we have infinite number of productions with a string as the right-hand side. Hence, we allow each production to be of the form $A \to \beta$, where $A \in N$ and β is a regular expression over $N \cup \Sigma$. A production $A \to w$ is an *instance* of $A \to \beta$, if w belongs to the regular language defined by β. We call these grammars *extended context-free grammars*. For each extended context-free grammar G there is a context-free grammar G' such that $L(G) = L(G')$ [Woo87].

Chapter 3

Generating grammars for structured documents

In this chapter we introduce some basic concepts of structured documents (Section 3.1). We also formulate the problem of automatic generation of document type definitions and motivate the need for it (Section 3.2). Some related work is presented (Section 3.4) as well as an overview of our own method (Section 3.5).

3.1 Structured documents and SGML

Almost all documents have some kind of structural parts. For instance, a textbook contains chapters, sections, paragraphs, figures, tables, and so on. The structure of a document can be represented by distinguishing the parts with *markup*. Electronic document management is particularly useful when there are large amounts of documents. Then it is useful to group documents with similar properties and purposes into *document classes* and define constraints that the markup of all the documents of the class should satisfy. That is, these constraints specify the *markup language* for the document class. ISO standard *SGML (Standard Generalized Markup Language)* [SGM86, Gol90b, vH94] is a metalanguage that is used to define markup languages. SGML markup languages are *descriptive*: every part has a start tag and an end tag, and the name of the tag tries to describe the logical meaning of the part. *Procedural* markup, in the contrary, is concentrated on the processing of text: the documents may contain formatting commands such as "start bold" and "go to the next page". Descriptive markup makes multiple use of documents possible, since the markup is not designed for one particular purpose only. A fragment of SGML tagged text can be seen in Figure 3.1.

```
<Entry><Headword>kaame/a</Headword><Inflection>15</Inflection>
<Sense> kammottava, kamala, kauhea, karmea, hirveä </Sense>.
<Example_block><Example>Kaamea onnettomuus, verityö. </Example>
<Example> Tuliaseet tekivät kaameaa jälkeä. </Example>
<Example> Kertoa kaameita kummitusjuttuja.</Example></Example_block>
<Sense_structure> <Technical_field>Ark.</Technical_field>
<Example_block><Example>Kaamea hattu.</Example>
<Example> On kaamean kylmä.</Example> </Example_block>
</Sense_structure> </Entry>

<Entry><Headword>kaameasti</Headword> <Example_block>
<Example>Sireenit ulvoivat kaameasti.</Example></Example_block>
</Entry>

<Entry><Headword>kaameus</Headword><Inflection>40</Inflection>
<Example_block><Example> Sodan kaameus.</Example></Example_block>
</Entry>
```

Figure 3.1: Three dictionary entries tagged with SGML-tags.

In SGML the specification of a markup language, i.e., the grammar of it, is called a *document type definition* (DTD) and it specifies the logical parts, *elements*, that are permissible in a document of this type, and for each element, its *content model*, i.e., either the structure of its content in terms of the other elements, or, for the unstructured parts of text, the type of data that can occur in its content. Document types in SGML are defined by bracketed, extended context-free grammars [GH67], while the content models are essentially regular expressions.

3.1.1 Document type definition in SGML

We concentrate here on the parts of a document type definition that are crucial for our study, i.e., on the element declarations.

Elements

In Figure 3.2 we can see a part of a document type definition for the logical structure of a dictionary entry, with the element declarations for each of the logical elements. Each element declaration consists of the keyword ELEMENT followed by the element name. The content of an element is given either via a *content model* or *declared content*. Declared content types are used if, e.g., the element contains some markup-like characters which we do not want the parser to interpret as markup delimiters. There are three declared content types:

CDATA All markup delimiters are ignored, except the end tags.

RCDATA This type is similar to CDATA except that entity references (see below) are recognized.

EMPTY The element does not have any content, but the content is assumed to be generated during some processing stage. For instance, table of contents could be marked up by the empty <TOC> tag, so the processing system knows that the table of contents should be generated. Empty elements never have an end tag.

```
<!--      ELEMENTS         CONTENT                          -->
<!ELEMENT Entry            (Headword, Inflection?, Sense?,
                             Example_block, Sense_structure?)  >
<!ELEMENT Headword         (#PCDATA)                           >
<!ELEMENT Inflection       (#PCDATA)                           >
<!ELEMENT Sense            (#PCDATA)                           >
<!ELEMENT Example_block    (Example)*                          >
<!ELEMENT Sense_structure  (Technical_field, Example_block)    >
```

Figure 3.2: A document type definition for a dictionary entry.

A content model specifies the structure of an element in terms of other elements. If the element does not have any subelements, its content model contains the reserved word #PCDATA, standing for parsed character data. Another special case of a content model is the keyword ANY. It means that the element contains #PCDATA or any of the elements defined in the DTD, in any order. If the element is not #PCDATA or ANY, it is specified with a *model group*.

Definition 3.1 A model group is defined as follows.

1. for each element name e: (e) is a model group
2. if A and B are model groups, then (A, B), $(A \mid B)$, $(A\&B)$, (A^*), (A^+), and $(A?)$ are model groups.

Here the following *connectors* are used:

- "," : *sequence* connector; (A, B) means A followed by B.
- "&" : *and* connector; $(A\&B)$ means A followed by B or B followed by A.
- " | " : *or* connector; $(A \mid B)$ means A or B.

There are also *occurrence indicators* that specify how many times the element or model group can be repeated:

- "?" : *optional* element or model group; $(A?)$ means A does not occur or it occurs once.
- "+" : *required and repeatable* element; (A^+) means A occurs at least once.
- "*" : *optional and repeatable* element; (A^*) means A either does not occur, occurs once, or occurs many times.

□

If a content model contains both #PCDATA and subelements, it is declared to have *mixed content*.

Attributes and entities

An SGML document may also contain *attributes* and *entities*. An element can have one or more optional or mandatory *attributes* that provide further information about the element. The semantics of the attribute values is left for the applications. The following example illustrates a common use of attributes as identifiers that can be used to implement cross-reference links. In the following declaration an element *Entry* has an attribute *id* that gets as a value the headword, which is supposed to be unique within a dictionary. Another attribute tells the language of the entry.

```
<!ATTLIST Entry       id        ID                    #IMPLIED
                      language  CDATA                 #IMPLIED >
```

In a document instance the attribute values are included within the start tags:

```
<Entry id="kaamea" language="Finnish"><Headword>kaame/a</Headword>
<Inflection>...
```

Entities are named parts of the content that are not dependent on the element declarations. For instance, an external image file could be an entity that is represented in the contents of the document by an *entity reference*. The entities are also used as a shorthand notation if we want to avoid repeating a text string. If we define the following entity "SGML",

```
<!ENTITY SGML "Standard Generalized Markup Language">
```

we can put the entity reference "&SGML;" anywhere we want to produce the text "Standard Generalized Markup Language":

```
As we know &SGML; is an ISO standard.
```

In addition to the *general entities* described above there are *parameter entities* that can be used in DTDs to name frequently occurring substructures that can then be used in the element declarations of many elements. For instance, we can define an entity "addr" that collects several types of addresses:

```
<!ENTITY % addr  "(street|city|state|country|postcode|san|email|
                   postbox|phone|fax)*"
```

The parameter entity reference "%addr" can now be shared by the content models:

```
<!ELEMENT author        (fname, surname, %addr)        >
<!ELEMENT publisher     (pubname, organization, %addr)  >
```

Exceptions

The content models of the elements can be extended with *exceptions* — *inclusions* and *exclusions*.

Some elements of a document may be floating, i.e., they do not have a specific context but can occur within and between many other elements. Typical examples of floating elements are citations, index entries, table of contents entries, and floating figures or tables. It would be tedious to include these floating elements in each element declaration, since they would have to be mentioned in almost every position of the definition of every model group. Instead, the *inclusion* parameter can be used. For instance, in the following a textbook may contain references to figures (*figref*) anywhere:

```
<!--       ELEMENTS     CONTENT                (EXCEPTIONS)?   -->
<!ELEMENT  textbook     (front?, body, read?)  +(figref)>
```

As the same kind of elements, e.g., paragraphs, are used as building blocks in many situations, it may be desirable to restrict their structure in some of these contexts. For instance, according to the following declaration, a figure may contain paragraphs and on the other hand a paragraph may contain figures, but we may not want a figure containing another figure. Hence, we add an exclusion *-(fig)* to prevent this.

```
<!--      ELEMENTS  CONTENT                 (EXCEPTIONS)?-->
<!ELEMENT fig       (figbody, figcap?)       -(fig)       >
<!ELEMENT figbody   (artwork | paragraph+)                >
<!ELEMENT paragraph (#PCDATA | fig)                       >
```

Both inclusions and exclusions are in effect at all the levels, i.e., an included element can be a part of the element in the declaration of which it is defined, or a part of a part of that element, and so on. Hence, an inclusion often needs an exclusion, if we want to prevent an inclusion from containing itself.

Ambiguity

The SGML standard requires that the content models have to be unambiguous in the following sense. A content model is ambiguous if an element appearing in the document instance can be matched with more than one occurrence of the corresponding element in the content model without look-ahead, i.e., without scanning the text ahead to decide which occurrence should be chosen. For instance, if we have the following element declaration

```
<!ELEMENT Inflections     (Infl_index?, Infl_index) >
```

and part of a document instance

```
     <Inflections>
     <Infl_index>12</Infl_index>
     </Inflections>
```

it is impossible to say whether 12 is an instance of the first *Infl_index* or the second.

There can also be ambiguity in choosing between alternatives. In the following, when seeing

```
<Headword>tact
```

in the text we cannot say which alternative should be chosen.

```
<!ELEMENT Entry        ((Headword, Sense) | (Headword, Example)) >

     <Entry>
     <Headword>tact</Headword>
     <Example>Phil had the tact to leave a moment's
              respectful silence.</Example>
     </Entry>
```

Iterations are still another cause of ambiguity:

```
<!ELEMENT Senses      ((Sense, Example)*, Sense) >

     <Senses>
     <Sense>Extremely unpleasant.</Sense>
     <Example>She would never harm an insect,
                  however noxious</Example>
     <Sense>A noxious gas or substance is
                  harmful or poisonous.</Sense>
     </Senses>
```

When seeing

```
<Sense> Extremely
```

we do not know if we are starting a new iteration or whether we already are in the second occurrence of *Sense*. To be valid, a DTD should avoid all the structures shown above. Unambiguity is further discussed in Section 5.2 below.

Minimization

Earlier, when no SGML (WYSIWYG) editors were available, it was reasonable to try to reduce the work needed for creating tags. Hence, minimization rules that allow leaving out certain tags were introduced into the standard.

Minimization is noted in the element declaration with the signs "–"(no minimization) and "O"(the tag can be omitted). The first position is for the start tag and the second one for the end tag. For instance, the following declaration allows a list item lack an end tag.

```
<!ELEMENT list        - - (list_item)* >
<!ELEMENT list_item   - O (#PCDATA)    >
```

Hence, we can write the following list:

```
<list>
   <list_item>electronic publishing
   <list_item>document management
   <list_item>hypermedia
</list>
```

3.2 Automatic generation of content models

Traditionally DTD creation has been seen as a similar effort as database design: A group of specialists defines a DTD, possibly in cooperation with the users, and after that the users start writing the documents. This idealistic view is, however, too restricted. There are huge amounts of existing documents that can be utilized with the help of SGML, and there are also various other cases where we obtain a set of documents but do not know the DTD they conform.

The problem we consider in this thesis is: given a set of tagged document instances, how to generate a DTD for this set. There are at least three trivial solutions for building the element declarations:

1. Find each element within the document, and allow it any content.

```
<!ELEMENT Entry                  ANY >
<!ELEMENT Headword               ANY >
<!ELEMENT Inflection             ANY >
<!ELEMENT Sense                  ANY >
<!ELEMENT Example_block          ANY >
<!ELEMENT Example                ANY >
<!ELEMENT Technical_field        ANY >
<!ELEMENT Sense_structure        ANY >
```

2. For each element find all the elements that appear within it, and form an optional and repeatable model group.

```
<!ELEMENT Entry                  (Headword | Inflection
                                  | Sense | Example_block
                                  | Sense_structure)*>
<!ELEMENT Sense_structure        (Technical_field |
                                   Example_block)*>
<!ELEMENT Example_block          (Example)*>
<!ELEMENT Headword               #PCDATA  >
<!ELEMENT Inflection             #PCDATA  >
   ...
```

3. Take all the structures of the instances to be the DTD. This solution is also called the *de-facto grammar* [Che91].

```
<!ELEMENT Entry             ((Headword, Inflection, Sense,
                               Example_block,
                                Sense_structure) |
                             (Headword, Example_block) |
                             (Headword, Inflection,
                               Example_block)) >
<!ELEMENT Sense_structure   (Technical_field, Example_block)>
<!ELEMENT Example_block     ((Example, Example, Example)
                             | (Example, Example) | Example)>
```

Solutions 1 and 2 are usually overgeneralizing, while solution 3 is too restrictive. A non-trivial solution should find a good compromise between these two extremes and capture at least some knowledge of the order of the elements, and whether the elements are optional, required, or iterating, like in the following.

```
<!ELEMENT Entry             (Headword, Inflection?, Sense?,
                             Example_block, Sense_structure?) >
<!ELEMENT Sense_structure   (Technical_field, Example_block)  >
<!ELEMENT Example_block     (Example)* >
```

Of course, we can ask whether we need non-trivial DTDs in the first place, since tagged documents may be used even without any DTD. For instance, we can make structured queries that just specify a string inside some element. However, it is certainly difficult for a user to get the most benefit from the structure, if s/he does not even know it. Many application programs, like editors and transformation tools, require a DTD. A trivial DTD may not be satisfying, if we, e.g., have strong validation needs. In the next section we discuss further the ways in which we can utilize automatical DTD generation.

3.3 Applying automatic DTD generation

3.3.1 Characteristics of applications

Automatic DTD generation can be used for many purposes. The actual needs of the application determine how the generation has to be applied, and what kind of result we can expect. The needs can be examined along the following dimensions:

Complexity of the structure. If the structure of the document instances is simple, automatic DTD generation is easy and the quality of the result is good, i.e., we can easily achieve all the requirements stated below. Hence, in the following, we assume that the structure is somehow complex.

Time frame. We may have online application that requires immediate result. Sometimes we need the result quickly, and sometimes. e.g., in a design process, we have more time.

Completeness. The result either has to be a valid SGML DTD that is immediately usable with an SGML application, or the result is used as a source of information and hence a partial solution can also be useful.

Richness of structure. Some applications just need some DTD, thus a trivial solution that loses most of the structure may be appropriate. If a trivial DTD is not adequate, there is usually some tendency how much trivialization is preferred. If DTD generation is used mainly to gather some information of the documents, the more structure can be discovered the better.

Readability. The resulting DTD may be used directly by some application program, hence the machine-readable form is adequate. Instead, if the DTD has to be understandable also by human users there are different requirements for readability.

Tagging can/cannot be modified. The source instances may be "given as such", i.e., there is either a fixed structure, or there is no time to modify the structure. On the other hand, for instance in a design process, the DTD generation can be used to reveal problems with the instance structures and tagging, which may be then updated according to the results.

Batch or interactive processing. Usually there is always first some kind of batch processing, which is followed by an interactive session, if there is a need for it and sufficient resources.

In addition to the interactions already mentioned, these dimensions affect each other in several ways. The most important of these is: if the DTD has to be complete, and the structure is complex, we cannot get both full rich structure and readability, i.e., there has to be some trivialization. This trade off is even emphasized, if the result is needed online, whereupon the user cannot simplify the result interactively.

In addition to the first case mentioned, i.e., when the structure is simple, there are at least two cases where the other dimensions do not affect the result. Clearly, if some trivial solution is adequate, we can always form it quickly, it is complete and readable, and there is no need to change the structure of instances. Similarly, if we do not require completeness, some sort of a partial solution should be easy to generate. In the following sections we present applications that illustrate some combinations of the above-mentioned dimensions.

3.3.2 Examples of applications

DTDs for static documents

Some document collections are "given as such", e.g., archives may have a very conventional structure, and it is not possible to modify it. Hence there is no need for a design process in the usual sense, but the task is to find a DTD that makes it possible to use the documents in the best possible way with SGML application programs. Moreover, with this kind of "historical" documents, which were never designed with any grammar in mind, it may be very interesting to see the exact structure found in the documents. The lifespan of such documents may have been very long, even decades, which has made the consistency of the structure difficult to maintain. Hence, the discovery of the structure both has a meaning as such and gives extra information for various processing needs.

The second case of static documents are documents that come from some external source, e.g., messages imported to a company from outside. We may want to transform these messages to our own format, and for the transformation we need some DTD for the source documents.

Interactive design tool

Above we considered static documents. The situation is different when we are designing our own document collection, e.g., all the documentation needed in our enterprise. In this case DTD creation is more or less similar to database design: modelling of the application area is needed. Automatic DTD generation can be used as a design tool. If the planned documentation is based on existing documents, we can first gather information from their structure and use it for evaluation and further design.

Assume we have used some non-SGML word processor for our documentation, and the writers have been obliged to use some common styles in a rigid way. Whenever we want to convert the whole documentation to

SGML, we might hope to get the DTD directly from the style rules. However, people do not necessarily obey the rules very strictly, since there is no way to make them obligatory. Therefore, when the documents are converted into SGML, it is useful to check what actually is the real structure. All the deviations of the rules might not be totally negative: there might be ways the people have created to overcome some in the design process unexpected situations. Hence, the automatic DTD generation can reveal useful knowledge that can benefit the design of the new SGML documentation.

DTDs for different views and subdocuments

It is often wise to use some standard DTD. However, these DTDs are usually large and designed to cover many varying cases. If we then need a simpler DTD for some task, or a DTD for some subdocument, we can use automatic generation to find a DTD that accepts the selected documents. One example of this kind of DTD is an author DTD [MA96]: a DTD that is given to the people who create documents. For instance, in our research project [AHH+96] we have used ISO 12083 standard DTD for books [ISO94] as a DTD for engineering text books. By now we have converted existing non-SGML books to this DTD, but since the conversion process is very tedious, the authors are recommended to use an SGML editor to create the new books. Hence, we need now a simpler DTD that the writers can use. We can utilize the already converted tagged textbooks to generate a DTD automatically, and check the result against the standard DTD to make the obvious generalizations, like allowing more than one author.

Online DTDs

We have studied in our research project [AHH+96] *intelligent assembly* of documents, which means that the user can configure new, individualized documents from a collection of documents and possibly also from external information sources. In document assembly, the basic operations are *queries* that return the selected fragments of documents, and *configuration* of these fragments to form sensible documents. Hence, the assembled document is a transformation of one or more existing documents. If, instead of simple printing of documents, we want that the resulting documents are valid SGML documents that can be reused later, possibly in assembling new documents, we need a complete DTD for them. These *online DTDs* have to be generated automatically. Of course, we can always use some trivial DTD, but if we want to get the maximal use of the assembled document, it is better to find a solution that is more structured.

We can form a DTD from the tagging of the assembled document, but this DTD may be too restricted. Hence, we are going to study how the source DTD and the transformations can be used to generalize the target DTD in a suitable way.

3.4 Related work

There have been several attempts to generate grammars automatically, mostly in practical contexts [Che91, FX94, Sol94, Sha95].

Online Computer Library Center (OCLC) receives several tagged data sources for its reference databases. While this tagged text appears to be SGML, it does not always have a DTD. Despite this, OCLC must build data transformations, databases, and interfaces for this tagged text. Therefore they have built FRED Grammar Builder [Sha95] to generate the necessary DTDs. The generation is intended to be run mainly in a batch processing. OCLC has even offered a free online service for anyone who needs to find a DTD for some documents.

The primary goal of Fankhauser and Xu [FX94] is to recognize the logical structure of untagged electronic documents in order to transform them into structured SGML. The documents are mainly intended to be publicly available electronic information sources, such as public databases, bulletin boards, and electronic mail. The idea is to recognize unstructured documents, mark them up semi-automatically, and incrementally construct a grammar for the documents. First, the user marks up some examples, i.e., names some parts of document instances. Second, the initial grammar is constructed based on these examples, and the grammar is used to parse new examples. If some document cannot be parsed, the user completes the structuring of this document, and the grammar is modified to accept the new structure. The modification contains the *unification* part and the *abstraction* part. In the unification part the grammar is extended to accept the new structure, and in the abstraction part the grammar is heuristically generalized so that it accepts slightly broader class of documents. Both parts contain a set of rules that are used for the unification and abstraction, respectively.

The basic goal of Solstrand [Sol94] is to find DTDs for documents in the Wittgenstein archive in Bergen, Norway. The documents were originally marked up with particular coding, and also the methods to convert the codes into SGML markup were available. Only the DTDs were missing. This application is a typical example of a static document collection, the structure of which cannot be modified. Solstrand has also noted that the researchers of Wittgenstein have pointed a remarkable interest in the description of the underlying structure.

All the approaches above start from the de-facto grammar and combine the structures of each element to form a content model. Also the basic idea is common: there is a set of more or less heuristic rules that are applied. The actual collection of rules varies, but the following features appear frequently:

- Consecutive similar elements are combined with operator $^+$, i.e., $(ab^n c)$ is transformed to (ab^+c), if n is greater than or equal to a given value, usually $n = 2$.
- Similar parts of the alternatives are combined. If the parts are not identical, the content model is generalized.
- Since mixed content may cause ambiguity, #PCDATA is recommended to occur only in a model group in which *Or connector* is the only connector. Hence, for instance, the structure $(ab| \#PCDATA\ b)$ is transformed to $((a \mid b \mid \ \#PCDATA\ )^+)$
- The content models are simplified, for example by removing unnecessary parentheses.

The main part of the generation is to find the similar parts and combine them. The degree of similarity in the approaches varies. The parts may be required to be identical, or there can be a definition of similarity or subsuming relation.

For instance, Chen [Che91] defines a rule

$$ab(op)c \mid ab(op')c \rightarrow ab(op'')c,$$

where op and op' can be "·", "?", "+", or "*". The resulting operator op'' is defined by the *least common operator function* LC, i.e. $op'' = LC(op, op')$: $LC($"*",$op) =$ "*", $LC($"·", $op) = op$, $LC($"+", "?"$) =$"*", $LC($"+","·"$) =$ "+", and $LC($"?","·"$) =$"?". Fankhauser and Xu [FX94] use the partial ordering on regular expressions and Solstrand [Sol94] gives a certain similarity definition. Fred Grammar Builder [Sha95] contains the following heuristics:

- "Identical Bases": combines all subrules that have identical base elements.
  E.g. $((ab)^* \mid (a?b?) \mid ab)$ is transformed to $((a?b?)^*)$.

- "Off by One": finds all subrules that differ by at most one place.
  E.g. $(abc \mid ac \mid ab)$ is transformed to $(ab?c \mid abc?)$.

- "Redundant": removes obvious duplicates based on subsuming rules for repetitions and equality of atoms.
  E.g. $(ab \mid ab^* \mid ab \mid c)$ is transformed to $(ab^* \mid c)$.

3.5 Overview of the method

Our research goal was to find a general method that can also handle complex structures and that is flexible enough to be adjusted to the specific needs of various applications. As DTDs are extended context-free grammars and content models are regular expressions, we also wanted to study how the general methods of formal language inference would apply to the generation of content models.

Our method proceeds as follows, given as input a set of document instances. First, a parser picks up the structures of each element and forms a set of simple rules, i.e., a de-facto grammar. For instance, the dictionary entries in Figure 3.1 would produce the following set of rules:

Entry → Headword Inflection Sense Example_block Sense_structure
Entry → Headword Example_block
Entry → Headword Inflection Example_block
Sense_structure → Technical_field Example_block
Example_block → Example Example Example
Example_block → Example Example
Example_block → Example

Second, Algorithm 3.2 is applied to the set.

Algorithm 3.2 Automatic generation of a content model

Input: A set I of simple rules describing the structures of instances.
Output: A content model.
Method:
1. **for** each element e on the left-hand side of some rule in I
2. construct a finite-state automaton that accepts the structures of e;
3. generalize the automaton by merging states;
4. disambiguate the automaton;
5. convert the automaton into a content model for e;

Generalization of automata applies *grammatical inference* methods [Ang82, Mug90]. Some background and general solutions of grammatical inference are presented in the next chapter. The solutions used in the method are detailly described in Chapter 5.

Algorithm 3.2 produces the first candidate for the DTD. If it does not fulfill the requirements of the application, and if some further processing is possible, the method contains several *refining operations* that can be applied to the result. These operations are presented in Section 5.3.

