Document transformation by tree transducers

Eila Kuikka, Paula Leinonen, and Prof. Martti Penttonen

Department of Computer Science and Applied Mathematics, University of Kuopio

P.O.Box 1627 FIN-70211 Kuopio, Finland

E-mail: {`kuikka`, `leinonen`, `penttonen`}`@cs.uku.fi`

Abstract

We report a syntax-directed approach to creation and transformation of structured documents. We assume that the documents to be handled have a syntactically definable structure. Whatever is done with the document, at creation, at later transformations or other reuses, it is done in accordance with the grammatical structure. In this work we focus at the transformation of documents from structure to another. We show that in an important case, called local transformations, the transformation can be performed by finite state tree transducers, and suggest a system supporting this kind of transformations.

Contents

1 Introduction

Usually, when we write a text for professional reasons, like a report, the text has some logical structure. We want to make the content of our text understandable for the reader. There are many ways to externally express the structure, like using the empty space, the size and type of the font etc. It is usual in word processing to "decorate" the text with these external methods, indirectly giving it some structure. In structured document processing, on the contrary, the structure comes first and external form is secondary. Structure and form may even be completely separated so that a document with a structure marking does not say anything about the form how the document should look like. Still it is possible to produce a beautifully typeset document starting from a structured document, even automatically, as soon as rules for associating form to structure are given. It is even possible that different styles of typesetting can be applied to get different forms of the same structured document. Even if these documents look different, they have the same contents and the same structure elements expressed in the original structured document. Structure markup brings some other advantages, too. Structure markups make possible to filtering out parts of the document by structure information, instead of mere linear string search. It also makes automatic changes in the structure possible, see [5], e.g. It is the aim of this article to study, how structured documents can be automatically transformed to other structure. Here we try to report in a strict and concise form the work presented in [8, 6, 7].

In syntax-directed approach to structured document processing (see Figure 1), the structure of the document is defined by a formal grammar, called document type definition in SGML and XML culture [4]. When the document is being written, the software forces, supports or checks that the document indeed follows the structure defined by the grammar.

When an existing document is transformed to another structure, it is assumed that the source document and the target document have much in common. One cannot transform an almanac to "Don Quijote". Most of the content elements are common and there is some similarity in the structure, even if some elements are missing in one of the documents and the order of structure elements may change. It is likely that some human interpretation is needed to tell the transformation system that "author" required by the source grammar and "writer" of the target grammar probably mean the same.

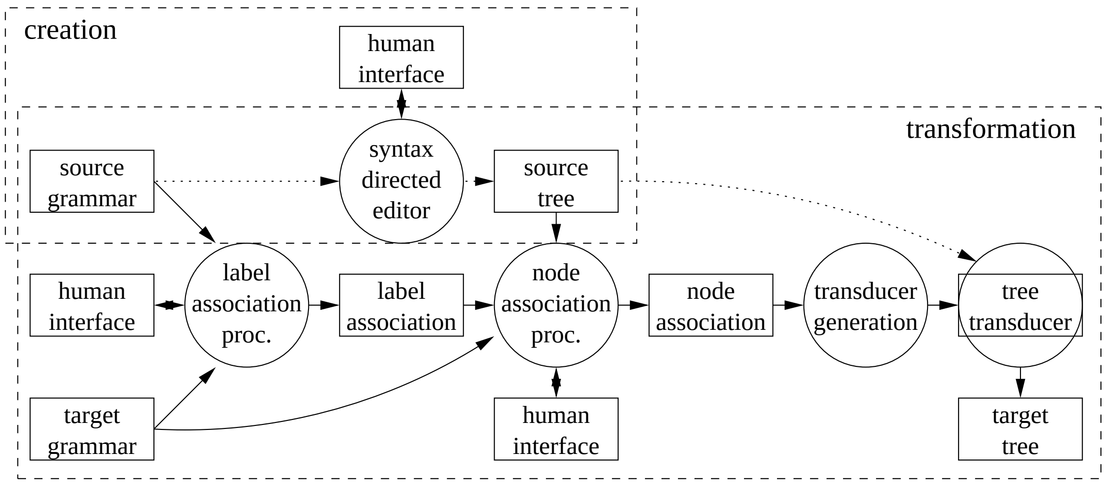

Figure 1: *Architecture of a document creation/transformation system*

A scheme of a document creation/transformation system is depicted in Figure 1.