The method does not cover all of the SGML standard. It produces element declarations with the following characteristics. The content models contain element names and operators. Operators are ",", " | ", and "*". Operator "+" could be easily added, whereas utilizing operator "&" would need more fundamental alterations in the method. All text portions of the sample instances are replaced by #PCDATA. An element never receives a declared content, particularly, we do not attempt to discover which elements may have an empty content.

Additionally, resulting content models may contain parameter entity references that abbreviate frequent model groups. Respective entities are included in the DTD. Attributes found in the sample instances are not utilized, although it would be quite straightforward to list all the attributes occurred in an element and the ranges of values for them.

As for learning of exceptions, inclusions may be noticed, and actually finding them is quite necessary to produce a readable DTD. Instead, missing elements can hardly be used for learning, so our method cannot find exclusions.

The tagging used in automatic DTD generating need not be fully complete. A recursive parser can easily handle missing end tags, if the corresponding elements have no subelements, i.e., the end tag is assumed to be before the next tag seen. However, since the minimization feature is not very useful currently, when SGML editors are available, we do not try to learn any minimization used in tagging to include in the element declarations. Instead, all declarations have "– –" as minimization notation.

No other features of SGML are produced or utilized in the generation.

Chapter 4

Grammatical inference problem

Our problem is to infer a document type definition for a document class using specific examples of document structures. The examples can look like the following:

Entry → Headword Sense Sense
Sense → Sense_number Description
Sense → Sense_number Technical_field Description
Headword → *delta*
Sense_number → *1.*
Sense_number → *2.*
Description → *eräs kreikk. kirjaimiston aakkonen.*
Description → *suisto(maa).*
Technical_field → *maant.*

Document type definitions are, as mentioned above, essentially context-free grammars. Thus our problem can be formulated as a *grammatical inference problem*. Grammatical inference is a part of machine learning that uses inductive inference method to learn grammars and their equivalent representations (e.g., regular grammars, deterministic finite automata, or context-free grammars) from given examples. In this chapter we show how grammatical inference problems can be specified (Section 4.1) and present some approaches for inferencing context-free languages (Section 4.2) and regular languages (Section 4.3).

4.1 Specification of grammatical inference problem

A grammatical inference problem can be specified by giving the following items [AS83]:

1. the class of languages,
2. the hypothesis space, i.e., a set of representations for the languages,
3. for each language, its set of positive and negative examples, and the admissible sequences of examples,
4. the class of inference methods,
5. the criteria for a successful inference.

Assume, for instance, that we want to infer a regular language from positive and negative examples. The class of languages is regular sets over the alphabet, say $\{0, 1\}$, and the hypothesis space could be either deterministic finite automata, nondeterministic finite automata, regular expressions, or regular grammars over the same alphabet. An example of a regular language L is of the form $\langle s, d\rangle$, where d is "yes" if string s belongs to L and "no" otherwise. An admissible sequence of examples contains every string over the alphabet at least once. An inference method is any method representable by a computer program that takes a finite initial sequence of examples of any regular language as input, always halts, and produces as output a representation from the chosen hypothesis space. The most important criterion of success is *identification in the limit*, which is defined as follows [Gol67]:

Definition 4.1 Method M *identifies* language L *in the limit* if, after a finite number of examples, M makes a correct guess and does not alter its guess thereafter. A class of languages is identifiable in the limit if there is a method M such that given any language of the class and given any admissible example sequence for this language, M identifies the language in the limit. □

A large amount of inference methods has been introduced in the literature. In the following we consider some of them.

4.2 Inference of context-free languages

The trivial inference method is called *identification by enumeration* [AS83]. It systematically searches the hypothesis space to find a representation that is consistent with all the examples seen so far. In order that the method be computable, the enumeration of hypotheses has to be computable, and it must be possible to decide whether a given representation is consistent with the given set of examples. The enumeration method is clearly rather impractical, since the size of the space that must be searched is typically exponential in the length of the representation of the correct guess.

One inference method that avoids exhaustive search and backtracking necessary in identification by enumeration is the *version space strategy*. Mitchell [Mit77] defines the *version space* to be a set of all generalizations that are consistent with the sample. In case of grammars, and if there are positive examples only, the version space contains all the grammars that generate the sample strings.

The version space strategy requires that it is possible to define a partial general-to-specific ordering in the hypothesis space. A version space can be represented — instead of explicitly storing all the generalizations — by the sets of *maximally general* and *maximally specific versions*. A version, i.e., a hypothesis, belongs to a version space if and only if it is less general than or equal to one of the maximally general versions, and less specific than or equal to one of the maximally specific versions.

The strategy operates on the version space of all plausible hypotheses at each step. It begins with the version space consistent with the first positive example, and eliminates hypotheses that are in conflict with subsequent examples. Any positive example may force some element of the set of maximally specific versions to become more general. Respectively any negative example may force some element of maximally general versions to become more specific. It is impossible to make any maximally general version more general, or any maximally specific version more specific, since they would not be consistent with the previous examples any more. Making a version more general (respectively more specific) is done by replacing a version with a version that is maximally specific (minimally general) of all those versions that generalize (specify) the old version.

At any moment the version space represents all the hypotheses that are consistent with the examples seen so far. If the sets of maximally general and maximally specific versions become equal, no further examples can change the final solution. This success criterion is a great advantage of the strategy.

The version space strategy cannot be applied to grammatical inference as such [VB87]. First, if the version space contains all the grammars that generate the sample strings, the version space is certainly infinite. An infinite version space would not be a problem for the version space strategy, if the boundaries, i.e., the sets of maximally general and maximally specific versions, were finite. Unfortunately, this is not the case: also the boundaries are infinite. Second, checking whether some grammar G generalizes another grammar G' is equivalent to testing whether the language generated by G includes the language generated by G'. This problem is known to be undecidable for context-free grammars [HU79]. To reduce the complexity of the problem [VB87] defines restrictions for context-free grammars. These restrictions, which state that the grammars have to be *simple* and *reduced*, are quite natural: they exclude grammars that also "common-sense" would exclude, for instance grammars that have nonterminals not used to derive any string.

Definition 4.2 A context-free grammar is *simple* if

1. no rule has the empty string as the right-hand side,
2. if a rule has just one symbol on its right-hand side, then the symbol is a terminal, and
3. every nonterminal appears in a derivation of some string.

□

Definition 4.3 Given a sample set S, a grammar is *reduced* if it is consistent with S and there is no proper subset of its rules that is consistent with S. □

Given a finite sample, there are finitely many reduced simple context-free grammars consistent with the sample. Hence, the reduced sets of maximally general and maximally specific versions are also finite. This solves the first problem. However, partial ordering of grammars is still undecidable. To solve the second problem [VB87] defines a concept of *derivational version space* that is a superset of the reduced version space and a subset of the version space. In the following definition a *simple derivation tree* is a derivation tree produced by a simple grammar, i.e., if a node in a simple derivation tree has a single child, that child must be a terminal. Hence, a simple derivation tree never contains long, unbranching chains, and, consequently, there are only finitely many unlabelled simple derivation trees for any given string.

Definition 4.4 Given a set of positive strings, the *derivational version space* is the set of grammars corresponding to all possible labelings of each set of simple derivation trees for those strings. Given a set of positive and negative strings, the derivational version space is the derivational version space for the positive strings minus those grammars that generate any of the negative strings. □

The derivational version space is finite. Additionally, [VB87] defines a predicate that implements the partial ordering in derivational version space. The inference method described constructs the set of unlabelled simple derivation trees for the sample. The method then considers all derivations and all labelings, and gives as a solution all grammars consistent with the sample.

Since the method produces as a solution all grammars that are consistent with the sample, it seems not to be suitable for our purposes. In our setting only one solution is needed, since even a single solution can be very complicated. Besides, the number of solutions can be huge. The version space strategy of [VB87] behaves especially poor in practice if there are no negative examples. The example productions of our application, however, are all positive examples. That is, the user gives no examples of illegal document structures. This is natural for the user, but it causes problems, not only for version space strategy, but also in general, since it can make the learning task undecidable:

Theorem 4.5 (Gold [Gol67]) Any class of languages containing all the finite languages and at least one infinite language cannot be identified in the limit from positive samples. □

Thus the class of context-free languages and even the class of regular languages cannot be learned from positive samples. In fact, this is quite natural: a consistent generalization of a set of positive examples would be a context-free grammar generating all the strings that can be constructed using the alphabet. Hence, the only possibility is to start from the most specific case and somehow generalize the representation. Still, we have the problem of overgeneralization: when to stop the generalizing process. To learn from positive examples, one needs some restrictions on the allowed result of the generalization in the form of background knowledge.

There are various ways to give background knowledge. The user can give derivation trees for grammars, the maximum number of states for automata, or restrict the subclass of the target class. The algorithm can also ask queries. In the following we look at some methods that utilize background knowledge.

Sakakibara [Sak88] presents a polynomial time algorithm that uses structural data and queries for learning context free grammars. The user gives structural descriptions of the sample in the form of derivation trees without labels. Additionally, the user has to answer *structural membership queries* and *equivalence queries*. A structural membership query proposes an unlabelled tree and asks whether it is an allowable derivation tree for the target grammar. The answer is either *yes* or *no*. A structural equivalence query proposes a grammar and asks whether its set of allowable derivation trees is the same as that of the target grammar. The answer is either *yes* or *no*. If it is *no*, then the user has to provide a counterexample. The algorithm is a modification of Angluin's algorithm for regular languages [Ang87], and it outputs a grammar structurally equivalent to the target grammar.

Sakakibara [Sak92] describes another approach to the same problem. Again, the user has to give the structural description of the sample but, instead of queries, the form of the context-free grammars is restricted by defining the condition of *reversibility*:

Definition 4.6 Let $G = (N, \Sigma, P, I)$ be a context-free grammar. G is said to be reversible if

1. if $A \to \alpha$ and $B \to \alpha$ in P, then $A = B$, and
2. if $A \to \alpha B\beta$ and $A \to \alpha C\beta$ in P, then $B = C$.

□

Reversibility is a normal form, i.e., any context-free language can be generated by some reversible context-free grammar. The inference method of [Sak92] uses the equivalence of context-free grammars and tree automata. It constructs first a simple tree automaton and merges states until the language that the automaton accepts is reversible. A number of inference methods, especially for regular languages, use this schema first introduced in [FGHR69]: Construct first a grammar that generates exactly the examples, and generalize the grammar by merging nonterminals. The same idea applied to automata merges states.

The inference method of Knuutila [Knu93] restricts the form of context-free grammars by inferring so-called *k-testable tree languages*. Any k-testable tree language can be defined by a finite set of patterns that can appear in the trees of that language. The size of the patterns is bound by constant k. The inference is made by considering the patterns of the sample and constructing a tree automaton that accepts the smallest k-testable tree language defined by those patterns. This approach is a generalization of the method described in [GV90] for k-testable string languages.

The methods of [Sak92, Knu93] are inappropriate for our applications. First, our examples contain labelled derivation trees, i.e., we have more background information. Additionally, we want to generate extended context-free grammars, that is, grammars in which the right-hand sides of productions are regular expressions. For these reasons, our method reduces the inference of context-free grammars to the inference of regular expressions that can appear on the right-hand sides of productions.

4.3 Inference of regular languages

The undecidability problem with positive examples (Theorem 4.5) concerns the class of regular languages as well as the context-free languages. However, there are a plenty of inference methods presented in the literature.

Some inference methods for regular languages are heuristic in the sense that their result does not belong to any particular class of languages. Some methods, however, are characteristic: they guarantee that the result belongs to some specified subclass of regular languages. In the following we first look at some heuristic inference methods and then present some subclasses of regular languages that can be used as target classes.

Biermann and Feldman [BF72] present a heuristic that constructs first a so-called *prefix-tree automaton* that accepts exactly the given examples. Then it merges states having identical *k-tail sets*, that is, states that have similar paths to accepting states with a look-ahead of k. The algorithm of Richetin and Vernadat [RV84] take into account one successor only: an input symbol a is b's successor if there is at least one example in which a occurs immediately after b. Kudo and Shimbo [KS88] generalize this idea further and let the user construct various merging conditions using both predecessors and successors. For instance, a condition can define that two states are merged if m of their predecessors and n of their successors are identical, and another can define that the states are merged if there is a common substring of length k in concatenated strings of predecessors and successors. With this approach also the merging conditions of [BF72] and [RV84] can be described.

Levine [Lev82] proposes a heuristic that locates different substructures that seem to occur at the same position in the examples and infers that these substructures could be interchangeable. Compared to the heuristic of Biermann and Feldman [BF72] Levine constructs tail sets that are equivalent to k-tail sets with k set to infinity. Additionally, inference process has a parameter the value of which determines the level of similarity that is needed for merging two states.

The method of Pao and Carr [PC78] constructs first a prefix-tree automaton, considers all the partitions of the states, and then tries to infer which of these partitions is the right one. Inference process compares every pair of partitions to check if they correspond to automata that accept the same language. If they do, one of the partitions is removed. If they do not, the process chooses one string from the difference of the partitions and asks the user if it belongs to the target language. If it belongs, one of the partitions can be removed since it corresponds to an automaton that does not accept that string.

Although the above approaches do not mention any language (sub)classes — and are in that sense heuristic — the successor method of [RV84] results in a language that is a so-called *local language* [MP71, GVC87], and the predecessor and successor method of [KS88] can result, for instance, in a *k-testable language* [GV90]. Actually, a local language is a k-testable language with k having a value 2. A k-testable language is defined by a finite set of substrings of length k that are permitted to appear in the strings of the language. Garcia and Vidal [GV90] present also their own algorithm for inferring k-testable languages.

In Section 5.1 we present three subclasses of regular languages: *k-reversible languages* [Ang82], *k-contextual languages* [Mug90] and our generalization of them, *(k,h)-contextual languages*. The corresponding algorithms use the general schema: a prefix-tree automaton is generalized by merging states until the automaton is k-reversible (k-contextual, (k, h)-contextual, respectively). Actually, the class of k-contextual languages is equivalent to the class of k-testable languages. We use the name k-contextual since we utilize also the definitions and algorithm of Muggleton [Mug90].

Now we can specify our grammatical inference problem, or actually the subproblem of generalizing the right-hand sides of productions. The class of languages is the class of (k, h)-contextual languages over some alphabet. The hypothesis space is the set of (k, h)-contextual automata. For each (k, h)-contextual language there is a set of positive examples, i.e., strings of the language, and an admissible sequence of examples is any sequence that contains every string over the alphabet at least once. The class of inference methods is presented in Section 5.1. The criterion for a successful inference is the (k, h)-contextuality of the language.

In practice our setting will slightly differ from the theoretical one above. The examples are the right-hand sides of simple productions that describe the structures of sample documents. Hence, the alphabet is not known beforehand. The user has not necessarily an underlying target language in mind: the target can change during the process, and the user can make errors. There are always a finite number of examples. Hence, the result is more like a minimization of the examples: set of simple productions is minimized to produce fewer productions. However, some generalization is allowed, for instance iterations generalize the examples. In addition to (k, h)-contextuality, an important criterion of success is the user's satisfaction that is affected by the clarity and compactness of the resulting grammar. For these practical reasons our automatic inference method is augmented by some *refining operations* that allow the user to influence the learning process (See Section 5.3).

Chapter 5

Generation of content models

In this chapter we present our method for generating content models from document instances. It is assumed that there is a parser available which is able to form simple productions from the instances, i.e., to form for each structure element a set of productions, each right-hand side of which describes the structure of one instance of the element.

In Section 5.1 we consider the basic generalization of the right-hand sides. As mentioned in Section 3.1.1, the SGML standard requires the content models to be unambiguous. Section 5.2 describes how the automata resulting from the previous step can be disambiguated and how the corresponding unambiguous content models can be constructed. The resulting content models after these steps, however, may not satisfy the requirements of the application, e.g., readibility. Hence, Section 5.3 presents several *refining operations* that can be applied to improve the quality of the result.

5.1 Generalizing the right-hand sides of productions

In this section we describe how the right-hand sides of sample productions can be generalized. In Section 5.1.1 we describe how the examples are represented by prefix-tree automata, and how these automata may be generalized by merging states. We introduce the families of *k-reversible, k-contextual*, and *(k,h)-contextual* languages. These families are subclasses of the family of regular languages. These language classes may serve as target classes for the generalization, i.e., states are merged until the language of an automaton belongs to a selected class. A k-contextual, and a (k, h)-contextual, language can be defined by a finite set of substrings of length $k + 1$ that are permitted to appear in the strings of the language. This set can be used as an alternative representation of the language, instead of an automaton. Section 5.1.2 defines the concept of *a set of k-grams*, and shows how the generalization can be implemented using this representation. Some complexity results are also given.

5.1.1 Generalizing automata

The right-hand sides of productions obtained from the examples are represented by an automaton called the prefix-tree automaton. To construct the prefix-tree automaton we first take a set of sample productions which have the same left-hand side. The right-hand sides of these productions are added to the prefix-tree automaton one by one. The arcs are labeled according to elements. Adding a new example starts from the first state. The existing path is followed as long as possible, i.e., until there is no arc from the current state that is labeled with the next element of the sample string, or until the sample string is totally processed. A new branch is created, if there is no other way to continue. The state corresponding to the end of an example is designated as a final state.[1]

For example, if the following productions are added into a prefix-tree automaton, the result is the automaton shown in Figure 5.1.

Entry → Headword Inflection Example Example
Entry → Headword Inflection Parallel_form Example Example Example
Entry → Headword Parallel_form Example Example
Entry → Headword Preferred_form Example
Entry → Headword Inflection Preferred_form Example Example

A prefix tree automaton accepts exactly the right-hand sides of the examples. To obtain useful grammars, we need some way of generalizing the examples, and the automaton describing them, in a meaningful way. By merging some of the states we get an automaton which accepts more strings, i.e., this automaton generalizes the examples. To merge states s_i and s_j we first choose one of them to represent the new state, say s_i. All the incoming arcs of s_j are then redirected to the set of incoming arcs of s_i, and all the outgoing arcs of s_j are redirected to the set of outgoing arcs of s_i. If one of the states is a final state, the new state is final. The generic algorithm is the following.

[1] Note that if one example is a substring of another, this state is an internal node of the tree.

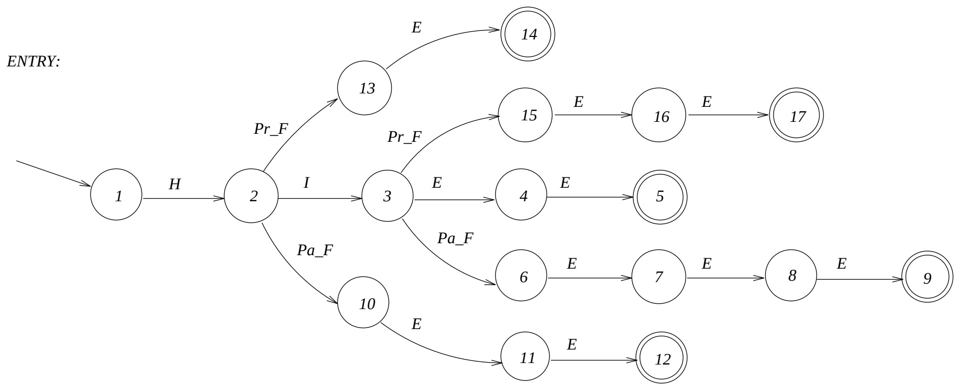

Figure 5.1: Prefix-tree automaton containing all the examples.

Algorithm 5.1 Generalizing a set of productions using some criterion for merging states.

Input: The generalization condition and a sample

$$I = \{A \rightarrow \alpha \mid A \in N,\ \alpha \in (N \cup T)^*\}$$

consisting of productions for some elements.
Output: A set

$$O = \{A \rightarrow \alpha' \mid A \in N,\ \alpha' \text{ is a regular expression over the alphabet } (N \cup T)\}$$

of generalized productions such that for all $A \rightarrow \alpha \in I$ there is a production $A \rightarrow \alpha' \in O$ such that α is an instance of α'.
Method:

```
for each element A do
    construct a prefix-tree automaton M_A from
    the productions of I with left-hand side A;
    repeat
        for each pair p, q of states of M_A do
            if p and q fulfill the generalization condition
            then modify M_A by merging p and q;
    until no more states can be merged;
    convert M_A to an equivalent regular expression E_A;
    output the production A → E_A.
```

We can formalize the process of merging states by considering the *partitions* of the states of the prefix-tree automaton [Ang82].

Definition 5.2 A *partition* of a set S is a set of pairwise disjoint nonempty subsets of S such that the union of the subsets is S. If π is a partition of S, then for any element s in S there is a unique element of π containing s, which we denote by $B(s, \pi)$, and call the *block* of π containing s. A partition π is said to *refine* another partition π' if and only if every block of π' is a union of blocks of π. A partition is the *finest* partition in the set of partitions, if it refines all the other partitions. If π is a partition of a set S and S' is a subset of S, then the *restriction* of π to S' is the partition π' consisting of all those sets B' that are nonempty and are the intersection of S' and some block of π.

Let Σ be the alphabet of set S. A *right congruence* is a partition π of Σ^* with the property that $B(w_1, \pi) = B(w_2, \pi)$ implies $B(w_1u, \pi) = B(w_2u, \pi)$ for all $w_1, w_2, u \in \Sigma^*$. If L is any language, then $T_L(w_1) = T_L(w_2)$ implies $T_L(w_1u) = T_L(w_2u)$ for all u, so L determines an associated right congruence π_L by $B(w_1, \pi_L) = B(w_2, \pi_L)$ if and only if $T_L(w_1) = T_L(w_2)$. If L is regular, the set of blocks B is finite (Myhill-Nerode theorem [HU79]). Therefore we can define the *canonical automaton* for L, the states of which are the blocks defined by π_L. We define the canonical automaton for L, $M(L) = (Q, \Sigma, \delta, I, F)$, as follows:

$$\begin{aligned}
Q &= \{T_L(u) : u \in Pr(L)\}, \\
I &= T_L(\epsilon), \text{ if } L \neq \emptyset, \text{ otherwise } I = \emptyset, \\
F &= \{T_L(w) : w \in L\}, \\
\delta(T_L(u), a) &= T_L(ua), \text{ if } u\ , \ ua \in Pr(L).
\end{aligned}$$

The automaton $M(L)$ accepts the language L and has the minimum possible number of states among all automata of L [Moo56]. Additionally, $M(L)$ has useful states only. An automaton M is called *canonical* if and only if M is isomorphic to the canonical automaton for the language of M.

Let $M = (Q, \Sigma, \delta, I, F)$ be any automaton. If π is any partition of Q, we define another automaton $M/\pi = (Q', \Sigma', \delta', I', F')$ as follows: Q' is the set of blocks of π, I' is the block of π containing the element of I, and F' is the set of all blocks of π that contain an element of F. The block B_2 is in $\delta'(B_1, a)$ whenever there exist $q_1 \in B_1$ and $q_2 \in B_2$ such that $q_2 \in \delta(q_1, a)$. The automaton M/π is called the *quotient* of M and π. □

k-reversible and k-contextual languages

How do we choose the generalization condition in Algorithm 5.1? Our assumption is that the grammars used in structured documents have only limited context in the following sense. If a sufficiently long sequence of elements occurs in two places in the examples, the elements that can follow this sequence are independent of the position of the sequence in the document structure. The classes of *k-reversible* languages [Ang82], *k-contextual* languages [Mug90], and *(k,h)-contextual* languages satisfy this condition in varying degrees. As the work presented is considerably based on the work of Angluin, we define first the property of k-reversibility, although in our experiments it has not proved to be suitable for the inference of document structures.

Definition 5.3 A regular language L is *k-reversible* if and only if for all strings u_1, u_2, w and v, if u_1vw and u_2vw are in L and $|v| = k$, then $T_L(u_1v) = T_L(u_2v)$. □

We now move on to consider the properties of k-contextuality and (k,h)-contextuality. The condition of k-contextuality is not an ideal generalization condition for document structures, either, but understanding it clarifies the concept of (k,h)-contextuality. The condition of k-contextuality is defined formally as follows.

Definition 5.4 A regular language L is *k-contextual* if and only if for all strings u_1, u_2, w_1, w_2 and v, if u_1vw_1 and u_2vw_2 are in L and $|v| = k$, then $T_L(u_1v) = T_L(u_2v)$. □

The condition of k-contextuality can be described simply in terms of automata.[2]

Lemma 5.5 A regular language L is k-contextual if and only if there is a finite automaton M such that $L = L(M)$, and for any two states p_0 and q_0 of M and any string v with $|v| = k$ we have: if there are states p_k and q_k of M such that $\delta(p_0, v) = p_k$ and $\delta(q_0, v) = q_k$, then $p_k = q_k$.