2 Grammars and transducers

Instead of XML notation we prefer using the plain grammar notation here. In this section, two devices that form the basis of this work, are presented: the grammar that is used to define a structure and to create a document, and the tree transducer that is used to transform a document with a structure to a document with another structure.

2.1 Context–free grammars

A *context-free grammar* is a quadruple $G = (V, T, P, S)$, where V and T are finite sets called *nonterminals* and *terminals*, $S \in V$ is the *start symbol* and P is a finite set of *rules* of the form $A \to x$, $A \in V$, $x \in (V \cup T)^*$. In particular, the empty string ϵ is allowed as the right hand side of a rule.

A string $uAv \in (V \cup T)^*$ *derives directly* to uxv, denote $uAv \Rightarrow uxv$, if there is a rule $A \to x \in P$. A string u *derives* to a string v, denote $u \Rightarrow^* v$, if there are strings $u = u_0, u_1, \ldots, u_m$ such that $u_i - 1 \Rightarrow u_i$ for all i. In other words, $\Rightarrow^*$ is the reflexive, transitive closure of $\Rightarrow$. If $S \Rightarrow w$, w is called a sentential form of the grammar G. The *language generated* by G is the set $L(G) = \{w|S \Rightarrow^* w, w \in T^*\}$.

To each derivation $A \Rightarrow^* w$ of G, a *derivation tree* is associated as follows. A node labeled with A is the *root* of the derivation tree. If the whole derivation is $A \Rightarrow w_1 \Rightarrow \ldots \Rightarrow w_i \Rightarrow w_{i+1} \Rightarrow \ldots \Rightarrow w$ and we already have the derivation tree for $A \Rightarrow w_1 \Rightarrow \ldots \Rightarrow w_i$, it is extended for the derivation $A \Rightarrow w_1 \Rightarrow \ldots \Rightarrow w_i \Rightarrow w_{i+1}$ as follows. Let the direct derivation step be $w_i = uBv \Rightarrow uyv = w_{i+1}$, where $B \rightarrow y \in P$. Assume that $y = y_0y_1 \ldots y_j$, where $y_k \in V \cup T$ for all k. Then there are arcs from the node labeled with B to new nodes labeled with $y_0, \ldots, y_j$ and these nodes are called *children* of Y. We also use the term *parent* for the inverse of child, *descendant* for the reflexive, transitive closure of child, and *ancestor* for the reflexive, transitive closure of parent.

Example 1 Consider the grammar, whose start symbol is `article`, other nonterminals are `author`, `date`, `title`, `content`, `abstract`, `section`, `heading`, `paragraph`, `itemlist`, `textpara`, and `item`, the only terminal is `text`, and the rules are

```
article -> author+ [date] title content
author -> text
date -> text
title -> text
content -> abstract section+
abstract -> text
section -> heading paragraph+
heading -> text
paragraph -> textpara
paragraph -> itemlist
itemlist -> item+
textpara -> text
item -> text
```

In the rules some abbreviations is used. $B+$ means "any number of B's", i.e. $B+$ behaves as if it were a nonterminal and there were rules $B+ \rightarrow B\ B+$ and $B+ \rightarrow B$ in the grammar. $[B]$ means "zero or one B" and it has the same effect as rules $[B] \rightarrow \epsilon$ and $[B] \rightarrow B$. Figure 2 presents a derivation tree generated by this grammar. Actually, the output of the