Proof Let L be a k-contextual language. Consider the canonical automaton $M(L)$. Assume there are states p_0 and q_0 in $M(L)$ such that $\delta(p_0, v) = p_k$ and $\delta(q_0, v) = q_k$. Then there are strings $u_1, u_2, w_1, w_2 \in \Sigma^*$ such that $u_1vw_1 \in L$ and $u_2vw_2 \in L$, where $\delta(I, u_1v) = p_k$ and $\delta(I, u_2v) = q_k$. Since L is k-contextual, $T_L(u_1v) = T_L(u_2v)$ and since $M(L)$ is canonical $p_k = q_k$.

Assume then M is a finite automaton with the given property, and assume $u_1vw_1 \in L(M)$ and $u_2vw_2 \in L(M)$, with $|v| = k$. Then $\delta(I, u_1v) = \delta(I, u_2v)$, and clearly $T_L(u_1v) = T_L(u_2v)$. Hence, $L(M)$ is k-contextual. □

[2] In the following we consider automata consisting of useful states only. The automata obtained from prefix-tree automata by merging states always fullfill this condition.

Definition 5.6 For a set of strings H, a k-contextual language L such that

1. $H \subseteq L$, and
2. for all k-contextual languages M, if $H \subseteq M$ then $L \subseteq M$;

is called a *minimal k-contextual language containing H*. If there is only one minimal k-contextual language containing L, it is called the *smallest* k-contextual language. □

If an automaton M fullfills the condition of Lemma 5.5, we say that M is a *k-contextual automaton*. Lemma 5.5 and Algorithm 5.1 give a method for constructing a k-contextual automaton which accepts, given as input an automaton C, a k-contextual language containing the language $L(C)$. The states of C satisfying the conditions in the implication of the lemma are merged until no such states remain. Theorem 5.8 below shows that the accepted language is the smallest k-contextual language containing $L(C)$. The algorithm is the following.

Algorithm 5.7 Constructing an automaton that accepts the smallest k-contextual regular language containing a given regular language.

Input: A regular language represented by a finite-state automaton $M = (Q, \Sigma, \delta, I, F)$; a positive integer k.
Output: A k-contextual automaton M' such that $L(M) \subseteq L(M')$ and $L(M')$ is the smallest k-contextual regular language including $L(M)$.
Method:
1. **repeat**
2. **if** for some states $p_0, q_0, p_k, q_k \in Q$
and input symbols $a, a_1, a_2, \ldots, a_k \in \Sigma$
$\delta(p_0, a_1a_2 \ldots a_k) = p_k$ **and** $\delta(q_0, a_1a_2 \ldots a_k) = q_k$
or there exists a state r and an input symbol a such that
$\delta(r, a) = \{p_k, q_k\}$
3. **then** merge p_k and q_k;
4. **until** no states can be merged.

The check for nondeterminism is not necessary if the automaton given as input is deterministic, as for instance a prefix tree automaton always is. Since the automaton is deterministic originally, the only situation that can introduce nondeterminism is the one in which there are similar paths longer than k. Then there are some states p_0, q_0, p_k, q_k, and input symbols $a_1, \ldots a_{k+1}$ such that $\delta(p_0, a_1a_2 \cdots a_k) = p_k$, $\delta(q_0, a_1a_2 \cdots a_k) = q_k$, $\delta(p_k, a_{k+1}) = p_{k+1}$, $\delta(q_k, a_{k+1}) = q_{k+1}$, and $p_{k+1} \neq q_{k+1}$. If the algorithm merges first states p_k and q_k, nondeterminism appears, i.e., $\delta(p_k, a_{k+1}) =$

$\{p_{k+1}, q_{k+1}\}$. Hence, $\delta(p_1, a_2 \cdots a_k a_{k+1}) = p_{k+1}$ and $\delta(q_1, a_2 \cdots a_k a_{k+1}) = q_{k+1}$ and the algorithm merges p_{k+1} and q_{k+1}.

In our example of Figure 5.1, let $k = 2$. We can choose first the paths $(2, 13, 14)$ and $(3, 15, 16)$ and merge states 14 and 16; the result is shown in Figure 5.2. In Figure 5.3 states 5, 8, 9, and 17 have been merged together; all of these were reachable by a path of length 2 with the labels *(E,E)*.

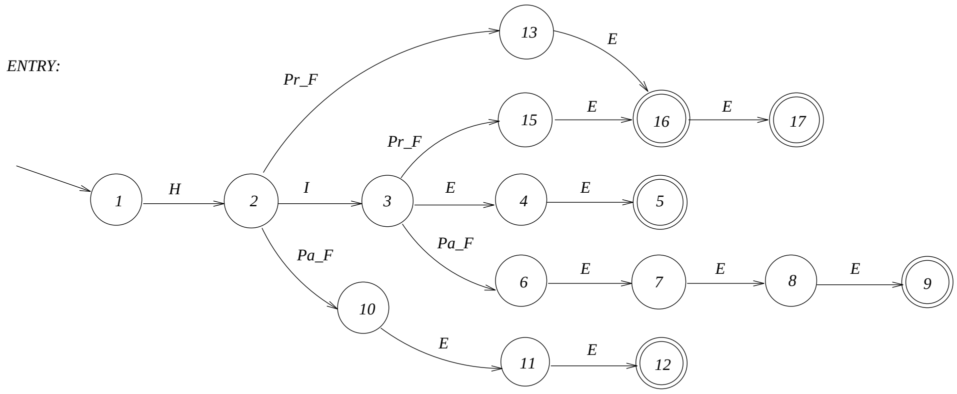

Figure 5.2: After merging states 14 and 16 in Figure 5.1.

In Figure 5.4 we can see how the algorithm can introduce nondeterminism in an automaton. We find the paths $(2, 10, 11)$ and $(3, 6, 7)$ and merge the states 7 and 11. However, there is an arc labeled E leaving both of the states. The nondeterminism is removed by merging the states 5 and 12.

Finally, the 2-contextual automaton looks like the one in Figure 5.5. We can see that it generalizes the examples quite well. There are, however, a few generalizations one would prefer to do. The automaton accepts the structures *Headword Inflection Preferred_Form Example* and *Headword Inflection Example Example*, but not the structure *Headword Inflection Example*, i.e., the existence of *Preferred_Form* changes the number of examples allowed. To accept the latter structure one should merge states 4 and 5. Similarly, also states 5, 7, and 16 could be merged. Actually, the generalization condition presented in the next section, (k, h)-*contextuality*, would make these generalizations.

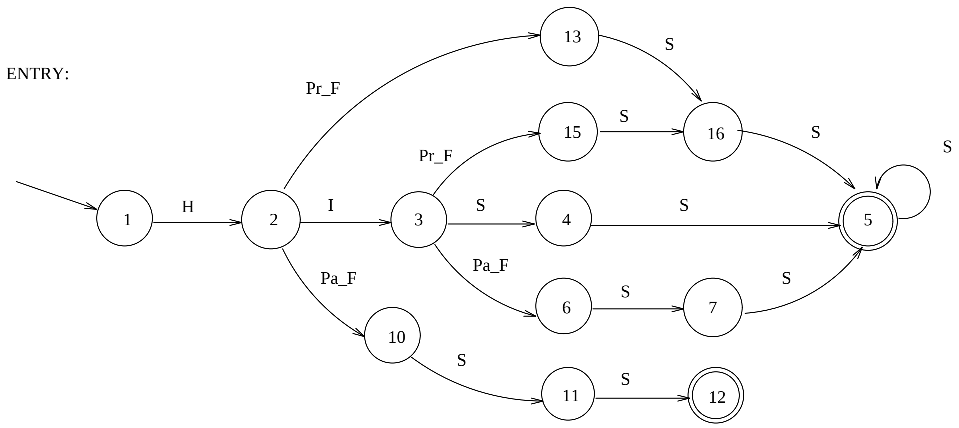

Figure 5.3: After merging states 5, 8, 9, and 17 in Figure 5.2.

Theorem 5.8 Algorithm 5.7 works correctly, i.e., for an input automaton M it produces an automaton M' accepting the smallest k-contextual language that includes $L(M)$.
Proof Can be found in [Mug90]. The corresponding proof for (k,h)-contextuality (Theorem 5.17 below) follows the idea of the proof of Muggleton. □

(k,h)-contextual languages

The intuition in using k-contextuality is that two occurrences of a sequence of length k imply that the subsequent elements are the same in both cases. We relax this condition and generalize the k-contextual languages further to (k,h)-contextual languages. In these languages two occurrences of a sequence of length k implies that the subsequent elements are the same *already after h characters.*

Definition 5.9 Let $0 \leq h \leq k$. A regular language L is (k,h)-*contextual* if and only if for all strings u_1, u_2, w_1, and w_2, and input symbols $a_1, \ldots, a_k$, if $u_1 a_1 \ldots a_k w_1$ and $u_2 a_1 \ldots a_k w_2$ are in L, then for every i, with $0 \leq h \leq i \leq k$ and each pair of strings v_1 and v_2 such that $v_1 a_i \ldots a_k w_1 \in L$ and $v_2 a_i \ldots a_k w_2 \in L$, we have $T_L(v_1 a_i) = T_L(v_2 a_i)$.

□

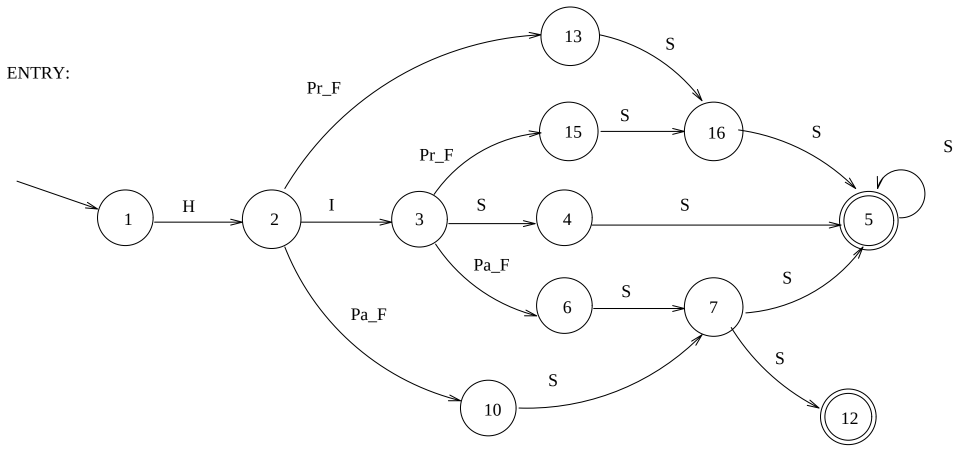

Figure 5.4: Nondeterminism occurs when 7 and 11 are merged in Figure 5.3.

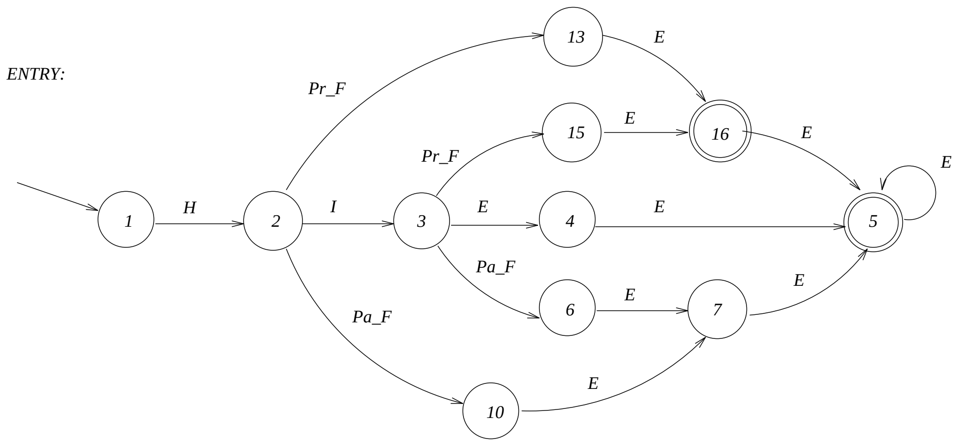

Figure 5.5: 2-contextual automaton.

As for k-contextuality, we obtain an easy characterization in terms of automata.

Lemma 5.10 A regular language L is (k,h)-contextual if and only if there is a finite automaton M such that $L = L(M)$, and for any two states p_0 and q_0 of M, and all input symbols $a_1a_2\ldots a_k$ we have: if there are states $p_1,\ldots,p_k$ and $q_1,\ldots,q_k$ such that $\delta(p_0,a_1) = p_1, \delta(p_1,a_2) = p_2,\ldots,\delta(p_{k-1},a_k) = p_k$ and $\delta(q_0,a_1) = q_1, \delta(q_1,a_2) = q_2,\ldots,\delta(q_{k-1},a_k) = q_k$, then $p_i = q_i$, for every i with $0 < h \leq i \leq k$.

Proof Let L be a (k,h)-contextual language, and let $M(L)$ be the canonical automaton of L. Assume there are states p_0 and q_0 in $M(L)$ such that $\delta(p_0,a_1) = p_1, \delta(p_1,a_2) = p_2,\ldots,\delta(p_{k-1},a_k) = p_k$ and $\delta(q_0,a_1) = q_1, \delta(q_1,a_2) = q_2,\ldots,\delta(q_{k-1},a_k) = q_k$. Then $M(L)$ accepts strings $u_1a_1\cdots a_kw_1$ and $u_2a_1\cdots a_kw_2$, such that $\delta(I,u_1a_1\cdots a_i) = p_i$ and $\delta(I,u_2a_1\cdots a_i) = q_i$. Since L is (k,h)-contextual, for each pair of prefixes $u_1a_1\cdots a_i$ and $u_2a_1\cdots a_i$, we have $T_L(u_1a_1\cdots a_i) = T_L(u_2a_1\cdots a_i)$, and hence $p_i = q_i$ for each i with $h \leq i \leq k$.

Assume then M is a finite automaton with the given property, and assume $u_1a_1\cdots a_kw_1$ and $u_2a_1\cdots a_kw_2 \in L(M)$. Since $p_i = q_i$ for each $i \geq h$, clearly all the strings that can precede $a_i\cdots a_kw_1$ and $a_i\cdots a_kw_2$ may be continued with the same suffixes. □

Again, an automaton M is said to be (k,h)-contextual, if it fullfills the conditions of Lemma 5.10.

Remark 5.11 If $M = (Q,\Sigma,I,F,\delta)$ is (k,h)-contextual and $u_1a_1\ldots a_h\ldots a_kw_1$ and $u_2a_1\ldots a_h\ldots a_kw_2 \in L(M)$, then for each i with $h \leq i \leq k$ there is a unique state q_i such that $\delta(I,u_1a_1\ldots a_i) = \delta(I,u_2a_1\ldots a_i) = q_i$. □

The algorithm for producing the automaton that accepts a (k,h)-contextual language is similar to the previous algorithm: one looks for states satisfying the conditions of the above lemma, and merges states. If similar paths of length k are found, not only the last states but also some other states along the paths are merged. If $h = k$ only the last states are merged. If $h < k$ the paths have a similar prefix of length h before they are joined, i.e., $k - h + 1$ states are merged.

Algorithm 5.12 constructs an automaton that accepts the smallest (k,h)-contextual language containing the language accepted by the input automaton. Continuing our example, in Figure 5.6 we can see the final (2,1)-contextual automaton.

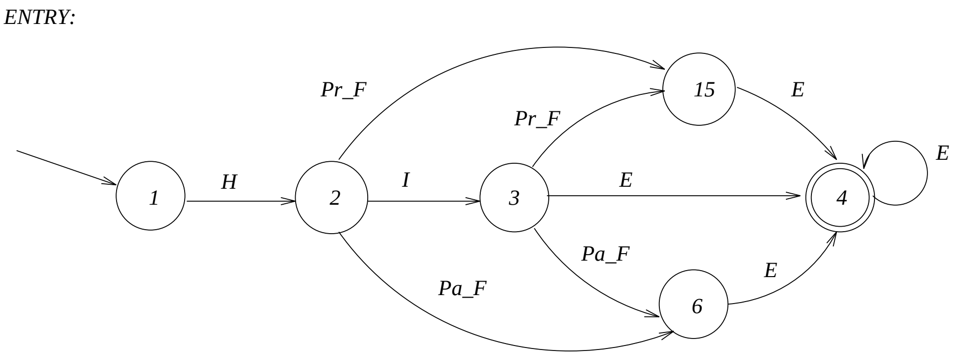

Figure 5.6: (2,1)-contextual automaton.

Algorithm 5.12 Constructing an automaton accepting the smallest (k, h)-contextual regular language containing a given regular language.

Input: A regular language represented by a finite-state automaton M, and positive integers k and h, with $h \leq k$.
Output: An automaton M' such that $L(M) \subseteq L(M')$ and $L(M')$ is the smallest (k, h)-contextual regular language including $L(M)$.
Method:

repeat
  if for some states $p_0, \ldots, p_k$ and $q_0, \ldots, q_k$, and
    input symbols $a_1, a_2, \ldots, a_k$,
    $\delta(q_0, a_1) = q_1,\ \delta(q_1, a_2) = q_2,\ \ldots,\ \delta(q_{k-1}, a_k) = q_k$ **and**
    $\delta(p_0, a_1) = p_1,\ \delta(p_1, a_2) = p_2,\ \ldots,\ \delta(p_{k-1}, a_k) = p_k$
  then
    for $i \leftarrow h$ **to** k **do**
      merge p_i and q_i;
until no states can be merged.

As for k-contextual languages, we have to show that Algorithm 5.12 works correctly, i.e., that it constructs an automaton accepting the smallest (k, h)-contextual language containing the given set of strings. The proof follows the corresponding proofs in [Ang82, Mug90]. Before stating the actual theorem, we have to present the following lemmas. First we show that a subautomaton of a (k, h)-contextual automaton is also (k, h)-contextual.

Lemma 5.13 If M is a (k, h)-contextual automaton and M' is any subautomaton of M, then M' is a (k, h)-contextual automaton.
Proof Let k and h be positive integers, $M = (Q, \Sigma, \delta, I, F)$ a (k, h)-contextual automaton, and $M' = (Q', \Sigma', \delta', I', F')$ a subautomaton of M,

i.e., $Q' \subseteq Q, \Sigma' \subseteq \Sigma, I' \subseteq I, F' \subseteq F$, and $\delta'(q,a) \subseteq \delta(q,a)$. Assume that M' is not (k,h)-contextual. Thus for some i with $h \leq i \leq k$ there exists strings $u_1a_1 \ldots a_h \ldots a_i \ldots a_k w_1 \in L(M')$ and $u_2a_1 \ldots a_h \ldots a_i \ldots a_k w_2 \in L(M')$, such that $\delta'(I, u_1a_1 \ldots a_h \ldots a_i) \neq \delta'(I, u_2a_1 \ldots a_h \ldots a_i)$.

Remark 5.11 shows that $\delta(I, u_1a_1 \ldots a_i) = \delta(I, u_2a_1 \ldots a_i) = \{q_i\}$ for every i, where $h \leq i \leq k$. By the definition of subautomata $\delta'(q', b) \subseteq \delta(q', b)$ for all $q' \in Q'$ and $b \in \Sigma'$. It can be shown by induction that also $\delta'(q', w) \subseteq \delta(q', w)$ for all $w \in \Sigma'^*$. Thus it follows that both $\delta'(I, u_1a_1 \ldots a_i) \subseteq \delta(I, u_1a_1 \ldots a_i) = \{q_i\}$ and $\delta'(I, u_2a_1 \ldots a_i) \subseteq \delta(I, u_2a_1 \ldots a_i) = \{q_i\}$ for every i, $h \leq i \leq k$. However, since $u_1a_1 \ldots a_h \ldots a_i \ldots a_k w_1$ and $u_2a_1 \ldots a_h \ldots a_i \ldots a_k w_2 \in L(M')$, $\delta'(I, u_1a_1 \ldots a_i) \neq \emptyset$ and $\delta'(I, u_2a_1 \ldots a_i) \neq \emptyset$. Thus, for every i, $\delta'(I, u_1a_1 \ldots a_i) = \delta'(I, u_2a_1 \ldots a_i) = \{q_i\}$. This contradicts our assumption that M' is not (k,h)-contextual. Therefore M' is (k,h)-contextual. □

The following lemma shows that merging states in an automaton generalizes the language.

Lemma 5.14 Let S be a nonempty positive sample, and let M_0 be the prefix-tree automaton for S, and $M_1, \ldots, M_i, \ldots, M_f$ be the automata obtained by merging states of M_0 by Algorithm 5.12, Then $S = L(M_0) \subseteq L(M_1) \subseteq \ldots \subseteq L(M_i) \subseteq \ldots \subseteq L(M_f)$.

Proof Let M_i and M_{i+1} be automata obtained from M_0 by merging states, and assume especially that M_{i+1} is obtained from M_i by merging states p and q. All the transitions to and from both of the states in M_i remain in M_{i+1}. Hence $L(M_i) \subseteq L(M_{i+1})$. □

Next we show that the automaton output by Algorithm 5.12 is the largest (k,h)-contextual automaton obtained by merging states of the given prefix-tree automaton.

Lemma 5.15 Let S be a nonempty positive sample, k and h positive integers with $h \leq k$, $M_0 = (Q_0, \Sigma_0, \delta_0, I_0, F_0)$ the prefix-tree automaton for S, and let $M_f = M_0/\pi_f$ be the automaton output by Algorithm 5.12 on input S, k, and h. Then π_f is the finest partition of the states of M_0 such that quotient M_0/π_f is (k,h)-contextual.

Proof Algorithm 5.12 continues to merge states until there is no pair of states that violates the (k,h)-contextuality condition. Thus the automaton M_0/π_f is (k,h)-contextual.

It remains to be shown that if π is any partition of Q_0 such that M_0/π is (k,h)-contextual, then π_f refines π. Let us assume the opposite, i.e., there exists some π different from π_f such that π_f does not refine π and M_0/π is (k,h)-contextual. Thus at least one block of π_f contains elements from more than one block of π. This means that there are states p and q of Q_0 that are merged in M_0/π_f but not in M_0/π.

If two states p and q are merged in M_0/π_f, there exist some $p_0,\ldots,p_k$, $q_0,\ldots,q_k$, and input symbols $a_1,a_2,\ldots,a_k$ such that $\delta(q_0,a_1) = q_1$, $\delta(q_1,a_2) = q_2$, $\ldots$, $\delta(q_{k-1},a_k) = q_k$ and $\delta(p_0,a_1) = p_1$, $\delta(p_1,a_2) = p_2$, $\ldots$, $\delta(p_{k-1},a_k) = p_k$, and $p = p_i$ and $q = q_i$ for some i, where $h \leq i \leq k$. If states p and q are not merged in M_0/π, they do not fullfill the (k,h)-contextual condition. Hence M_0/π is not (k,h)-contextual, which contradicts the original assumption and shows that π_f refines all partitions π for which M_0/π is (k,h)-contextual. □

Finally, the next lemma states that any automaton we may form by merging states of some prefix-tree automaton is isomorphic to a subautomaton of the canonical automaton of some language.

Lemma 5.16 Let S be a positive sample of the regular language L, and let M_0 be the prefix-tree automaton for S. Let π be the partition π_L restricted to the set $Pr(S)$ of prefixes of elements of S. Then M_0/π is isomorphic to a subautomaton of the canonical automaton $M(L)$. Thus, $L(M_0/\pi)$ is contained in L.
Proof Angluin [Ang82]. □

Now we are ready to show that Algorithm 5.12 works correctly.

Theorem 5.17 Let S be a nonempty positive sample, k and h positive integers, and let M_f be the automaton output by Algorithm 5.12 on input S, k, and h. Then $L(M_f)$ is the smallest (k,h)-contextual language containing S.
Proof Lemma 5.15 shows that $L(M_f)$ is a (k,h)-contextual language, and Lemma 5.14 shows that it contains S. Let L be any (k,h)-contextual language containing S, and let π be the restriction of the partition π_L to the elements of $Pr(S)$. If M_0 denotes the prefix tree automaton for S, then Lemma 5.16 shows that M_0/π is isomorphic to a subautomaton of $M(L)$, and thus $L(M_0/\pi)$ is contained in L. From Lemma 5.13 $L(M_0/\pi)$ is (k,h)-contextual since M_0/π is isomorphic to a subautomaton of $M(L)$ that is (k,h)-contextual. Thus by Lemma 5.15 π_f refines π, so according to Lemma 5.14 $L(M_0/\pi_f)$ is contained in $L(M_0/\pi)$. Consequently, $L(M_f)$ is contained in L, and $L(M_f)$ is the smallest (k,h)-contextual language containing S.

□

Properties of (k, h)-contextual languages

We now show some properties of the class of (k, h)-contextual languages. First, we state some inclusion relationships between (k, h)-contextual languages, k-reversible languages, and k-contextual languages. Second, we consider closure properties.

Theorem 5.18 Let k and h be positive integers.

1. Every k-contextual language is k-reversible.
2. Every (k, h)-contextual language is k-reversible.
3. Every k-contextual language is (k, k)-contextual, and vice versa.
4. For every $h < k$, we have: every (k, h)-contextual language is $(k , h + 1)$-contextual.
5. Each k-contextual language is (k, h)-contextual for some h.

Proof

1. A language L is k-reversible if and only if whenever u_1vw and u_2vw are in L and $|v| = k$, then $T_L(u_1v) = T_L(u_2v)$. A language L is k-contextual if and only if whenever u_1vw_1 and u_2vw_2 are in L and $|v| = k$, $T_L(u_1v) = T_L(u_2v)$. Since the latter condition is tighter, any k-contextual language is k-reversible.
2. A language L is (k, h)-contextual if and only if whenever $u_1a_1 \dots a_h \dots a_kw_1$ and $u_2a_1 \dots a_h \dots a_kw_2$ are in L, $T_L(u_1a_1 \dots a_i) = T_L(u_2a_1 \dots a_i)$, where $h \leq i \leq k$. Specifically, if $u_1a_1 \dots a_kw$ and $u_2a_1 \dots a_kw$ are in L, then $T_L(u_1a_1 \dots a_k) = T_L(u_2a_1 \dots a_k)$. Hence, any (k, h)-contextual language is k-reversible.
3. Obvious from the definitions.
4. Obvious from the definitions.
5. Since any (k, h)-contextual language is $(k, h + 1)$-contextual we can show by induction that any (k, h)-contextual language is (k, k)-contextual. Additionally, as shown above, any k-contextual language is (k, k)-contextual, and vice versa.

$\square$

Theorem 5.19 Let k and h be positive integers, and let $L_1, L_2 \subseteq \Sigma^*$ be (k,h)-contextual languages. Then

1. $L_1 \cup L_2$ is not necessarily a (k,h)-contextual language.
2. $L_1 \cap L_2$ is a (k,h)-contextual language.

Proof

1. Assume, for instance, that $L_1 = \{abcd, ebcd\}$ and $L_2 = \{abg\}$. Then $L_1 \cup L_2$ is not $(2,1)$-contextual while both the languages are $(2,1)$-contextual.
2. $L_1 \cap L_2$ is regular because L_1 and L_2 are regular [HU79]. Let $u_1a_1 \cdots a_kv_1 \in L_1 \cap L_2$, $u_2a_1 \cdots a_kv_2 \in L_1 \cap L_2$, and $u_1a_1 \cdots a_h \cdots a_iw \in L_1 \cap L_2$, where $u_1, u_2, v_1, v_2, w \in \Sigma^*$, $a_1, \cdots, a_k \in \Sigma$, and $h \leq i \leq k$. Since $u_1a_1 \cdots a_kv_1 \in L_1$, $u_2a_1 \cdots a_kv_2 \in L_1$, $u_1a_1 \cdots a_h \cdots a_iw \in L_1$, and L_1 is (k,h)-contextual, also $u_2a_1 \cdots a_h \cdots a_iw \in L_1$. Respectively, $u_2a_1 \cdots a_h \cdots a_iw \in L_2$. Hence $u_2a_1 \cdots a_h \cdots a_iw \in L_1 \cap L_2$, and thus $L_1 \cap L_2$ is (k,h)-contextual.

□

5.1.2 Implementation of generalization

Sets of k-grams

In the previous sections we have represented the right-hand sides of productions as k- and (k,h)-contextual automata. If an automaton is k-contextual, all the paths of length k that contain the same sequence of input symbols end at the same state. This feature illustrates the point that for any k-contextual language L there exists a finite set of strings of length $k+1$ that uniquely identifies L. We prove this shortly.

As mentioned in Chapter 4, the family of k-contextual languages is also known as the family of k-testable languages. Garcia and Vidal [GV90] define a k-testable language by a regular expression of the form

$$L = (I\Sigma^* \cap \Sigma^* F) \setminus \Sigma^* T \Sigma^*,$$

where Σ is the alphabet, and $I \subseteq \cup_{i=1}^{k-1}\Sigma^i$ and $F \subseteq \cup_{i=1}^{k-1}\Sigma^i$ are sets of initial and final substrings, respectively, while $T \subset \Sigma^k$ is a set of forbidden substrings.

We define k-contextual languages not with the help of forbidden substrings but using the allowed substrings.

Definition 5.20 Let S be a set of strings over Σ, where $\# \notin \Sigma$. The set of k-grams of S is defined as

$$grams(S, k) = \{u \mid u \text{ is a substring of } \#^{k-1}s\#,\ |u| = k,\ s \in S\}.$$

□

Definition 5.21 Let G be a set of k-grams over alphabet Σ. Then G *generates* the language $L(G) = \{w \mid grams(\{w\}, k) \subset G\}$. A k-gram of the form $\#^{k-1}a$, where $a \in \Sigma$, is called an *initial* k-gram. Respectively, a k-gram of the form $u\#$, where $u \in \Sigma^{k-1}$, is called a *final* k-gram. A k-gram g is *useless* in G if there is no string $w = \#^k v\#$, where $v \in L(G)$, such that $g \in grams(\{w\}, k)$. □

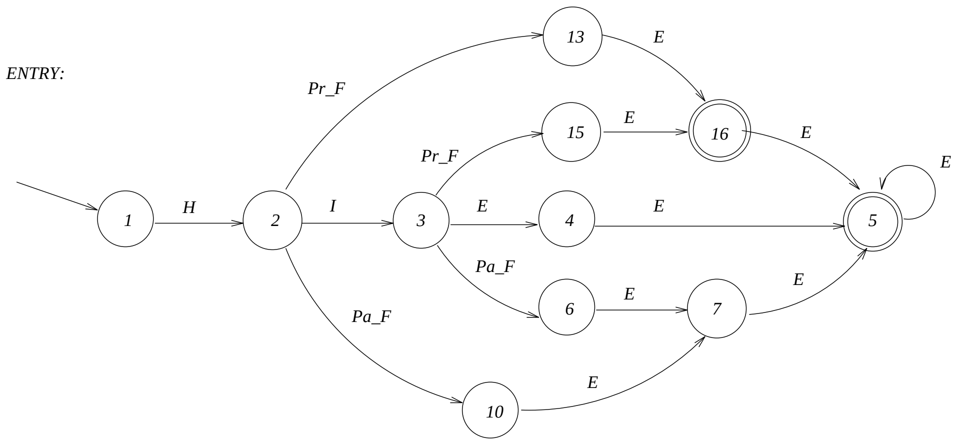

Figure 5.7: (2,1)-contextual automaton.

Example 5.22 The 3-grams of the 2-contextual language accepted by the automaton in Figure 5.7 are

# # H	# H Pr_F	# H I	# H Pa_F
H Pr_F E	I Pr_F E	H I Pr_F	H I E
H I Pa_F	H Pa_F E	I Pa_F E	Pr_F E E
Pa_F E E	I E E	E E E	Pr_F E #
Pa_F E #	I E #	E E #	

Here # # H is the initial gram, and Pr_F E #, Pa_F E #, I E #, and E E # are the final grams. If there were a gram H E E in this gram set, it would be useless. □

Since the alphabet is finite, the set of $k+1$-grams for a k-contextual language is also finite. The set of grams for a k-contextual language is unique, assumed that the set does not have any useless grams.

Theorem 5.23 If a language L is k-contextual, then there exists a set G of $k+1$-grams such that $L(G) = L$.

Proof Let L be a k-contextual language, and let $G = grams(L, k+1)$. Clearly $L \subseteq L(G)$, since for each $w \in L$, $grams(\{w\}, k+1) \subset G$. Hence, it remains to be shown that $L(G) \subseteq L$. Let us assume the opposite, i.e., there is a string w such that $w \in L(G)$ but $w \notin L$.

As $w \in L(G)$, we have $grams(\{w\}, k+1) \subseteq G = grams(L, k+1)$. So every $k+1$ -gram of w is a gram of some word in L. As $w \notin L$, there is a shortest prefix $w_3 a$ of w such that $w_3 a \notin Pr(L)$, but $w_3 \in Pr(L)$. Denote $l = |w_3 a|$. If $l < k+1$, then $\#^{k+1-l} w_3 a$ is a gram of w, and hence a gram of some word in L, which is a contradiction. Thus $l \geq k+1$. Let $w_3 = u_1 a_1 \ldots a_{k+1}$. Thus $a_1 \ldots a_{k+1}$ and $a_2 \ldots a_{k+1} a$ are grams of w, and hence grams of some words $w_1 \in L$ and $w_2 \in L$. Hence, $w_1 = u_1 a_1 \ldots a_{k+1} v_1 = w_3 v_1 \in L$ and $w_2 = u_2 a_2 \ldots a_{k+1} a v_2 \in L$. As L is k-contextual, there is a string $w_4 \in L$ with prefix $u_1 a_1 \ldots a_{k+1} a = w_3 a$, i.e., $w_3 a \in Pr(L)$, which is a contradiction.

□

Theorem 5.24 Let G be a set of k-grams. Then $L(G)$ is $k-1$-contextual.

Proof Remove all useless grams for G from set G. If G is empty, then L is empty and hence $k-1$-contextual. Otherwise, $L = L(G) = \{w \mid grams(\{w\}, k) \subset G\}$. If there are strings $u_1 a_1 \cdots a_{k-1} v_1$ and $u_2 a_1 \cdots a_{k-1} v_2$ in $L(G)$, also the strings $u_1 a_1 \cdots a_{k-1} v_2$ and $u_2 a_1 \cdots a_{k-1} v_1$ belong to $L(G)$, since both prefixes $u_1 a_1 \cdots a_{k-1}$ and $u_2 a_1 \cdots a_{k-1}$ can be continued with all the grams starting $a_1 \cdots a_{k-1}$. □

Generalizing a sample to a k-contextual language can be done simply by adding all substrings of length $k+1$ to the set of $k+1$-grams. The algorithm is the following.

Algorithm 5.25 Constructing a set of $k+1$-grams for the smallest k-contextual language containing a given sample.

Input: A finite set of strings S, and a positive integer k.
Output: A set G of $k+1$-grams for language L such that L is the smallest k-contextual regular language including S.
Method:
1. **for** each $g \in grams(S, k+1)$ **do**
2. $\quad G := G \cup \{g\}$.

The (k,h)-contextual languages require additionally that the set is the set of $k+1$-grams for some (k,h)-contextual language. A set of $k+1$-grams can be made (k,h)-contextual by adding grams as in Algorithm 5.27 below. Adding grams corresponds merging states in automata.

First we have to define the concepts of the predecessors and the successors of a k-gram.

Definition 5.26 A k-gram u_1 is said to be a *predecessor* of a k-gram $u_2 = a_1a_2\cdots a_k$ if $u_1 = ba_1\cdots a_{k-1}$ for some b or u_1 is a predecessor of $ba_1\cdots a_{k-1}$ for some b. Respectively, a k-gram u_1 is said to be a *successor* of a k-gram $u_2 = a_1a_2\cdots a_k$ if $u_1 = a_2a_3\cdots a_kc$ for some c or u_1 is a successor of $a_2a_3\cdots a_kc$ for some c. □

Algorithm 5.27 Converting a set of $k+1$-grams to represent a (k,h)-contextual language.

Input: A set H of $k+1$-grams for language L.
Output: A set G of $k+1$-grams for the smallest (k,h)-contextual language L' such that $L \subseteq L'$.
Method:
1. G := H;
2. **for** each $k+1$-gram $g = ab_1\dots b_k \in G$ **do**
3. $\quad$ **if** there is a $k+1$-gram $f = cb_1\dots b_k \in G$ for some $c \neq a$
4. $\quad$ **then**
5. $\quad\quad$ **for** each $k+1$-gram $f' = cb_1\dots b_hw_1 \in G$ **do**
6. $\quad\quad\quad$ **if** f and f' have a common predecessor $wcb_1\dots b_h \in G$
7. $\quad\quad\quad$ **then** $G := G \cup \{ab_1\dots b_hw_1\}$;
8. $\quad\quad$ **for** each $k+1$-gram $g' = ab_1\dots b_hw_2 \in G$ **do**
9. $\quad\quad\quad$ **if** g and g' have a common predecessor $w'ab_1\dots b_h \in G$
10. $\quad\quad\quad$ **then** $G := G \cup \{cb_1\dots b_hw_2\}$.

Theorem 5.28 Algorithms 5.25 and 5.27 work correctly. That is, let S be a nonempty positive sample, and let G be the set of $k+1$-grams output by the Algorithm 5.25 followed by Algorithm 5.27 on input S. Then $L(G)$ is the smallest (k,h)-contextual language containing S.

Proof Algorithm 5.25 constructs first a k-contextual set of $k+1$-grams, let us call it G'. Assume L is any k-contextual language containing S. Clearly $grams(S, k+1) \subseteq grams(L, k+1)$. Since $grams(L(G'), k+1)) = grams(S, k+1)$, also $grams(L(G'), k+1) \subseteq grams(L, k+1)$. Hence, $L(G') \subseteq L$, and thus G' corresponds the smallest k-contextual language containing S.

Next we show that the output G of Algorithm 5.27 on G represents the smallest (k,h)-contextual language containing S. Let L' be any (k,h)-contextual language such that $S \subseteq L'$ and $L' \subseteq L(G)$. Since $S \subseteq L'$, $grams(S, k+1) \subseteq grams(L', k+1)$. If a $k+1$-gram $g = ab_1 \ldots b_h w$ has to be added to G', there are $k+1$-grams $g_1 = ab_1 \ldots b_h \ldots b_{k+1}$, $g_2 = cb_1 \ldots b_h \ldots b_{k+1}$, and $g_3 = cb_1 \ldots b_h w$ in G'. Hence there are strings $u_1, u_2, u_3 \in S$ such that $u_1 = v_1 g_1 w_1, u_2 = v_2 g_2 w_2$, and $u_3 = v_2 g_3 w_3$. Therefore a string $u = v_1 g w_3$ has to belong to any (k,h)-contextual language containing S. If a $k+1$-gram g does not belong to $grams(L', k+1)$, $u \notin L'$, and L' is not (k,h)-contextual. Hence, after all additions of $k+1$-grams, $L(G') = L(G)$ is the smallest (k,h)-contextual language containing S. □

Data structure and complexity results

Figure 5.8 shows an overview of the data structure used in the implementation. The grams are stored in a table that is mainly accessed via a hashtable that contains the $k-1$ -prefixes and suffixes of the grams. Since the collissions in the hashtable are rare, we assume in the following that a prefix can be accessed in a constant time.

In Figure 5.9 can be seen the detailed structures of a prefix node and a gram. A prefix contains a list of grams that have this prefix, and it also contains a list of all the grams that have this prefix as a suffix. Hence, we can easily access both the grams that can follow a gram in the strings of the language and the grams that can precede it. A prefix contains the actual values, the gram itself contains links to its prefix and suffix only, and also the kth symbol.

The linking structure facilitates fast manipulation of sets of grams needed in various operations. As an example, we consider some common operations in the following. Assume Σ is the alphabet.

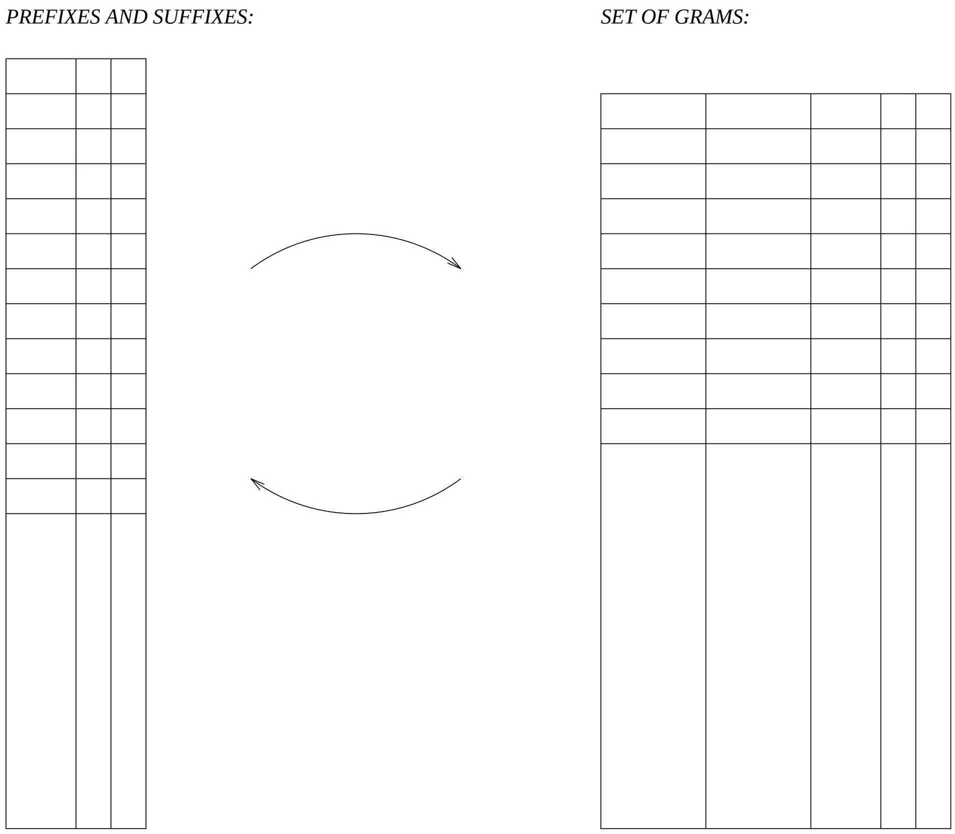

Figure 5.8: Overview of the data structure.

Add a gram. The prefix of the gram is found via the hashtable. If the prefix does not exist it is created (constant time). Then the list of grams is traversed, and if the gram does not exist, it is added to the end of the table (traversal takes less than $\mathcal{O}(|\Sigma|)$ time), and to the end of the list of the grams with the same prefix. The gram is also added to the list of the grams with the same suffix, which takes constant time, if the gram is added to the head of the list.

Access a gram. The prefix of the gram is found via the hashtable, and the list of the grams with the same prefix is followed until the gram is found. The traversal takes less than $\mathcal{O}(|\Sigma|)$ time.

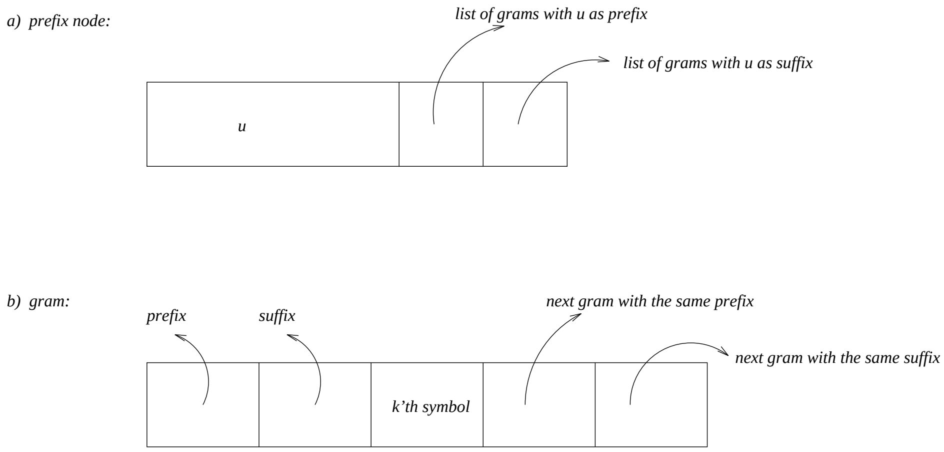

Figure 5.9: The structure of a) a prefix node, $|u| = k - 1$, b) a gram.

Next grams of a gram. The suffix of the given gram can be accessed in constant time, and the next grams can be found from the list of the grams the prefix of which is this suffix. Again, the number of these grams is less than $|\Sigma|$.

Theorem 5.29 Algorithms 5.25 and 5.27 can be implemented to run in time $\mathcal{O}(n)$, where n is the sum of the lengths of the input strings.

Proof Let $|S|$ be the number of the input strings. The set of $k+1$-grams can be constructed in time $\mathcal{O}(n)$. The number of the $k+1$-grams in the set is at most $n + |S|$. Given a set of examples, a (k,h)-contextual set of $k+1$-grams has more grams than a k-contextual set but the size is still less than $n + |S|$, since the extra $k+1$-grams exist only if there are similar substrings in examples. Hence, assumed that a $k+1$-gram can be accessed in constant time using as a key either the first k or the last k symbols, constructing a (k,h)-contextual set of $k+1$-grams can be done in time $\mathcal{O}(n)$. □

Additionally, the number of the k-grams is always bounded by $(|\Sigma|+1)^k$, i.e., it is independent of the size of the sample. In a practical situation, however, the alphabet is not given beforehand but is considered to be the set of symbols seen so far in the sample. Hence, the size of the alphabet may increase as more examples are seen. Generally, the set of k-grams also grows for a while but since the examples usually have common substrings the size gradually converges.

The respective results of Angluin and Muggleton are the following. According to Angluin [Ang82], a k-reversible automaton can be constructed in time $\mathcal{O}(kn^3)$, where n is the sum of the lengths of the input strings. Muggleton [Mug90] presents a more efficient $\mathcal{O}(n^2)$–algorithm for the same problem. His implementation of the inference of k-contextual languages needs linear time.

5.2 Disambiguation

A context-free grammar G such that some word w in $L(G)$ has two parse trees is said to be *ambiguous*. A context-free language for which every context-free grammar is ambiguous is said to be an *inherently ambiguous* context-free language.

The SGML standard requires that the content models have to be unambiguous in the following stricter sense. A content model is ambiguous if an element appearing in the document instance can be matched with more than one occurrence of the corresponding element in the content model without look-ahead, i.e., without scanning the text ahead to decide which occurrence should be chosen.

Brüggemann-Klein and Wood [BKW92, BKW94] have presented an algorithm that can decide whether a content model is unambiguous. We have developed their ideas further and present a disambiguation algorithm that transforms an ambiguous content model into an unambiguous one. The resulting content model generalizes the original, i.e., accepts more element structures. Since our method never creates content models containing &-operators, our disambiguation algorithm cannot disambiguate content models containing them.

The next section outlines the decision algorithm of Brüggemann-Klein and Wood and the basic concepts needed. In Section 5.2.2 we present our disambiguation algorithm for automata, and in Section 5.2.3 the conversion into an unambiguous content model, as presented by Brüggemann-Klein and Wood.

5.2.1 1-unambiguity

Brüggemann-Klein and Wood [BKW94] call the unambiguity required in the SGML standard 1-*unambiguity*, and give a definition for it in terms of the pairs of positions that follow each other in a word. First, they define the following sets.

Definition 5.30 Let Σ be a set of elements. For $L \subset \Sigma^*$, let

- $\text{first}(L) = \{a \in \Sigma \mid aw \text{ is in } L \text{ for some string } w\}$,
- $\text{last}(L) = \{a \in \Sigma \mid wa \text{ is in } L \text{ for some string } w\}$,
- $\text{followlast}(L) = \{a \in \Sigma \mid vaw \in L$, for some string v in $L \setminus \{\epsilon\}$ and some string $w\}$.

Furthermore, the definitions of the sets above are extended to expressions E by defining $\text{first}(E) = \text{first}(L(E))$, for each expression E, and similarly for the other sets. □

Definition 5.31 1-unambiguity of a regular expression E is defined inductively:

- $E = \emptyset$, $E = \epsilon$, or $E = a$, with $a \in \Sigma$: E is 1-unambiguous.
- $E = F \mid G$: E is 1-unambiguous if and only if F and G are 1-unambiguous, and $\text{first}(F) \cap \text{first}(G) = \emptyset$.
- $E = FG$: If $L(E) = \emptyset$, then E is 1-unambiguous. If $L(E) \neq \emptyset$ and $\epsilon \in L(F)$, then E is 1-unambiguous if and only if F and G are 1-unambiguous, $\text{first}(F) \cap \text{first}(G) = \emptyset$, and $\text{followlast}(F) \cap \text{first}(G) = \emptyset$. If $L(E) \neq \emptyset$ and $\epsilon \notin L(F)$, then E is 1-unambiguous if and only if F and G are 1-unambiguous and $\text{followlast}(F) \cap \text{first}(G) = \emptyset$.
- $E = F^*$: E is 1-unambiguous if and only if F is 1-unambiguous and $\text{followlast}(F) \cap \text{first}(F) = \emptyset$.

□

A regular language is 1 -unambiguous if it is denoted by some 1-*unambiguous expression*. Brüggemann-Klein and Wood characterize the set of 1-unambiguous regular languages in terms of structural properties of the minimal deterministic automata that recognize them and show how a 1-unambiguous expression can be constructed from a 1-unambiguous deterministic automaton. In the following, we consider first the structural properties of 1-unambiguous automata, as presented in [BKW94].

Major causes of ambiguities are iterations, i.e., cycles in automata. Therefore, we have to consider the strongly connected components, so-called *orbits*, of an automaton.

Definition 5.32 Let $M = (Q, \Sigma, \delta, I, F)$ be a finite automaton. For state $q \in Q$, the strongly connected component of q, i.e., the states of M that can be reached from q and from which q can be reached as well, is called the *orbit* of q and denoted by $O(q)$. We consider the orbit of q to be *trivial* if $O(q) = \{q\}$ and there are no transitions from q to itself in M. □

Definition 5.33 A state q in a finite automaton M is called a *gate* of its orbit if either q is a final state or if there are $q' \in Q \setminus O(q)$ and $a \in \Sigma$ with $\delta(q, a) = q'$. The finite automaton M has the *orbit property* if each orbit of M is homogenous with respect to its gates, i.e., if, for all orbits O of M and for all gates q_1 and q_2 of O, we have

- q_1 is a final state if and only if q_2 is a final state.
- $\delta(q_1, a) = q'$ if and only if $\delta(q_2, a) = q'$, for all $q' \in Q \setminus O(q_1) = Q \setminus O(q_2)$, and for all $a \in \Sigma$.

□

Example 5.34 The automaton in Figure 5.10 has four orbits, $\{1\}$, $\{2\}$, $\{3, 4, 5\}$, and $\{6\}$. The automaton does not have the orbit property, since the gates of the third orbit, states 3 and 4, are not homogenous. The state 3 is a final state, while the state 4 is not, and there is a R-transition from the state 4 but none from the state 3. □

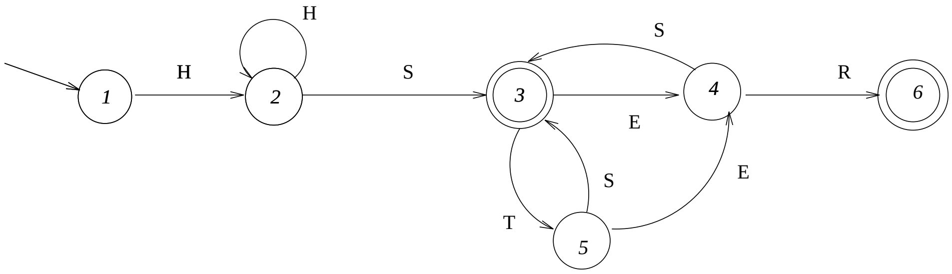

Figure 5.10: Automaton with four orbits.

Definition 5.35 For a state q of a finite automaton M, let the *orbit automaton* M_q of q be the automaton obtained by restricting the state set of M to $O(q)$ with initial state q and with the gates of $O(q)$ as the final states of M_q. The language of M_q is called the *orbit language* of q. The languages $L(M_q)$ are also called the orbit languages of M. We also consider a larger subautomaton of M related to q: the finite automaton M^q is M with its state set restricted to the states reachable from q and with q as the initial state. □

Example 5.36 In the automaton M in Figure 5.10, the orbit automaton M_5 contains the states 3, 4, and 5, with the state 5 as the initial state and the states 3 and 4 as the final states, whereas the automaton M^5 contains the states 3, 4, 5, and 6, with the state 5 as the initial state and the states 3 and 6 as the final states. □

Definition 5.37 For a deterministic finite automaton M, a symbol a in Σ is *M-consistent* if there is a state $f(a)$ in M such that all final states of M have an a-transition to $f(a)$. A set S of symbols is M-consistent if each symbol in S is M-consistent. □

Definition 5.38 Let M be a finite automaton and S be a set of symbols. The *S-cut* M_S of M is constructed from M by removing for each $a \in S$ all a-transitions that leave a final state of M. □

Now the structural properties of a 1-unambiguous deterministic automaton can be stated as the following theorem.

Theorem 5.39 Let M be a minimal deterministic finite automaton and S be the set of M-consistent symbols. Then $L(M)$ is 1-unambiguous if and only if

1. M_S satisfies the orbit property and
2. All orbit languages of M_S are 1-unambiguous.

Furthermore, if M consists of a single, nontrivial orbit, and $L(M)$ is 1-unambiguous, M has at least one M-consistent symbol. □

Based on Theorem 5.39, Brüggemann-Klein and Wood present the following algorithm that can decide whether a given content model is 1-unambiguous or not.

Algorithm 5.40 1-unambiguous; the decision algorithm for 1-unambiguity.

Input: A minimal deterministic finite automaton $M = (Q, \Sigma, \delta, I, F)$.
Output: *true*, if M is 1-unambiguous, *false*, otherwise.
Method:
1. compute $S := \{a \in \Sigma \mid a \text{ is } M\text{-consistent}\}$;
2. **if** M has a single, trivial orbit **then** return true;
3. **if** M has a single, nontrivial orbit **and** $S = \emptyset$ **then** return false;
4. compute the orbits of M_S;
5. **if not** OrbitProperty(M_S) **then** return false;
6. **for** each orbit K of M_S **do**
7. choose $x \in K$;
8. **if not** 1-unambiguous($(M_S)_x$) **then** return false;
9. return true.

Example 5.41 When deciding the unambiguity of the automaton M in Figure 5.11 the set of M-consistent symbols is in the beginning empty. Since there are several orbits, we compute the orbits of M. Note that $M_S = M$, if S is empty. The automaton fulfills the orbit property. Thus, we check recursively the unambiguity of each of the orbit automata.

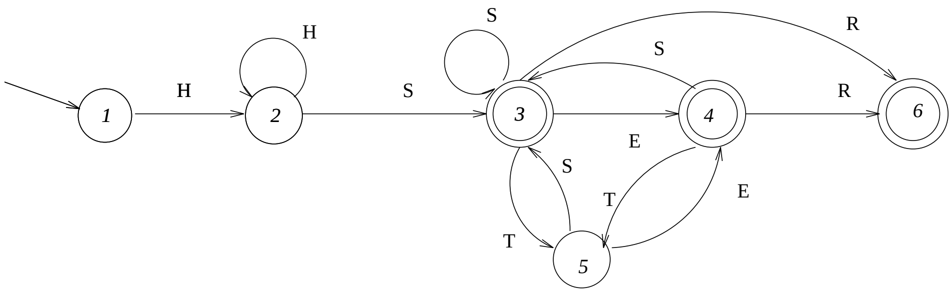

Figure 5.11: A 1-unambiguous automaton.

The orbits $\{1\}$ and $\{6\}$ are trivial, and hence 1-unambiguous. The orbit automaton M_2 has one consistent symbol, namely H, and the $\{H\}$-cut of the orbit automaton is trivial. Hence, the orbit automaton M_2 is 1-unambiguous.

The orbit automaton $M_3 = M_4 = M_5$ has two consistent symbols, S and T. In Figure 5.12 we can see the orbit automaton when the S- and T-transitions have been removed, i.e., the $\{T, S\}$-cut. Now we have three trivial orbits, and hence, $(M_3)_{\{S,T\}}$ and the whole automaton are 1-unambiguous. $\square$

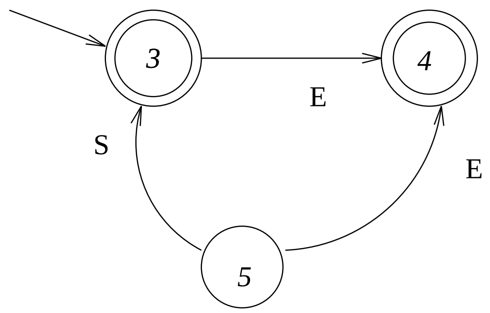

Figure 5.12: The cut $M_{\{T,S\}}$, consistent symbols have been removed.

Theorem 5.42 [BKW94] Algorithm 5.40 can be implemented to run in time $\mathcal{O}(e^2)$.

□

5.2.2 Disambiguation of automata

If the language of a content model is not 1-unambiguous, we have to disambiguate the corresponding automaton. Disambiguation generalizes the language, i.e., the resulting content model accepts more element structures than the original one. Hence, all the existing documents are still valid.

Algorithm 5.40 can be modified so that the automaton is transformed into a 1-unambiguous automaton if it is not 1-unambiguous originally. There are two cases when the 1-unambiguity test fails: first, we have a single, nontrivial orbit automaton M, but there are no M-consistent symbols, and second, the automaton does not have the orbit property.

The modified algorithm is the following. Contrary to the previous algorithm, we allow the automaton to be temporarily nondeterministic.

Algorithm 5.43 Disambiguate; the transformation algorithm for 1 - unambiguity.

Input: A minimal deterministic finite automaton $M = (Q, \Sigma, \delta, I, F)$.
Output: An automaton $M' = (Q', \Sigma, \delta', I', F')$, such that $L(M) \subseteq L(M')$ and $L(M')$ is 1-unambiguous.
Method:

$Q' := Q; \delta' := \delta; I' := I; F' := F;$
compute $S := \{a \in \Sigma \mid a \text{ is } M'\text{-consistent}\}$;
if M' has a single, trivial orbit **then** exit;
if M' has a single, nontrivial orbit **and** $S = \emptyset$;
then
  choose a symbol a such that for some final state q of M'
  there is a transition $f(a) \in \delta'(q, a)$;
  for each final state q' of M'
    if $q'' \in \delta'(q', a)$, with $q'' \neq f(a)$ and $q'' \in M'$
    then merge states $f(a)$ and q'';
    else $\delta'(q', a) := \delta'(q', a) \cup \{f(a)\}$;
  $S := \{a\}$;
compute the orbits of M'_S;
ForceOrbitProperty(M'_S);
for each orbit K of M'_S **do**
  choose $x \in K$;
  Disambiguate($(M'_S)_x$).

How to choose a consistent symbol? In our implementation we use a symbol a such that there are a maximum amount of final states q' such that $f(a) \in \delta(q', a)$, but any symbol leaving a final state is a possible candidate. A good choice is also a symbol that leaves all the final states but not to the same state. The choice of the consistent symbol affects the result of the disambiguation.

Algorithm 5.44 ForceOrbitProperty.

Input: An automaton M.
Output: A minimal automaton M' that has the orbit property and $L(M) \subseteq L(M')$.
Method:

for each orbit K of M
  let $g_1, \ldots, g_k$ be the gates of K;
  if there exists a gate $g_i \in F$
  then for each g_i
    $F := F \cup \{g_i\}$;
  for each ordered pair of gates (g_i, g_j)
    if there exists a symbol a such that $q \in \delta(g_i, a)$,
    with $q \in Q \backslash O(g_i)$, and $q \notin \delta(g_j, a)$
    then
      if $q' \in \delta(q_j, a)$, s.t. $q' \neq q$
      then $\delta(g_j, a) := \delta(g_j, a) \cup \{q\}$.

Example 5.45 The automaton M in Figure 5.13 does not have the orbit property, since in the orbit automaton M_3 the gate 4 is non-final while the gate 3 is final, and additionally, both states have a transition labelled R, but not to the same state. Hence, the algorithm first makes the gate 4 final and then adds new R-transitions from the state 3 to the state 6 and from the state 4 to the state 7.

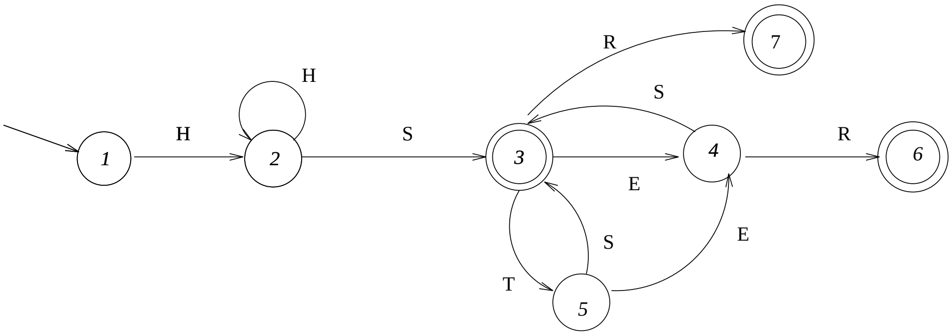

Figure 5.13: An ambiguous automaton.

All the other orbit automata are 1-unambiguous except the orbit automaton M_3, since there are no M_3-consistent symbols. We choose T to be consistent, add a transition from the state 4 to the state 5, and construct a {T}-cut by removing T-transitions (Figure 5.14).

Figure 5.14: a) T is consistent. b) $\{T\}$-cut.

The automaton still has one non-trivial orbit that is not 1-unambiguous. We choose S to be consistent, and now the {S}-cut has only trivial orbits.

Figure 5.15: a) S is consistent. b) $\{S\}$-cut.

□

Disambiguation of an orbit automaton may cause nondeterminism in the containing automaton. Hence, after disambiguation we have to merge the states causing nondeterminism and repeat the disambiguation of the whole automaton until the automaton is deterministic. In Example 5.45 we merge states 6 and 7, after which the next disambiguation step does not alter the automaton. The resulting 1-unambiguous automaton can be seen in Figure 5.11.

Since the generalizing of orbits creates new transitions, it is possible that the automaton does not remain (k, h)-contextual. Hence, as the non-determinism is deleted, also the new similar paths have to be combined.

The algorithm converges: if the automaton is not 1-unambiguous, the algorithm either adds new arcs or merges states. Since both an automaton with one state only and a totally connected automaton are 1-unambiguous, the algorithm stops in these situations at last.

Theorem 5.46 Algorithm 5.43 can be implemented to run in time $\mathcal{O}(e^2)$, where e is the number of the transitions in the input automaton.

Proof Computing the orbits requires two traversals of the transitions. The number of subautomata in which the orbits have to be computed is also $\mathcal{O}(e)$. □

5.2.3 Conversion into a content model

If an automaton accepts a 1-unambiguous language, it is possible to construct a corresponding 1-unambiguous regular expression.

We can represent the interconnections among the orbits by constructing a reduced automaton for M. The states of the reduced automaton are the orbits of M. There is a transition from state C to a different state C' of the reduced automaton if there is a transition in M from some state in the orbit C to some state in the orbit C'. The reduced automaton cannot have any cycles, because if there were a cycle, then all the orbits in the cycle would be one orbit, meaning that the orbits were not properly computed.

Now we can construct the expression for the automaton M by constructing the expression for the reduced automaton. Brüggemann-Klein and Wood [BKW94] give the following method. First we assume that the 1-unambiguous regular expressions for the orbits can be constructed.

We assume that M has more than one orbit and consider the orbit $O(q_0)$ of the initial state q_0. Let $b_1, \ldots, b_n$ be the distinct symbols of the transitions that leave $O(q_0)$. Since M satisfies the orbit property, there are states $q_1, \ldots, q_n$ outside $O(q_0)$ such that for each i, all gates of $O(q_0)$ have a b_i-transition to q_i, and there are no other outgoing transitions from $O(q_0)$ to the outside. Since M is deterministic, M_{q_0} has no b_i-transition from a final state. If E_0 and E^i are 1-unambiguous expressions that denote the languages $L(M_{q_0})$ and $L(M^{q_i})$, respectively, then the 1-unambiguous expression

$$E = E_0, (b_i, E^i \mid \ldots \mid b_n, E^n)$$

denotes $L(M)$.

Now we will show how the regular expressions for the orbits can be constructed. Clearly a regular expression for a trivial orbit is an empty string ϵ. For a single, nontrivial orbit automaton M, Brüggemann-Klein and Wood give the following construction method. If E_S and $E_S^{f(a_i)}$ are 1-unambiguous expressions denoting $L(M_S)$ and $L(M_S^{f(a_i)})$, where $S = \{a_1 \ldots a_k\}$ are the consistent symbols, then the 1-unambiguous expression E_S, $(a_1, E_S^{f(a_1)} \mid \ldots \mid a_k, E_S^{f(a_k)})^*$ denotes $L(M)$.

Example 5.47 Let E be a regular expression denoting the language $L(M)$ of $M = (Q, \Sigma, \delta, I, F)$, and let E_i, E^i, and E^i_S be regular expressions denoting the languages $L(M_{q_i})$, $L(M^{q_i})$, and $L(M^{q_i}_S)$, respectively, with $q_i \in Q$, and $S \subset \Sigma$. Now we can construct a 1-unambiguous expression for the automaton in Figure 5.11 in the following way.

$$\begin{aligned}
E &= E_1,\ h,\ E^2 \\
E_1 &= \epsilon \\
E^2 &= E_2,\ s,\ E^3 \\
E_2 &= \epsilon,\ (h,\ \epsilon)^* = h^* \\
E^3 &= E_3,\ (\epsilon \mid r,\ E^6) \\
E_3 &= e?,\ (s,\ E^3_{\{s,t\}} \mid t,\ E^5_{\{s,t\}})^* \\
E^3_{\{s,t\}} &= e? \\
E^5_{\{s,t\}} &= (s,\ (e \mid \epsilon)\ \mid e) = (s,\ e? \mid e) \\
E^6 &= \epsilon
\end{aligned}$$

and finally

$$\begin{aligned}
E_3 &= e?,\ (s,\ e?\ \mid\ t,\ (s,\ e? \mid e))^* \\
E &= h,\ h^*,\ s,\ (e?,\ (s,\ e?\ \mid\ t,\ (s,\ e? \mid e))^*),\ r?
\end{aligned}$$

□

5.3 Refining operations

After the basic generalizing phase the resulting content model may not satisfy the needs of the application, as described in Section 3.3.1. Especially, the content model may be difficult to understand. Hence, we have developed several *refining operations*, the main function of which is to obtain more readable content models without sacrificing the other needs. To achieve this goal, these operations both continue with the generalization of the resulting content model in a restricted way and attempt to divide the content model into smaller parts.

In this section we introduce a selection of operations and shortly sketch some possible implementation approaches. The operations include *isolating model groups* (Sections 5.3.1 and 5.3.2), *finding inclusions* (Section 5.3.3), using *frequency information* (Section 5.3.4), and *local trivialization* (Section 5.3.5).

5.3.1 Isolating model groups

If the document structure is very complicated, it is often useful to be able to introduce more nesting in the structure than present in the examples. For instance, in our dictionary entries there are clearly elements that form the first part of the entry (Headword, Inflection, Consonant_gradation) and others that appear in the end (Sense, Example). Since both of these parts vary a lot, processing them separately reduces the complexity of the productions remarkably. The separation can be done by isolating model groups.

Isolation takes as input a set of elements and a name for a new element, and replaces the model groups containing these elements by the new element. The replaced model groups are then used to construct an element declaration for the new element.

Example 5.48 Consider the following element declaration:

```
<!ELEMENT Entry_1 - - (Headword, Inflection,
                     ( ( Technical_field |
                      Consonant_gradation,
                      Pronunciation_instructions?),
                      (Sense, Example)*, Reference? |
                       Pronunciation_instructions, (Technical_field,
                       (Sense, Example)*, Reference? | Baseword)))>
```

If the elements appearing with the headword and the elements at the end are isolated we get the following simpler declarations:

```
<!ELEMENT Entry_1        - - (Headword, Headword_apps,
                                Rest_of_the_entry?) >
<!ELEMENT Headword_apps  - - (Inflection, (Consonant_gradation,
                              Pronunciation_instructions? |
                              Pronunciation_instructions)?) >
<!ELEMENT Rest_of_the_entry - - (Technical_field?,
                                  (Sense, Example)*, Reference? |
                                 Baseword )  >
```

□

Assume that we want to isolate elements $N = \{N_1, \ldots N_p\}$ from the content model of A, and replace them with name B.

Isolation consists of two parts (Algorithm 5.49). First, the transitions labelled N_i are copied to a new automaton. Second, the labels N_i are replaced with B in M_A.

Algorithm 5.49 Isolate model groups.

Input: An automaton $M_A = (Q, \Sigma, \delta, I, F)$; a set N of elements to be isolated; the name B of the new model group.
Output: An automaton $M_{A'} = (Q', \Sigma', \delta', I', F')$ such that there is a path $(q'_1, \ldots, q'_p)$ in M'_A with $\delta'(q'_i, a_i) = q'_{i+1}$ if and only if there is a corresponding path $(q_1, \ldots, q_p)$ in M_A with $\delta(q_i, a_i) = q_{i+1}$, where $a_i \in N$ and $1 \leq i \leq p-1$; an automaton $M''_A = (Q'', \Sigma'', \delta'', I'', F'')$ such that for each pair of states q_1 and $q_2 \in M_A$ with $\delta(q_1, a) = q_2$ and $a \notin N$, there are states q''_1 and $q''_2 \in M''_A$ with $\delta''(q''_1, a) = q''_2$. Additionally, for each maximal path $(q_1, \ldots, q_p)$ in M_A with $\delta(q_i, a_i) = q_{i+1}$, where $a_i \in N$, there are states q''_1 and $q''_p \in M''_A$ such that $\delta(q''_1, B) = q''_p$.
Method:
$M_{A'} :=$ Extract(M_A, N);
$M_{A''} :=$ Replace(M_A, N, B).

In Algorithm 5.50 all the N_i-transitions and their source and goal states are picked up to form a new automaton. A state that may start a path containing N_i-transitions becomes an initial state. Similarly, the algorithm finds the final states. Also the original initial and final states preserve their status, if they belong to the new automaton (Lines 10–11). If there are more than one initial state, the states are merged (Lines 12–13), since the automaton should be deterministic.

Algorithm 5.50 Extract.

Input: An automaton $M_A = (Q, \Sigma, \delta, I, F)$; a set N of elements to be isolated.
Output: An automaton $M_{A'} = (Q', \Sigma, \delta', I', F')$ as in Algorithm 5.49.
Method:
1. $I' := \emptyset$; $F' := \emptyset$; $\delta' := \emptyset$; $Q' := \emptyset$;
2. **for** each transition $\delta(q_1, a) = q_2$
3.   **if** $a \in N$
4.   **then** $Q' := Q' \cup \{q_1, q_2\}$;
5. $\delta'(q_1, a) := q_2$;
6.   **if** there exists a state r_1 such that $\delta(r_1, b) = q_1$ and $b \notin N$
7.   **then** $I' := I' \cup \{q_1\}$;
8.   **if** there exists a state r_2 such that $\delta(q_2, c) = r_2$ and $c \notin N$
9.   **then** $F' := F' \cup \{q_2\}$;
10. $F' := F' \cup (F \cap Q')$;
11. **if** $I \subseteq Q'$ **then** $I' := I' \cup \{I\}$;
12. **if** $|I'| > 1$;
13. **then** merge the states of I';
14. Remove the possible nondeterminism by merging states.

When replacing the isolated nonterminals (Algorithm 5.51) the automaton is contracted by shortening the paths containing labels in N to the length of 1 only. This is done by repetitively merging the goal states of two consecutive transitions labelled B. Replacements may also cause nondeterminism that has to be removed.

Algorithm 5.51 Replace.

Input: An automaton $M_A = (Q, \Sigma, \delta, I, F)$; a set N of elements to be isolated; the name B of the new model group.
Output: An automaton $M_{A''} = (Q'', \Sigma, \delta'', I'', F'')$ as in Algorithm 5.49.
Method:
1. $Q'' := Q$; $\delta'' := \delta$;
2. **for** each transition $\delta''(q_1, a) = q_2$ with $a \in N$
3. $\quad\delta''(q_1, a) := \emptyset$;
4. $\quad\delta''(q_1, B) = q_2$;
5. **for** each triple of states q_1, q_2, and $q_3 \in Q''$
   with $\delta''(q_1, B) = q_2$ and $\delta''(q_2, B) = q_3$ **or**
   $\quad\delta''(q_1, B) = q_2$ and $\delta''(q_1, B) = q_3$
6. $\quad$merge states q_2 and q_3 in Q'';
7. $I'' := I \cap Q''$;
8. $F'' := F \cap Q''$.

Example 5.52 Assume we want to isolate the elements in Example 5.48, i.e., N = *{Inflection, Consonant_ gradation, Pronunciation_ instructions}*, and the new element name is *Headword_ apps.*

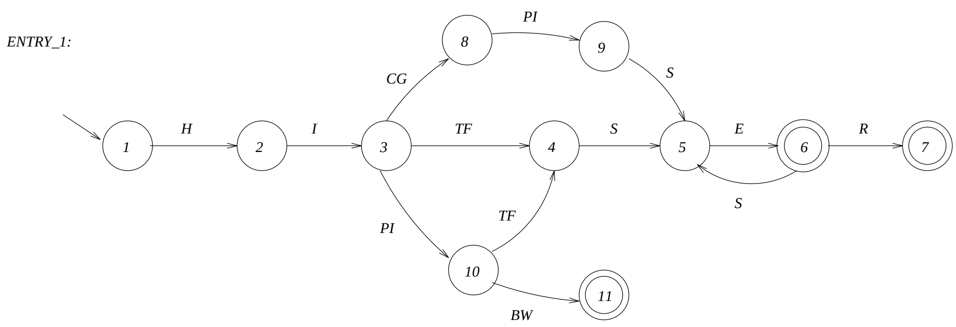

Figure 5.16: The automaton before isolation.

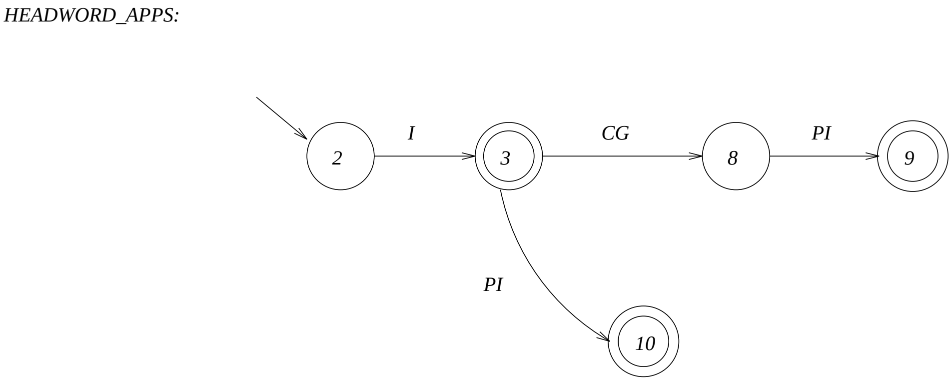

Figure 5.17: The new automaton isolated.

The original automaton for *Entry_1* can be seen in Figure 5.16, and the new automaton for *Headword_apps* in Figure 5.17. When replacing the elements in N, Algorithm 5.51 first replaces all the occurrences of the labels in N with Headword_apps (Figure 5.18), and then starts to contract the automaton. The result can be seen in Figure 5.19. □

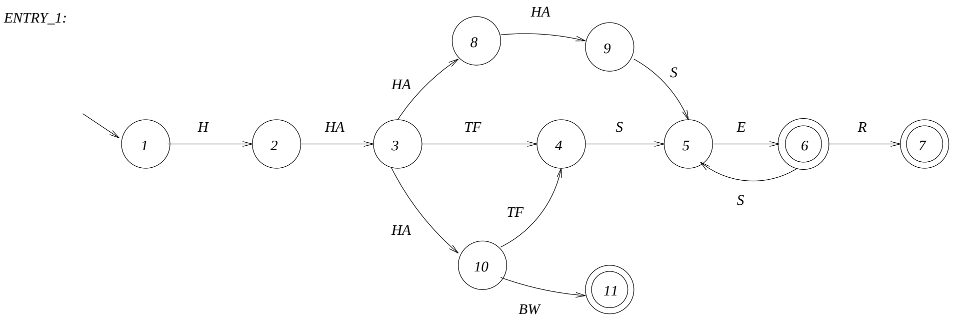

Figure 5.18: The original automaton after replacement.

Theorem 5.53 Algorithm 5.49 can be implemented to run in time $\mathcal{O}(e)$, where e is the number of transitions in the input automaton.

Proof The isolation part traverses all the transitions of the input automaton once. While processing a transition it can check in constant time

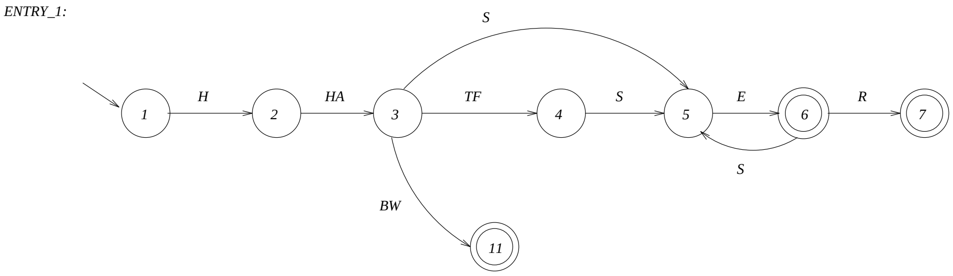

Figure 5.19: The resulting automaton after contracting.

whether the transition starts or ends a sequence to be isolated. Also the adding of a transition to the new automaton can be done in constant time. Similarly, the replacing part traverses all the transitions, and while processing a transition it can check locally whether there are states to be merged. Thus, the running time is $\mathcal{O}(e)$. □

There are two ways how the isolation of model groups can be used in the SGML document structuring process. First, the new expression, i.e., content model, can be described as a parameter entity that can then be included in various other content models as a kind of short-hand notation. In the case of Example 5.48 the definitions would look like the following:

```
<!ELEMENT Entry_1           (Headword, %Headword_apps; ,
                                       %Rest_of_the_entry;?)>

<!ENTITY % Headword_apps  "Inflection,
                                 (Consonant_gradation,
                                    Pronunciation_instructions? |
                                  Pronunciation_instructions)">
<!ENTITY % Rest_of_the_entry    "(Technical_field?,
                                  (Sense, Example)*, Reference?
                                 | Baseword)" >
```

Use of entities does not affect the document instances. The second way is to form elements instead of entities. Then also the instances have to be updated, and the isolation method should generate a parser that can convert original documents to contain the new structures. The second way could be useful, if the isolation reveals that some essential structure is not explicit in the original documents.

5.3.2 Automatic isolation

In batch processing, or if the user does not have any knowledge of the structure, it is useful to be able to automatically discover the strongly related elements, i.e., the candidate sets for isolation. In our method we apply the clustering schema of [PR87] to our application domain.

The goal of clustering is to find, given a set S of elements, a clustering, i.e., a partition of S such that each cluster, i.e., each set in the partition, covers similar elements, i.e. is *tight*, and different clusters cover dissimilar elements, i.e. have large *distance*. In our case similarity should be higher if two elements appear often together, and lower, if they seldom appear together. The quality of the clustering is in the method measured by a *goodness function* G that is defined as a minimum of *distance* and *tightness functions* D and T, respectively, i.e., $G(C) = \min\{D(C), T(C)\}$ for a clustering C.

We present one possible way of defining the distance and tightness functions. Let M_A be the automaton for some content model, and let $N = \{N_1, \ldots, N_p\}$ be the elements appearing in this content model. First, we define the following measures.

Definition 5.54 Let a and b be elements.

- $n_{ab} = \begin{cases} 1, & \text{if there is a path labelled } (a,b) \text{ in } M_A \\ 0, & \text{otherwise} \end{cases}$
- $n_a^{left} = \sum_{i=1}^{p} n_{aN_i}$ and
- $n_a^{right} = \sum_{i=1}^{p} n_{N_ia}$.

□

The distance function can be defined as follows. Let $C = \{c_1, \ldots, c_k\}$ be a clustering of S, i.e., $c_i \subseteq S$, $c_i \cap c_j = \emptyset$ and $\cup_i\, c_i = S$. Let D be a distance function

$$D(C) = \min_{1 \le i \ne j \le k} \{d(c_i, c_j)\}.$$

If the clustering C has only one cluster, define $D(C) = 0$. Hence, the distance quality of the clustering is measured by the distances of the pairs of clusters. The distance of two clusters is defined to be the minimal distance between their elements:

$$d(c_i, c_j) = \min_{a \in c_i, b \in c_j} \{d(a, b)\}.$$

Finally, the distance of two elements is defined by

$$d(a,b) = \begin{cases} 0, & \text{if } a = b \\ 1, & \text{if } a \text{ and } b \text{ originate from the same orbit, } a \neq b \\ n_a^{left} \times n_b^{right}, & \text{if } n_{ab} = 1, a \neq b \\ \textit{undefined}, & \text{otherwise} \end{cases}$$

The special treatment of the elements appearing in the same orbit has its main implications to the tightness funtion presented below. The distance of elements that do not appear in the same orbit is the probability that they appear in the given order, among all the possible combinations in which these elements may appear.

Define the tightness function T by

$$T(C) = \min_{c_i \in C}\{t(c_i)\}.$$

The tightness of the whole clustering is the minimum tightness of the clusters. On the other hand, the tightness of a cluster is measured by the largest distance between two elements that have appeared at least once together. Since tightness should be higher if the distance is smaller, the distance is subtracted from the maximal distance between two elements in the set.

$$t(c_i) = max\_distance - \max_{a,b \in c_i, n_{ab}=1}\{d(a,b)\},$$

with max_distance= $\max_{a,b\in\Sigma,n_{ab}=1}\{d(a,b)\}$. As the elements of the same orbit have a distance 1, a cluster that contains such elements only always has the highest possible tightness. If there is also some other elements, the tightness is determined by the distances of these elements.

Example 5.55 Consider the automaton M_{ENTRY_2} in Figure 5.20. The values of n_{ab} for each element a and b in the automaton are given in Table 5.3.2. The value of n_a^{left} is the sum of the values on the line labelled a, and the value of n_a^{right} is the sum of the values on the column labelled a. For each pair of elements (a,b) such that $n_{ab} = 1$, the value of $d(a,b)$ can be seen in Table 5.3.2.

□

The algorithm to find the clusters is the following.

n_{ab}	H	I	CG	S	TF	E	R	EB	BW
H	0	1	0	1	1	0	0	0	0
I	0	0	1	1	0	0	0	1	0
CG	0	0	0	1	0	0	1	0	0
S	0	0	0	0	1	0	0	0	0
TF	0	0	0	0	0	1	0	0	0
E	0	0	0	1	0	0	1	0	0
R	0	0	0	0	0	0	0	1	1
EB	0	0	0	0	0	0	0	0	0
BW	0	0	0	0	0	0	0	1	0

Table 5.1: The values n_{ab}.

a	b	$d(a,b)$
H	I	3
H	S	12
H	TF	6
I	CG	3
I	S	12
I	EB	9
CG	S	8
CG	R	4
S	TF	1
TF	E	1
E	S	1
E	R	4
R	EB	6
R	BW	2
BW	EB	3

Table 5.2: Distances $d(a,b)$ of elements in the automaton M_{ENTRY_2} of Figure 5.20.

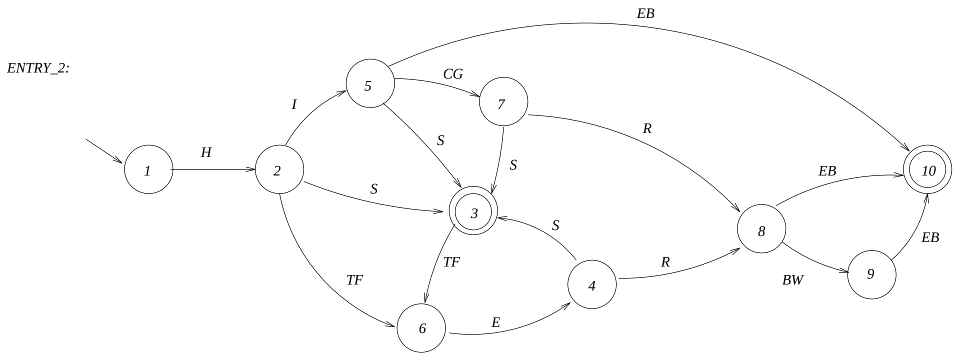

Figure 5.20: The automaton M_{ENTRY_2} before automatic isolation

Algorithm 5.56 Finding clusters of elements.

Input: An automaton M; for each $a, b \in \Sigma$ the values $n_{ab}, n_a^{left}, n_a^{right}$.
Output: A clustering C of elements.
Method:
1. $C_1 :=$ Initial_clustering(M);
2. $i := 1$;
3. **while** $|C_i| > 1$ **do**
4. compute $d(c_j, c_k)$ for each $c_j, c_k \in C_i$;
5. let $c, c' \in C_i$ be such that $d(c, c') = D(C_i)$;
6. $C_{i+1} = (C_i \setminus \{c, c'\}) \cup \{\{c, c'\}\}$;
7. $i := i + 1$;
8. Output any $C \in \{C_1, \ldots, C_{i-1}\}$ such that $G(C)$ is maximum.

We have chosen the orbits of the automaton to be the initial clustering, since appearing in the same orbit is a clear indication of the closeness. Another possibility would be to start from the clustering in which each element has its own cluster.

Algorithm 5.57 Initial_clustering.

Input: An automaton M.
Output: A clustering C_1 of elements.
Method:

```
C_1 := ∅;
compute the orbits O = {o_1, ... o_n} of M;
for each orbit o_i
    let c_i be the label set of o_i;
C_1 := C_1 ∪ {c_i};
for each set c_i and c_j such that c_i ∩ c_j ≠ ∅, i ≠ j do
    C_1 := (C_1 \ {c_i, c_j) ∪ {{c_i, c_j}}.
```

Let the *label set of an orbit* be the set of elements found from the transitions between the states of an orbit. There is one non-trivial orbit in M_{ENTRY_2}, namely $\{3, 4, 6\}$, the label set of which is {S, TF, E}. Hence, the initial clustering C_1 contains a non-trivial cluster {S, TF, E} and the trivial clusters for the other elements (Table 5.3.2). The minimal distance between clusters is 2, i.e. the distance between clusters {R} and {BW}. Hence $D(C_1) = 2$.

The maximal distance between elements is 12, e.g. D(H, S) = 12. The tightness of clustering C_1 is the minimal tightness of the clusters. The tightness of the trivial clusters is the maximum value of all the clusterings, i.e. the maximal distance, here 12. Hence, the tightness of the nontrivial cluster {S, TF, E} becomes the tightness of the whole clustering. As the elements of {S, TF, E} originate from the same orbit, the distances between the elements are 1. Hence the tightness is $max\_distance - 1 = 11$.

To compute clustering C_2 we have to find two clusters that have a minimal distance. As mentioned above, these two clusters are {R} and {BW}. Hence, in C_2 these clusters are combined. We continue in a similar way, combine the nearest clusters, and compute the overall distances and tightnesses of the clusterings (Table 5.3.2). In the end we choose the clustering the goodness of which is the best. In our example this is clustering $C_5 = \{\{S, TF, E\}, \{R, BW, EB\}, \{H, I, CG\}\}$, with $G(C_5) = min(D(C_5), T(C_5)) = 4$.

Theorem 5.58 Algorithm 5.56 can be implemented to run in time $\mathcal{O}(e + |\Sigma|^3)$.

Proof The values n_{ab} can be computed in time $O(e)$, and the values n_a^{right} and n_a^{left} in time $\mathcal{O}(|\Sigma|)$. Creating the initial clustering takes $\mathcal{O}(e)$ time if the orbits of the automaton are computed. Algorithm 5.56 computes clusters at most $|\Sigma|$ times. At each step the distance between each element has to be computed, which takes $\mathcal{O}(|\Sigma|^2)$ time, since the distance between

C_1	{H}, {I}, {CG}, {R}, {BW}, {EB} {S, TF, E}	$D = 2$ $T = 11$
C_2	{H}, {I}, {CG}, {EB} {S, TF, E} {R, BW}	D = 3 T = 10
C_3	{H}, {I}, {CG} {S, TF, E} {R, BW, EB}	D = 3 T = 6
C_4	{CG} {S, TF, E} {R, BW, EB} {H, I}	D = 3 T = 6
C_5	{S, TF, E} {R, BW, EB} {H, I, CG}	D = 4 T = 6
C_6	{S, TF, E} {H, I, CG, R, BW, EB}	D = 4 T = 3

Table 5.3: Clusters found by the Algorithm 5.56

two elements can be computed in constant time. Similarly, the tightness of the clustering needs in the worst case $\mathcal{O}(|\Sigma|^2)$ time, if the distances between each pair have to be computed. □

After finding the best clusters the method can either proceed automatically, and isolate the elements of clusters as described above, or the clusters can first be presented to the user for evaluation. Similarly, the method can either create the names for the new entities or elements itself, or it can ask them from the user. Hence, the automatic isolation can be applied both in batch and interactive processing.

5.3.3 Discovering inclusions

If some element appears more than once with many of the other elements, this element may be floating. Then it could be reasonable to interpret this element as an inclusion. Examples of floating elements include footnotes, figures, and various cross-references.

To find the candidates for inclusion we can use the same measure n_{ab} as in the automatic isolation. We simply count the number of elements that appear next to a element A, and if the total number is greater than a certain proportion of the whole number of elements, with some threshold, then A is considered an inclusion.

Algorithm 5.59 Find inclusions.

Input: An automaton $M = (Q, \Sigma, \delta, I, F)$; a threshold t.
Output: A set *Inclusions* of elements.
Method:
1. **for** each $a \in \Sigma$
2. $count = (n_a^{left} + n_a^{right}) \; / \; 2;$
3. **if** $count \; / \; |\Sigma| \; > \; t$
4. $Inclusions := Inclusions \; \cup \{a\}.$

After a possible check by the user, the inclusion element has to be removed from the automaton and inserted to the set of inclusions that is attached to the automaton.

Algorithm 5.60 Isolating inclusions.

Input: An automaton M_A, a set *Inclusions* of elements.
Output: An automaton M'_A that is M_A with all the occurrences of $N \in Inclusions$ deleted. The set of inclusions $A.inclusions$ of the element A is updated.
Method:
1. Delete(M_A, *Inclusions*)
2. $A.inclusions := A.inclusions \cup Inclusions$

Algorithm 5.61 shows how an element is deleted from an automaton. Assume that there is a transition $\delta(q, a) = q'$, and element a is to be deleted. If there is some other transition from q to q' with some element not to be deleted, we just delete the a-transition. If there is no other transition, we contract the automaton by merging the states q and q'. Note that the states q and q' are merged even if there is some transition from q' to q. This guarantees that a deletion of a transition does not make any states useless.

Algorithm 5.61 Delete.

Input: An automaton $M_A = (Q, \Sigma, \delta, I, F)$; a set R of elements to be deleted.
Output: An automaton $M''_A = (Q'', \Sigma, \delta'', I'', F'')$ that is M_A with all the transitions with the label in R deleted.
Method:

```
Q'' := Q; δ'' := δ; I'' := I; F'' := F;
for each transition δ''(q1, a) = q2;
    if a ∈ R
    then δ''(q1, a) = ∅
        if there does not exist b ∈ Σ such that δ''(q1, b) = q2 with b ∉ R
        then
            merge states q1 and q2 in M''_A;
            remove the possible nondeterminism by merging states.
```

Example 5.62 Assume we want to check if the following element declaration contains any candidates for inclusions:

```
<!--      ELEMENTS CONTENT                              -->
<!ELEMENT Entry_3  (Headword,
                     (Inflection,
                        (Annotation, Consonant_gradation,
                         Example, Annotation |
                         Consonant_gradation, Annotation,
                         Sense, Annotation?, Example) |
                         Annotation,
                         (Sense, Annotation?, Example |
                          Inflection, Sense)))        >
```

The automaton M_{ENTRY_3} describing the content model can be seen in Figure 5.21, and the corresponding table n_{ab} in Table 5.62.

In the table we can see that element *Annotation (AN)* appears on the right-hand side of all the other elements, and on the left-hand side of all the elements except *Headword (H)*. If we give, e.g., a threshold 0.75, then $4.5/6 > 0.75$, and hence *Annotation* is a good candidate for an inclusion.

If *Annotation* is accepted as an inclusion, we have to delete it from M_{ENTRY_3}. In Figure 5.22 we can see the automaton after the *AN*-transition between states 3 and 4 has been deleted. Since there is no other transition between 3 and 4 the states have been merged. The deletion

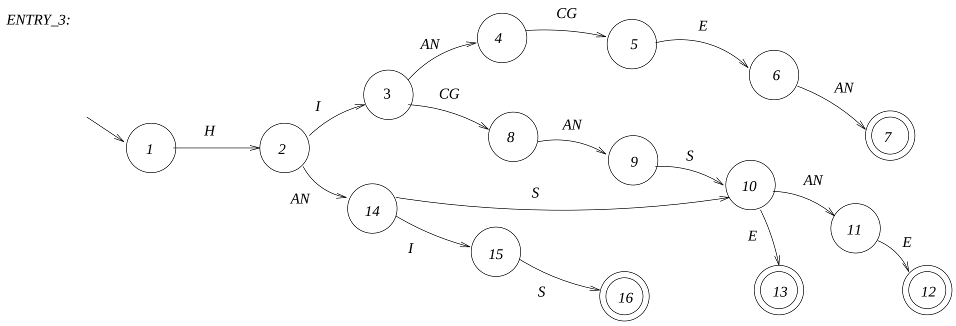

Figure 5.21: The automaton before finding inclusions.

n_{ab}	H	I	CG	AN	S	E
H	0	1	0	1	0	0
I	0	0	1	1	1	0
CG	0	0	0	1	0	1
AN	0	1	1	0	1	1
S	0	0	0	1	0	1
E	0	0	0	1	0	0

Table 5.4: The values n_{ab} for elements of Example 5.62.

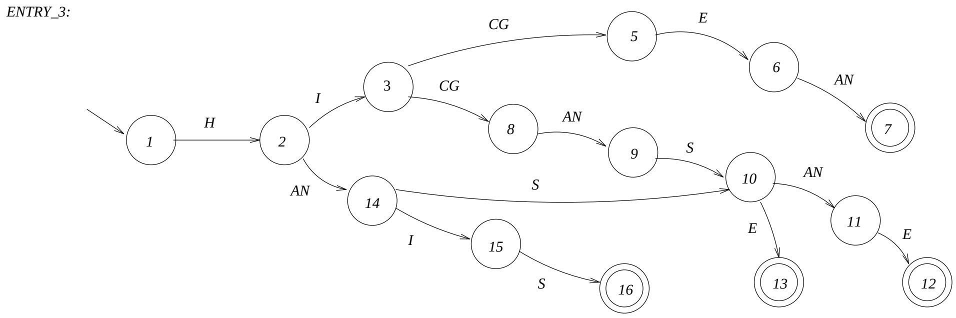

Figure 5.22: The first AN-transition has been deleted.

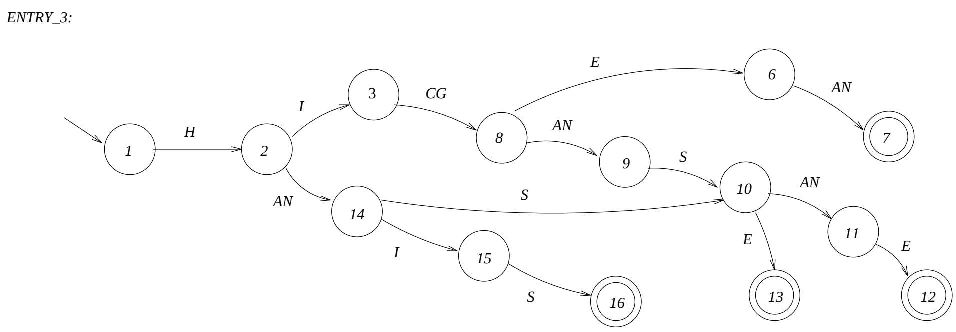

Figure 5.23: Nondeterminism has been removed by merging states 5 and 8.

introduces nondeterminism here: we get $\delta(3, CG) = \{5, 8\}$. The nondeterminism is removed by merging states 5 and 8 (Figure 5.23).

Similarly, the other AN-transitions are removed, and the automaton is contracted. The resulting automaton can be seen in Figure 5.24, and the corresponding element declaration is the following:

```
<!--      ELEMENTS CONTENT                    (EXCEPTIONS)?   -->
<!ELEMENT Entry_3  (Headword,
                     (Inflection,
                      (Sense |
                       Consonant_gradation,
                       Sense?, Example)  |
                       Sense, Example))        +(Annotation)    >
```

□

Theorem 5.63 Algorithms 5.59 and 5.61 can be implemented to run in time $\mathcal{O}(e + |\Sigma|)$, where e is the number of transitions.
Proof The values n_{ab} can be computed in time $\mathcal{O}(e)$. To find the inclusions all the elements are checked, which takes $\mathcal{O}(|\Sigma|)$ time. The deletion of the elements can be done with one traversal in the automaton.

5.3.4 Using frequency information

Frequency information can be used for quantifying the importance of different types of structures for the component. It is often desirable to obtain one

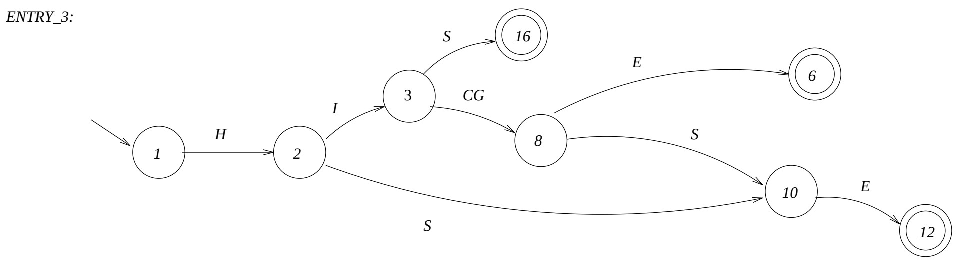

Figure 5.24: The final automaton without AN-transitions.

or a few productions which cover most of the examples, and then several productions which correspond to the exceptions.

Adding frequency information is easy: each transition is given a weight which is the number of examples that use this transition. A *weighted automaton* is an automaton in which each transition has a weight attached, that is the transition function δ is defined from $Q \times \Sigma^* \times N$ to the set of the subsets of the states in Q. In our method the user gives a threshold b which means that the program constructs a production that covers at least all the structures that appear b times in data. In addition to this production, another production is constructed to cover the remaining structures. The following algorithms search for transitions having the weight less than the given threshold. Such *weak transitions*, with necessary paths from the initial state and to a final state, are moved to a new automaton. In the following algorithms, a path consists of a set of quadruples (q, a, w, q'), where each quadruple corresponds to a transition $\delta(q, a) = q'$ with the weight of w.

Algorithm 5.64 Separate_weak_transitions.

Input: An automaton $M_A = (Q, \Sigma, \delta, I, F)$; a threshold t.
Output: An automaton $M_{weak} = (Q_{weak}, \Sigma, \delta_{weak}, I_{weak}, F_{weak})$ that contains the transitions the weights of which are less than the threshold t, and additionally transitions needed for complete paths; an automaton $M''_A = (Q'', \Sigma, \delta'', I'', F'')$ that is the automaton M_A with all the transitions with the weight of less than t removed.
Method:

$Q_{weak} := \emptyset$; $Q'' := Q$; $\delta'' := \delta$; $I'' := I$; $F'' := F$;
for each state $q \in Q$ **do** Marked(q) := *false*;
Weak_transitions(M''_A, M_{weak}, I, $\emptyset$);
for each useless state p in Q''
  for each transition $\delta'(p, a, w) = p'$
    $Q'' := Q'' \setminus \{p\}$;
    $Q_{weak} := Q_{weak} \cup \{p, p'\}$;
    $\delta_{weak}(p, a, w) := p'$.
$I_{weak} := I \cap Q_{weak}$; $F_{weak} := F \cap Q_{weak}$;

Algorithm 5.65 Weak_transitions.

Input: An automaton $M''_A = (Q, \Sigma, \delta, I, F)$; a state $q \in Q$; an automaton $M_{weak} = (Q_{weak}, \Sigma, \delta_{weak}, I_{weak}, F_{weak})$ that contains the transitions the weights of which are less than the threshold b, and additionally transitions needed for complete paths; a path *path* from the start state to q; a threshold t.
Output: M_{weak}; M''_A
Method:

if not Marked(q)
  Marked(q) := *true*;
  for each transition $\delta''(q, a, w) = q'$
    if $w > t$ **then**
      Weak_transitions(M'_A, M_{weak}, q', $path \cup (q, a, w, q')$, t);
    else
      $Q_{weak} := Q_{weak} \cup \{q, q'\}$;
      $\delta_{weak}(q, a, w) := q'$;
      $\delta''(q, a, w) := \emptyset$;
      $path\_to\_final :=$ Shortest_path_to_final(M''_A, q');
      $path := path \cup (q, a, w, q') \cup path\_to\_final$;
      for each quadruple $(q, a, w, q') \in path$
        $Q_{weak} := Q_{weak} \cup (q, q')$;
        $\delta_{weak}(q, a, w) := q'$.

Algorithm 5.64 starts a recursive depth-first search (Algorithm 5.65) from the initial state. If a weak transition from state q to state q' is found, its source and goal states q and q' are moved to a new automaton M_{weak}.

The states and transitions that form the traversed path from the initial state to q and the shortest path from q' to some final state are moved to M_{weak} as well. In the end, all the useless states and respective transitions are moved to a new automaton, since they have been accessed by weak transitions only.

The following Algorithm 5.66 outputs the shortest path from the state given as input to some final state. The algorithm traverses the automaton breadth-first by queueing the goal states of each state traversed. Here function *Enqueue(Queue, (q, path))* puts to the tail of the queue *Queue* a pair consisting of a state q and a path *path* from the state given as input to state q. Respectively, function *Dequeue(Queue)* takes from the front of the queue *Queue* a pair of a state and a path.

Algorithm 5.66 Shortest_path_to_final.

Input: An automaton $M_A = (Q, \Sigma, \delta, I, F)$, a state $q \in Q$.
Output: The shortest path *path* from q to a final state.
Method:

$path := \emptyset$;
Enqueue($Queue, (q, path)$);
repeat
    $(q, path) :=$ Dequeue($Queue$);
    if $q \in F$ **then** return path;
    else
        for each transition $\delta(q, a, w) = q'$ **do**
            Enqueue($Queue$, $(q', path \cup \{(q, a, w, q')\})$);
until $Queue = \emptyset$;
return $\emptyset$.

Example 5.67 Assume we want to find the weak transitions of the automaton M_{ENTRY_4} shown in Figure 5.25, given the threshold of weakness is 60. We traverse first the transitions $(1, 2)$ and $(2, 5)$, that are not weak. We continue with the transition $(5, 6)$, the weight of which is less than the threshold. Hence, we delete the transition from the automaton and move it to a new automaton. Then we find that the shortest path to a final state is the path $(6, 3)$, and copy this transition with the respective states to the new automaton.

We do not proceed with the depth-first search from state 6, but return to state 5. Here there is also another weak transition, namely transition $(5,3)$. As state 3 is a final state, we do not need to copy any states and transitions.

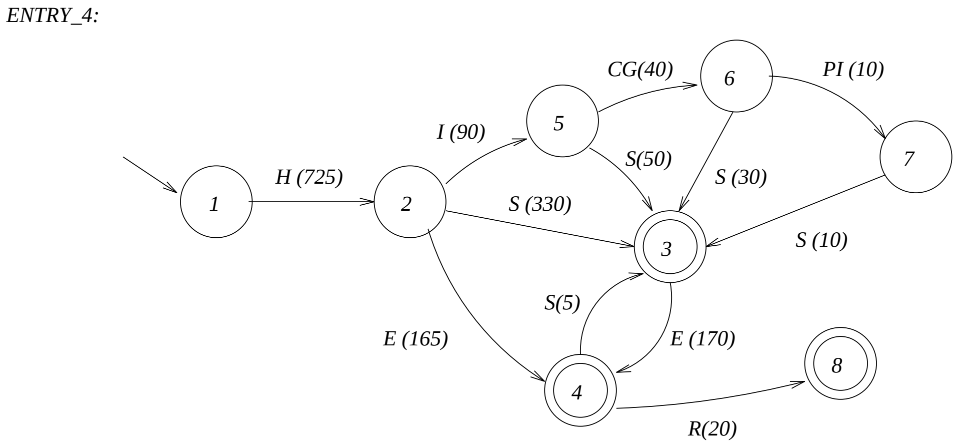

Figure 5.25: The automaton M_{ENTRY_4} before separation.

After the depth-first traversal the automaton M''_{ENTRY_4} looks like the one in Figure 5.26. As we can see, states $5,6,7$, and 8 have become useless, and therefore they are deleted from the automaton and moved to the new automaton that can be seen in Figure 5.27. The automaton M''_{ENTRY_4} has now been reduced to the one in Figure 5.28.

□

Theorem 5.68 Algorithm 5.64 can be implemented to run in time $\mathcal{O}(e^2)$ where e is the number of transitions.
Proof Algorithm 5.65 traverses at most all the transitions, and for each weak transition $\mathcal{O}(e)$ transitions are traversed in the search of a final state.

□

Frequency information can also be used in the following way. The user chooses the most common examples and let the learning program generate a grammar. The grammar is then used to parse rest of the example texts. If an example cannot be parsed, either the grammar is modified or the user changes the example. The latter gives the user a possibility to correct errors. The user should also be provided some statistical information, for instance the percentage of examples that the current grammar covers.

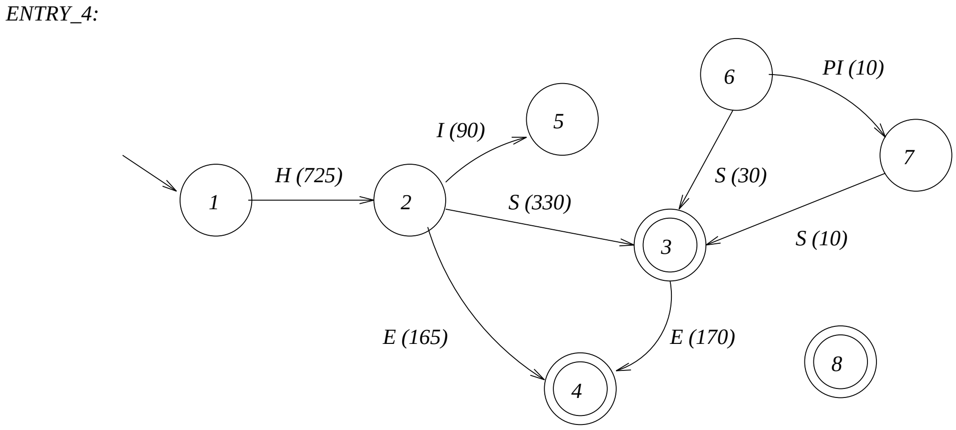

Figure 5.26: The automaton M''_{ENTRY_4} after weak transitions have been removed.

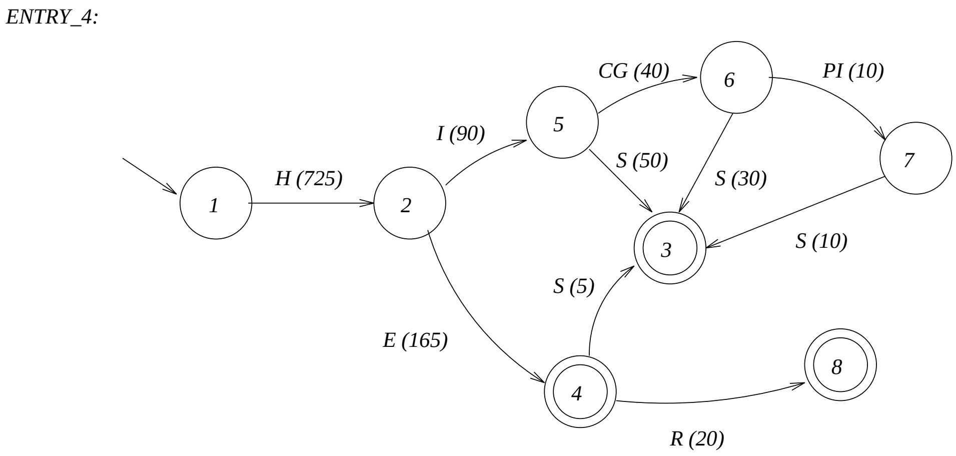

Figure 5.27: The new automaton for weak transitions.

ENTRY_4:

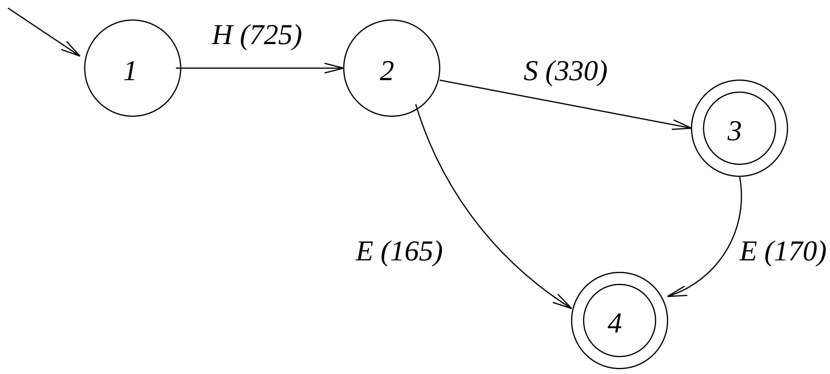

Figure 5.28: The automaton M_{ENTRY_4} after separation.

5.3.5 Local trivialization

The whole generalization process attempts to find a solution somewhere in between the exact set of instances and the trivial solution that contains all the structures. Basic generalization obtains some solution, but if it still contains too much variation, it can be further generalized. *Local trivialization* attempts to find the most complicated substructures and generalize them. If the complexity of some model group exceeds a given threshold, we form an optional and repeatable model group of all the elements included, i.e., generalize the corresponding model group to the trivial solution.

As non-trivial orbits produce the most complicated model groups, we can consider the connectivity of orbits to measure the complexity of the resulting model group. In the following algorithm we compare the number of transitions within an orbit to the maximal number of transitions this orbit could have.

Algorithm 5.69 Local trivialization of an orbit.

Input: An orbit automaton $M = (Q, \Sigma, \delta, I, F)$ that contains a single non-trivial orbit; a threshold t
Output: *true*, if M is totally connected, *false*, otherwise.
Method:

Let $nr\_of\_transitions$ be the number of transitions in M;
$label\_set := \{a \mid \delta(q, a) = q' \text{ with } q, q' \in Q, a \in \Sigma\}$;
$max\_transitions := |Q|^2$;
$connectivity := nr\_of\_transitions \;/\; max\_transitions$;
    if $connectivity \geq 1$
        return true
    else
    if $connectivity > t$
        for each pair of states q and $q' \in Q$
            for each label $a \in label\ set$
                $\delta(q, a) := q'$;
        return true;
    else return false.

Example 5.70 Assume the algorithm tries to trivialize locally the following content model. The corresponding automaton can be seen in Figure 5.29.

```
<!ELEMENT Entry_5 (Headword, (Inflection, Consonant_gradation?)?,
                   Sense, ( (Sense, (Technical_field | Example)? |
                              Example | Technical_field)
                   (Sense, (Technical_field | Example)? )*,
                   (Reference | Example) )? ) >
```

There is only one non-trivial orbit in the automaton, namely $\{3, 4, 6\}$. The number of transitions in this orbit is 5, while the maximal number of transitions in an orbit of three states is 9. Hence, if the threshold is, e.g., 0.5, the algorithm trivializes the orbit locally by adding all the possible transitions between states 3, 4, and 5. The result of trivialization can be seen in Figure 5.30.

If the algorithm returns true, we form an optional and repeatable model group from the label set of the orbit. In our example, this model group is

```
(Technical_field | Example | Sense)*.
```

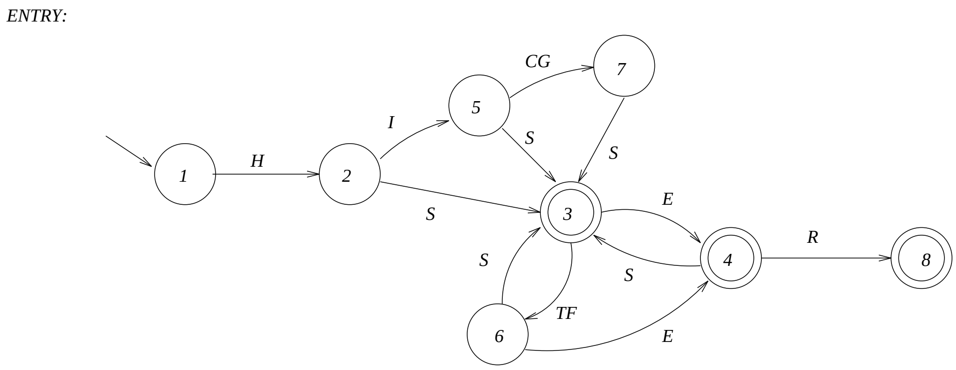

Figure 5.29: The automaton to be locally trivialized.

Hence, the resulting element declaration is the following:

```
<!ELEMENT Entry_5 (Headword, (Inflection, Consonant_gradation? )?,
                   ((Technical_field | Example | Sense)*,
                     Reference? )? )  >
```

□

Theorem 5.71 Algorithm 5.69 can be implemented to run in time $\mathcal{O}(e^2)$ where e is the number of transitions in the orbit.
Proof Counting the transitions of the orbit takes $\mathcal{O}(e)$ time. If the orbit is to be connected, at most e^2 transitions are added to the automaton. Adding of one transition can be done in constant time. □

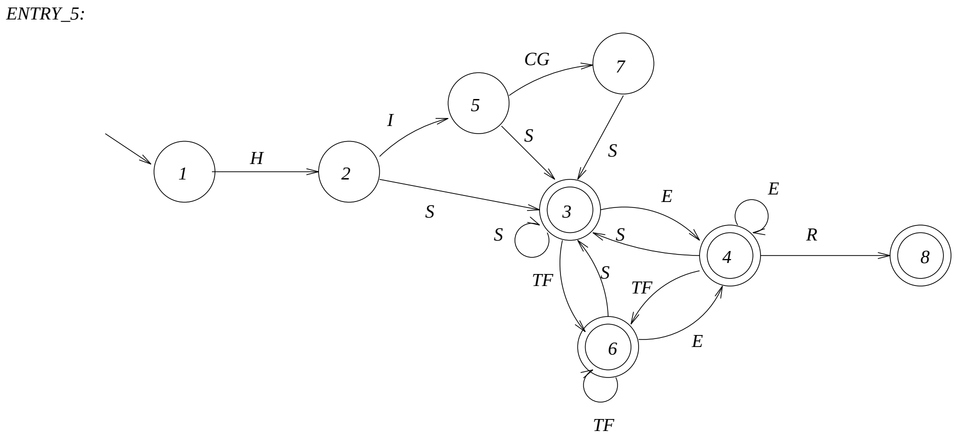

Figure 5.30: The orbit $(3, 4, 6)$ has been totally connected.

Chapter 6

Experimental results

We have implemented the method described above in Chapter 5, excluding automatic isolation and the discovery of inclusions. We have experimented with several document types and also with some artificial samples. In this chapter we present first two real applications (Section 6.1) and then consider some real and artifical samples that introduce cases problematic to the method (Section 6.2). Some ways to develop the method are also presented.

6.1 Two applications

In this section we present some experiments that illustrate how our method can be used to satisfy needs of varying applications.

6.1.1 Textbook

As mentioned earlier, we have converted one textbook on control engineering into SGML and structured it according to our ISO 12083 -based DTD [ISO94]. As we need a simpler DTD for the authors of new books, we generated a DTD for the book we already have, and now we can use this DTD as a basis for the author DTD. In Figure 6.1 we can see all the non-trivial element declarations resulted, i.e., the content models for all the elements seen on the right-hand side but not on the left-hand side are #PCDATA.

Our method produces a complete DTD, although in this case it is not totally necessary. After the basic generalization phase the method continued with local trivialization. It trivialized some model groups that have large variation, e.g., a part of the declaration for paragraph (`P`). All the processing was done without any interaction by the user.

```
<!ELEMENT ANSWER        - -   (TITLE?, P, ( (P, FIGGRP? | FIGGRP)
                               (P, FIGGRP?)*, FIGGRP? )? ) >
<!ELEMENT APPENDIX      - -   (NO, TITLE, (P | (SECTION)*) >
<!ELEMENT APPMAT        - -   (APPENDIX)* >
<!ELEMENT AUTHGRP       - -   (AUTHOR, AUTHOR) >
<!ELEMENT AUTHOR        - -   (FNAME, SURNAME | SURNAME, FNAME ) >
<!ELEMENT BACK          - -   (BIBLIST) >
<!ELEMENT BIBLIST       - -   (HEAD, (CITATION)*) >
<!ELEMENT BODY          - -   (CHAPTER)* >
<!ELEMENT BOOK          - -   (FRONT, BODY, APPMAT, BACK) >
<!ELEMENT CELL          - -   (P) >
<!ELEMENT CHAPTER       - -   (NO, TITLE, ((P)*, (EXAMPLE | FIGGRP)? )?,
                              (SECTION)*) >
<!ELEMENT CITATION      - -   (TITLE, (AUTHOR)*, (CORPAUTH)*,
                               OTHINFO?, DATE ) >
<!ELEMENT CORPAUTH      - -   (ORGNAME, CITY?) >
<!ELEMENT DFORMULA      - -   ( #PCDATA | FIGGRP, FIGGRP?) >
<!ELEMENT DOCUMENT      - -   (BOOK) >
<!ELEMENT EXAMPLE       - -   (P | TITLE, ((P | FIGGRP)*, ANSWER?)? ) >
<!ELEMENT EXERC         - -   (NO?, ((P)*, (FIGGRP, P? | ANSWER)? )? ) >
<!ELEMENT EXGROUP       - -   (TITLE?, ((EXERC)*, FIGGRP? )? ) >
<!ELEMENT FIGGRP        - -   ((FIG)*, TITLE?) >
<!ELEMENT FOREWORD      - -   (TITLE, (P)*) >
<!ELEMENT FRONT         - -   (TITLEGRP, FIGGRP, AUTHGRP, FIGGRP,
                               FOREWORD, PREFACE) >
<!ELEMENT ITEM          - -   (P)* >
<!ELEMENT LIST          - -   (HEAD, ITEM, (HEAD, ITEM)* | (ITEM)*) >
<!ELEMENT P             - -   ( #PCDATA | (DFORMULA | LIST | EMPH |
                               FORMULA | SUBSCR | SUPERSCR | FIGREF)*
                               (FIGGRP, FIGREF? | SECREF | SYMBOL)?
                               | FIGGRP | FORMREF | SECREF |
                               (SYMBOL, SYMBOL?)  | TABLE ) >
<!ELEMENT PREFACE       - -   (TITLE, (P)*) >
<!ELEMENT ROW           - -   (TSTUB?, CELL, CELL) >
<!ELEMENT SECTION       - -   (NO, TITLE, ((P | EXAMPLE | FIGGRP |
                               LIST)* (EXGROUP, (SUBSECT1)* | DFORMULA |
                                (SUBSECT1)*)? | (SUBSECT1)* )? ) >
<!ELEMENT SUBSECT1      - -   (NO, TITLE, ((P | FIGGRP | EXAMPLE)*,
                               EXGROUP?)? ) >
<!ELEMENT TABLE         - -   (TBODY) >
<!ELEMENT TBODY         - -   (ROW)* >
<!ELEMENT TITLE         - -   ( #PCDATA | (SUBSCR)*) >
<!ELEMENT TITLEGRP      - -   (TITLE) >
<!ELEMENT TSTUB         - -   (P) >
```

Figure 6.1: A DTD for one textbook

6.1.2 Dictionary data

Most challenging of our test cases has been the part $A - K$ of a Finnish dictionary [Suo90]. We converted the typographical tags of the dictionary, which consists of about 16000 entries, to structural tags, and obtained a set of 468 distinct structures. Every structure also received a frequency, i.e., the number of entries that the structure covers. We chose 55 of the most common structures (Figure 6.2), which together covered 14791 entries.

The dictionary was not originally designed for computer use, and therefore the structures of the entries have great variation. Even the editors cannot specify the desired structure for an entry. Hence, there is a strong need to gather information from the existing structures and use this knowledge to update instances to develop a more consistent structure. In this point this case differs from some otherwise similar cases, e.g., discovering the structure of ancient books or archives, where it is not desirable to change the structure.

Again, after the basic generalization with local trivialization the result was the following:

```
<!ELEMENT EN  (H, ((EX | TF | S)*, R? | I, ( (EX | TF | S)*, R?
               | CG, ((EX | TF | S)*, R? | (R, EX?)  | PrF | (BW, EX)
               | PaF, ((EX | TF | S)* R? )? )?  | (R, EX?)
               | PI, (S | ((TF | EX | S)*, R?) | R )?
               | PaF, ((EX | TF | S)*, R? )? | PrF
               | II, ((TF | EX | S)*, R? )?  | (BW, EX))? | (R,EX?)
               | PrF | PI, (S | (TF | EX | S)*, R? | R )? )?) >
```

This time we used some backgroung knowledge, i.e., that elements *H, I, CG, PI,* and *I* form the first part of each entry. Therefore we made one interactive isolation step and obtained the following declaration:

```
<!ELEMENT EN  ( %HP, (((EX | TF | S)*, R?)  | (R, EX?) | PrF |
               (BW, EX) | (PaF, (EX | TF | S)*, R? )?) )? ) >

<!ENTITY % HP         "(H, (I, (CG | PI | II)? | PI)? )" >
```

6.2 Evaluation of the method

In this section we evaluate the automatic DTD generation method by presenting both artificial and real samples that illustrate the features of the method. Unfortunately, no metrics for the goodness of a DTD exist, thus the evaluation is more or less based on the subjective feeling of readability. One possible measure for the complexity of a content model could be

```
2470 EN → H S                        1787 EN → H EX
1325 EN → H                          1122 EN → H I S
1056 EN → H S EX                     1031 EN → H I S EX
995 EN → H TF S                      574 EN → H I CG S EX
549 EN → H I TF S                    387 EN → H I EX
352 EN → H I CG S                    329 EN → H R
258 EN → H I TF S EX                 232 EN → H TF S EX
195 EN → H TF                        171 EN → H I R
138 EN → H I CG TF S                 125 EN → H I
117 EN → H TF EX                     100 EN → H PrF
97 EN → H I CG TF S EX               94 EN → H I PI S
92 EN → H EX S                       85 EN → H I CG R
84 EN → H TF R                       66 EN → H I S EX TF EX
54 EN → H I PaF S EX                 53 EN → H I TF R
51 EN → H I CG S EX TF EX            47 EN → H I CG PrF
46 EN → H I CG BW EX                 45 EN → H I S EX TF S EX
44 EN → H I PrF                      44 EN → H PI S
42 EN → H I EX S                     39 EN → H TF EX S
34 EN → H I PaF S                    34 EN → H I CG PaF S EX
34 EN → H I PI TF S                  31 EN → H I S TF S
30 EN → H I TF TF S                  29 EN → H I II TF S
29 EN → H I S EX S                   29 EN → H I BW EX
28 EN → H I CG S EX TF S EX          24 EN → H I CG EX
24 EN → H S EX S                     22 EN → H I R EX
22 EN → H I PI R                     22 EN → H TF TF S
21 EN → H R EX                       21 EN → H S TF S EX
21 EN → H S EX TF EX                 20 EN → H I CG R EX
20 EN → H EX TF S
```

Figure 6.2: Sample dictionary productions with their frequences.

number of tokens / number of elements, for instance, the complexity of the content model `(a [b] | c*)` would be $9/3 = 3$. However, a trivial case (here `(a | b | c)*`) would always receive the lowest value.

6.2.1 Ideal case

First we consider a sample that produces a non-trivial but at the same time readable result. The input structures are the following:

```
P -> A C D E
P -> A D E
P -> A D E D E D E
P -> A C D E F G
P -> A H I
P -> A H I F G
P -> A H H H I F G
P -> M N R
P -> M N S
```

and the resulting element declaration is:

```
<!ELEMENT P - -    ( A, (C?, D, E, (D, E)* | (H)*, I), (F, G )?  |
                     M, N, (R | S) )                               >
```

Why is this sample so well-behaving? First of all, there are similar sequences of length 2 in the structures, so that the generalization process can combine them. If this would not be the case, the result would nonetheless be readable but it would also be quite trivial (See Section 6.2.4 below). Second, the right-hand sides of the sample productions can be divided to two groups that do not have much in common, namely the right-hand sides starting with "A" and the right-hand sides starting with "M". If these groups would have some sequence of the length 2 in common, the method would try to combine the structures and the result would not probably be readable (See Section 6.2.2).

Third, there are structures `M N R` and `M N S`, which produce a model group

```
(M, N, (R | S)).
```

If we also would have a structure `M N R S`, the model group would be

```
(M, N, (R S? | S)).
```

If there are many such structures, the model groups tend to become complicated (See Section 6.2.3).

The sample in Figure 6.3 [Sol94] is another example of a reasonably well-behaving input. The content model produced by our method is:

```
<!ELEMENT PERSON - -  ((STILLING)*, FNAVN, ENAVN, STATUS?,
                       ((TLF | BYGN | ROMNR)*,
                         (EMAIL | TREFFTID, EMAIL?
                          | FAX | INTEROMR)? )? >
```

The content model produced by the method in [Sol94] does not differ much from this:

```
<!ELEMENT PERSON  - - (STILLING*, FNAVN, ENAVN, STATUS?,
                       (TLF | BYGN | ROMNR)*,
                          (INTEROMR | FAX | TREFFTID?)?, EMAIL?) >
```

```
PERSON -> STILLING FNAVN ENAVN ROMNR TLF EMAIL
PERSON -> STILLING FNAVN ENAVN ROMNR TLF TREFFTID EMAIL
PERSON -> STILLING FNAVN ENAVN ROMNR TLF TLF EMAIL
PERSON -> STILLING FNAVN ENAVN BYGN TLF EMAIL
PERSON -> STILLING FNAVN ENAVN ROMNR TLF TLF TREFFTID EMAIL
PERSON -> FNAVN ENAVN TLF
PERSON -> STILLING FNAVN ENAVN TLF EMAIL
PERSON -> STILLING FNAVN ENAVN STATUS ROMNR TLF EMAIL
PERSON -> STILLING STILLING FNAVN ENAVN TLF
PERSON -> STILLING STILLING FNAVN ENAVN TLF INTEROMR
PERSON -> STILLING FNAVN ENAVN TLF INTEROMR
PERSON -> STILLING FNAVN ENAVN TLF
PERSON -> STILLING STILLING FNAVN ENAVN TLF TLF
PERSON -> STILLING FNAVN ENAVN TLF TLF
PERSON -> STILLING STILLING FNAVN ENAVN ROMNR BYGN TLF EMAIL
PERSON -> STILLING FNAVN ENAVN ROMNR BYGN TLF EMAIL
PERSON -> STILLING FNAVN ENAVN ROMNR BYGN TLF TLF
PERSON -> STILLING FNAVN ENAVN STATUS
PERSON -> STILLING FNAVN ENAVN ROMNR BYGN TLF TREFFTID
PERSON -> STILLING FNAVN ENAVN BYGN TLF
PERSON -> STILLING FNAVN ENAVN ROMNR BYGN TLF
PERSON -> STILLING FNAVN ENAVN ROMNR TLF
PERSON -> STILLING FNAVN ENAVN ROMNR TREFFTID
PERSON -> STILLING FNAVN ENAVN ROMNR TLF ROMNR BYGN TLF
PERSON -> STILLING FNAVN ENAVN STATUS ROMNR BYGN TLF TREFFTID
PERSON -> STILLING FNAVN ENAVN BYGN TLF ROMNR BYGN TLF
PERSON -> STILLING FNAVN ENAVN STATUS ROMNR BYGN TLF
PERSON -> STILLING FNAVN ENAVN ROMNR TLF FAX
PERSON -> STILLING FNAVN ENAVN ROMNR TLF TREFFTID
```

Figure 6.3: A well-behaving sample.

6.2.2 Different structures interfere

Consider the following modification of the previous sample, i.e., the production `P -> M N R` has been replaced by `P -> M N A D R`:

```
P -> A C D E
P -> A D E
P -> A D E D E D E
P -> A C D E F G
P -> A H I
P -> A H I F G
P -> A H H H I F G
```

```
P -> M N A D R
P -> M N S
```

Now the method tries to combine the two apparently distinguished structures, which makes the resulting content model quite complex:

```
<!ELEMENT P - -    ( A, (C?, D, ( ( D, E? | E ), (D, E?)*,
                     ( R | F, G)? )? | (H)*, I, (F, G)? )
                      | M, N, (A, (C?, D, ((D, E? | E )
                      (D, E?)*, (R | F, G)?)? |
                      (H)*, I, (F, G)?) | S )>
```

This phenomenon could be avoided by preprocessing the given document structures to find dissimilar structures, e.g., by comparing the sets of elements. Obviously different structures could then be processed separately.

6.2.3 Alternating sequences

Consider the following sample. Note that there are eight elements that appear always in the same order, but all the elements may also be missing:

```
P -> A B C D E F G H
P -> A C D E G
P -> B C E G H
P -> A E G H
P -> B D E F
P -> C E F
P -> D E F G H
P -> E F H
P -> F G H
P -> G H
P -> A G
P -> B C
P -> D E H
```

The content model that would gather the essential of the structure would be:

```
<!ELEMENT P - - (A?, B?, C?, D?, E?, F?, G?, H?) >
```

As the method learns from a positive sample, it never obtains any example of the situation when the content is empty. Hence, it does not produce the content model above but, instead, tries to form all the alternatives:

```
<!ELEMENT P - -   (A, (B, (C, (D, E, (F, (G, H? | H)?  | G, H? | H)
                    | E, (G, H? | F, (G, H? | H)? | H) )?
                    | D, E, (F , (G, H? | H)?  | G, H? | H) )
                    | C, (D, E, (F, (G, H? | H)?
                    | G, H? | H)  | E, (G, H? | F, (G, H? | H)?
                    | H ) )?
                    | E,  (G, H? | F, (G, H? | H)? | H) | G )
                    | B, (C, (D, E, (F,  (G, H? | H)? | G, H? | H)
                    | E, (G, H? | F, (G, H? | H)? | H) )?
                    | D, E, (F, (G, H? | H)? | G, H? | H ) )
                    | C, (E, (G, H? | F, (G, H? | H)? | H)
                    | D, E, (F, (G, H? | H)? | G, H? | H) )?
                    | D, E, (F, (G, H? | H)? | G, H? | H)
                    | E, (F, (G, H? | H)? | G, H? | H)
                    | F, (G, H? | H)? | G, H? )   >
```

Local trivialization cannot help in this case, since trivialization is based on generalizing orbits. Here, all the orbits are trivial. If, on the other hand, we have iterating sequences, as in the following, local trivialization can be used:

```
MR  -> KA EL
MR  -> KA EL MV
MR  -> KA SEL
MR  -> KA SEL EL
MR  -> MN EL
MR  -> MN KA EL
MR  -> MN KA EL MR
MR  -> MN KA KA SEL EL
MR  -> MN KA MN KA EL
MR  -> MN KA SEL EL
MR  -> MN KA SEL EL MR
MR  -> MN MR
MR  -> MN SEL
MR  -> MN SEL EL
MR  -> MN SEL EL MN SEL EL
MR  -> MN SEL EL MR
MR  -> MN SEL EL MV
MR  -> MN SEL MR
MR  -> MN SUO EL
```

The resulting declaration without local trivialization is:

```
<!ELEMENT MR - -  ((KA)*, (SEL, (EL, MN?)? | MN | EL, MN? ),
                    (KA, (KA)*, (SEL, (EL, MN?)? | MN | EL, MN?)
                     | SEL, (EL, MN?)? )*,
                     (EL | MR | SUO EL | MV)? ) >
```

and after local trivialization:

```
<!ELEMENT MR - - ((MN | SEL | EL | KA)*,  (MR | SUO, EL | MV)? ) >
```

Also in this case, the following variation would gather more of the underlying structure:

```
<!ELEMENT MR - - ((MN?, KA?, SEL?, EL?)*, (MV | MR | SUO, EL)? ) >
```

Note, that the languages `(MN | SEL | EL | KA)*` and `(MN?, KA?, SEL?, EL?)*` are the same.

The problem with alternating sequences originates from the fact that the method does not utilize any order information. The partial (or total) order of the elements would be easy to compute, and at least if the elements always appear in the same order, this knowledge should be represented in the content model. Additionally, if almost any element can be missing, it may be necessary to allow the generalization

```
rs | r | s -> r? s?
```

to avoid all the combinations presented.

6.2.4 Dissimilar input structures

As mentioned above, if the input productions do not contain common subsequences of length 2, the method does not find anything to combine, although the productions would share the same elements. For instance, in the following sample the sequences `C E` and `E G` are the only sequences that occur twice in the sample:

```
P -> A B C D E F G H
P -> A C E G
P -> B D F H
P -> A D G
P -> B E G
P -> C E H
P -> D H
P -> A E
P -> A F
P -> B F
P -> A G
P -> B H
P -> A H
```

As we can see from the following content model, the generalizations are also restricted to these occurrences.

```
<!ELEMENT P - -  (A, (B, C, D, E, F, G, H | C, E, (G | H)
                  | D, G | E | F | G | H )
                  | B, (D, F, H | E, (G | H) | F | H )
                  | C, E, ( G | H ) | D, H )  >
```

A real example of this feature can be found in [Sol94]:

```
UNFELLES -> AVSN
UNFELLES -> NAVN
UNFELLES -> NAVN NAVN START
UNFELLES -> NAVN VARIGHET
UNFELLES -> NAVN NAVN NAVN START VARIGHET
UNFELLES -> NAVN START
UNFELLES -> START
UNFELLES -> AVSN STED NAVN
UNFELLES -> AVSN NAVN
UNFELLES -> NAVN TID DAG STED VARIGHET AVSN
UNFELLES -> TID STED NAVN
UNFELLES -> STED NAVN
```

The content model produced is:

```
<!ELEMENT UNFELLES - -  ( (AVSN, (STED?, NAVN)? | (NAVN)*,
                           (START, VARIGHET? | VARIGHET
                           | TID, DAG, STED, VARIGHET, AVSN)?
                           | START | TID, STED, NAVN
                           | STED, NAVN)? ) >
```

and, again, the corresponding content model of [Sol94] is:

```
<!ELEMENT UNFELLES  - - (AVSN?, TID?, STED?, NAVN*, (START? | TID?),
                         DAG?, STED?, VARIGHET?, AVSN?) >
```

Chapter 7

Conclusion

In this study we have presented a method for generating a context-free grammar for a set of documents from the examples of document structures. Specifically, our method can generate SGML document type definitions from tagged instances of documents.

We have discussed the possible uses of automatic DTD generation tools, and which characteristics are needed from such tools in the applications. We introduced seven dimensions that affect the process of DTD generation: complexity of the structure, the time frame, completeness, readability, richness of structure, the possibility to modify the tagging, and the possibility to utilize interactive processing. Every application sets its own values for each of these dimensions, and thus determines how the DTD generation can proceed and which kind of quality can be expected from the result.

Our DTD generating method consists of the following steps:

1. Form simple productions that describe the input structures.

2. Build a prefix-tree automaton for every element which appears on the left-hand side of some production.

3. Generalize the automata by merging states.

4. Disambiguate the automata.

5. Construct the unambiguous regular expressions which form the right-hand sides of the productions.

In the generalization of the examples we have first applied the idea of k-contextual languages and further developed them to (k, h)-contextual languages. These conditions seem to describe quite natural constraints in text structures.

The resulting grammar may be further processed with both interactive and batch operations. These refining operations include local trivialization of a model group, isolation of model groups, discovery of inclusions, and separation of common and rare cases.

We have implemented the described method in C++, and experimented with several document types. Most challenging of these experiments has been generating a grammar for a part of a Finnish dictionary. The experiments show that if documents are rather simple, the automatic generalization gives sufficient results. However, refining operations are necessary, if the structures are more complicated. There are still some problematic cases that are not properly covered by the current operations, but could be handled by developing the method slightly.

The primary conclusion of this study is that automatic DTD generation is both necessary and possible. Obviously, the importance of it is even increasing, because of the advanced demands and dynamic document management needs of various new applications.

References

[AFQ89a] J. André, R. Furuta, and V. Quint. By way of an introduction. Structured documents: What and why? In J. André, R. Furuta, and V. Quint, editors, *Structured Documents*, The Cambridge Series on Electronic Publishing, pages 1–6. Cambridge University Press, 1989.

[AFQ89b] J. André, R. Furuta, and V. Quint, editors. *Structured Documents.* The Cambridge Series on Electronic Publishing. Cambridge University Press, 1989.

[AHH+96] H. Ahonen, B. Heikkinen, O. Heinonen, J. Jaakkola, P. Kilpeläinen, G. Lindén, and H. Mannila. Intelligent assembly of structured documents. Report C–1996–40, Department of Computer Science, University of Helsinki, 1996.

[Ang82] D. Angluin. Inference of reversible languages. *Journal of the ACM*, 29(3):741–765, 1982.

[Ang87] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[AS83] D. Angluin and C. H. Smith. Inductive inference: Theory and methods. *Computing Surveys*, 15(3):237–269, 1983.

[Bar89] D. Barron. Why use SGML? *Electronic Publishing*, 2(1):3–24, 1989.

[BBKF96] A. Brown, A. Brüggemann-Klein, and A. Feng, editors. *Proceedings of the International Conference on Electronic Publishing, Document Manipulation & Typography, Palo Alto, USA, September 24–26.* Wiley Publishers, 1996.

[BF72] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers*, 21(6):592–597, 1972.

[BKW92] A. Brüggemann-Klein and D. Wood. Deterministic regular languages. In A. Finkel and M. Jantzen, editors, *STACS '92, Proceedings of the 9th Annual Symposium on Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science 577, pages 173–184. Springer–Verlag, 1992.

[BKW94] A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. Technical report, Institut für Informatik, Universität Freiburg, May 1994. Accessible at URL: http://www.informatik.uni-freiburg.de/Personalia/Brueggemann-Klein.html.

[BR84] F. Bancilhon and P. Richard. Managing texts and facts in a mixed data base environment. In G. Gardarin and E. Gelenbe, editors, *New Applications of Data Bases*, pages 87–107. Academic Press, 1984.

[Bro89] H. Brown. Standards for structured documents. *The Computer Journal*, 32(6):505–514, 1989.

[Che91] J. Chen. Grammar generation and query processing for text databases. Research proposal, University of Waterloo, January 1991.

[CIV86] G. Coray, R. Ingold, and C. Vanoirbeek. Formatting structured documents: Batch versus interactive. In J. C. van Vliet, editor, *Text Processing and Document Manipulation*, pages 154–170. Cambridge University Press, 1986.

[FGHR69] J. A. Feldman, J. Gips, J. J. Horning, and S. Reder. Grammatical complexity and inference. Report TR CS 125/1969, Stanford University, 1969.

[FQA88] R. Furuta, V. Quint, and J. André. Interactively editing structured documents. *Electronic Publishing*, 1(1):19–44, 1988.

[FX94] P. Fankhauser and Y. Xu. Markitup! An incremental approach to document structure recognition. *Electronic Publishing – Origination, Dissemination and Design*, 6(4):447–456, 1994.

[GH67] S. Ginsburg and M. A. Harrison. Bracketed context-free languages. *Journal of Computer and System Sciences*, 1(1):1–23, 1967.

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

[Gol90a] C. F. Goldfarb. *The SGML Handbook*. Oxford University Press, 1990.

[Gol90b] C. F. Goldfarb. *The SGML Handbook*. Oxford University Press, 1990.

[GT87] G. Gonnet and F. Tompa. Mind your grammar: A new approach to modelling text. In P. Hammersley, editor, *VLDB '87, Proceedings of the Conference on Very Large Data Bases*, pages 339–346. Morgan Kaufmann Publishers, 1987.

[GV90] P. Garcia and E. Vidal. Inference of k-testable languages in the strict sense and application to syntactic pattern recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(9):920–925, 1990.

[GVC87] P. Garcia, E. Vidal, and F. Casacuberta. Local languages, the successor method, and a step towards a general methodology for the inference of regular grammars. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 9(1):841–845, 1987.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, Reading, MA, 1979.

[ISO94] ISO. *Information and documentation – Electronic manuscript preparation and markup, ISO 12083*, 1994.

[Jol89] V. Joloboff. Document representation: Concepts and standards. In J. André, R. Furuta, and V. Quint, editors, *Structured Documents*, The Cambridge Series on Electronic Publishing, pages 75–105. Cambridge University Press, 1989.

[Knu93] T. Knuutila. Inference of k-testable tree languages. In H. Bunke, editor, *Proceedings of the International Workshop on Structural and Syntactic Pattern Recognition*, pages 109–120, 1993.

[KS88] M. Kudo and M. Shimbo. Efficient regular grammatical inference techniques by the use of partial similarities and their logical relationships. *Pattern Recognition*, 21(4):401–409, 1988.

[Lev82] B. Levine. The use of tree derivatives and a sample support parameter for inferring tree systems. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 4(1):25–34, 1982.

[MA96] E. Maler and J. E. Andaloussi. *Developing SGML DTDs — from text to model to markup.* Prentice Hall PTR, 1996.

[Mit77] T. M. Mitchell. Version spaces: a candidate elimination approach to rule learning. In *IJCAI '77, Proceedings of the Fifth International Conference on Artificial Intelligence*, pages 305–310, 1977.

[Moo56] E. F. Moore. Gedanken-experiments on sequential machines. In C. Shannon and J. McCarthy, editors, *Automata Studies*, number 34 in Annals of Mathematics Studies, pages 129–153, Princeton, NJ, 1956. Princeton University Press.

[MP71] R. McNaughton and S. Papert. *Counter-Free Automata.* The MIT Press, Cambridge, MA and London, England, 1971.

[Mug90] S. Muggleton. *Inductive Acquisition of Expert Knowledge.* Addison Wesley, Reading, MA, 1990.

[Nat91] B. K. Natarajan. *Machine Learning: A Theoretical Approach.* Morgan Kaufmann Publishers, San Mateo, CA, 1991.

[ODA89] Information Processing – Text and Office Systems – Office Document Architecture (ODA) and Interchange Format. Technical Report ISO/IEC 8613, International Organization for Standardization ISO/IEC, Geneva/New York, 1989.

[PC78] T.-W. Pao and J. W. Carr. A solution of the syntactical induction-inference problem for regular languages. *Computer languages*, 3(1):53–64, 1978.

[PR87] L. Pitt and R. E. Reinke. Polynomial-time solvability of clustering and conceptual clustering problems: The agglomerative-hierarchical algorithm. Technical Report UIUCDCS-R-87-1371, University of Illinois, Dept. of Computer Science, 1987.

[Qui89] V. Quint. Systems for the manipulation of structured documents. In J. André, R. Furuta, and V. Quint, editors, *Structured Documents*, The Cambridge Series on Electronic Publishing, pages 39–74. Cambridge University Press, 1989.

[QV86] V. Quint and I. Vatton. Grif: An interactive system for structured document manipulation. In J. C. van Vliet, editor, *Text Processing and Document Manipulation*, pages 200–213, Cambridge, 1986. Cambridge University Press.

[RV84] M. Richetin and F. Vernadat. Efficient regular grammatical inference for pattern recognition. *Pattern Recognition*, 17(2):245–250, 1984.

[Sak88] Y. Sakakibara. Learning context-free grammars from structural data in polynomial time. In D. Haussler and L. Pitt, editors, *Proceedings of the 1988 Workshop on Computational Learning Theory*, pages 330–344. Morgan Kaufmann Publishers, 1988.

[Sak92] Y. Sakakibara. Efficient learning of context-free grammars from positive structural examples. *Information and Computation*, 97(1):23–60, 1992.

[Sal69] A. Salomaa. *Theory of Automata.* Pergamon Press, 1969.

[SGM86] Information Processing – Text and Office Systems – Standard Generalized Markup Language (SGML). Technical Report ISO/IEC 8879, International Organization for Standardization ISO/IEC, Geneva/New York, 1986.

[Sha95] K. Shafer. Automatic DTD creation via the GB-Engine and Fred. Technical report, OCLC Online Computer Library Center, Inc., 6565 Frantz Road, Dublin, Ohio 43017-3395, 1995. Accessible at URL: http://www.oclc.org/fred/docs/papers/.

[Sol94] S. M. K. Solstrand. Automatisk generering av DTD fra SGML-kodet materiale. M.Sc.thesis, Institutt for informasjonsvitenskap, Universitetet i Bergen, September 1994.

[Suo90] *Suomen kielen perussanakirja. Ensimmäinen osa (A–K).* Valtion painatuskeskus, Helsinki, 1990.

[VB87] K. Vanlehn and W. Ball. A version space approach to learning context-free grammars. *Machine Learning*, 2(1):39–74, 1987.

[vH94] E. van Herwijnen. *Practical SGML, 2. ed.* Kluwer Academic Publishers, Boston, Dordrect, London, 1994.

[Woo87] D. Wood. *Theory of computation.* Harper & Row Publishers, New York, 1987.

ISSN 1238-8645
ISBN 951-45-7532-6
Helsinki 1996
Helsinki University Printing House

Helena Ahonen: Generating Grammars for Structured Documents Using Grammatical Inference Methods **A-1996-4**