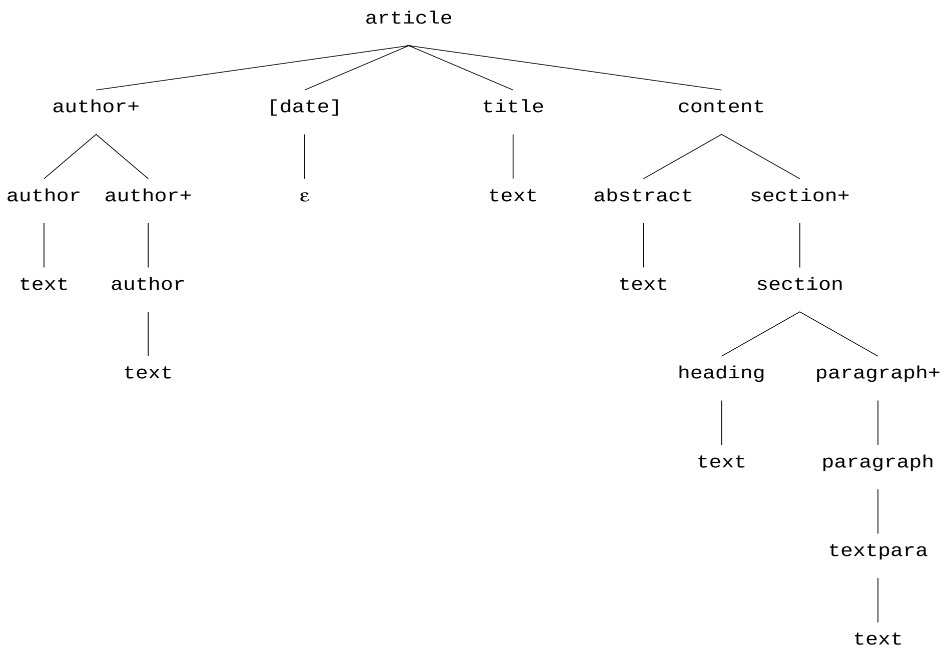

Figure 2: *A derivation tree generated by the grammar of example 1.*

grammar is `text text text text text text` read at the leaves of the derivation tree. Each `text` is a homogeneous content element of a document. In word processing it is customary that different content elements are laid out differently to express its role in the document, for example the `text` below `title` is typeset with a big font. In structured document processing, instead, we usually identify the document and its derivation tree. Hence, we would rather say that

```
article(
  author+(
    author(text),
    author(text)),
  [date](),
  title(text),
  content(
    abstract(text),
    section+(
      section(
        heading(text),
        paragraph(
        textpara(text))))
```

is the document, without any layout.

Grammar is a finite device that is capable of producing an infinite number of strings and derivation trees. If we consider a path from a leaf to the root of the derivation tree, if the length is greater than the size of the alphabet, a nonterminal A must appear twice on the path. Hence, the derivation must be of the form $S \Rightarrow^* uAv \Rightarrow^* uxAyv \Rightarrow^* uxwyv$. The subderivation $A \Rightarrow^* xAy$ is called a *pumping factor* and it can be applied any number of times to get $S \Rightarrow^* ux^iwy^iv$, $i \geq 0$. Notice that $i = 0$ corresponds the case when the pumping factor is not applied at all. Thus, a pumping factor can be eliminated.

Consider all possible derivation trees generated by a grammar G. If in all derivation trees all pumping factors are eliminated, we get a finite number of different trees (because the depth is bounded). We call these trees *elementary trees*. By reverse argument, all derivation trees can be constructed by adding pumping factors to these trees. Note also that we can choose pumping factors so that in paths from leaf to root the same nonterminal may occur only as the root node and the leaf node. Thus also the number of pumping factors is finite. We state these facts as

Lemma 1 *For any context-free grammar, all derivation trees can be represented as a composition of a finite set of elementary trees and a finite set of pumping factors.*

In [9] and [5] a system called SYNDOC was presented, which supports creating structured documents under control of a context-free grammar.

2.2 Tree transducers

One of the main motivations of the structure markup is making the reuse of the document possible. If the syntax of the document and content elements are clearly marked, there are many possible reuses. Here we restrict ourselves to article type documents and their reuses as articles with different structure. Therefore, we need a device for transforming trees. For our purposes, finite state tree transducer is a suitable device. For a more elaborate theory of tree transducers, consult [3].

In case of derivation trees, applying a rule $A \rightarrow x_1x_2 \ldots x_m$, where $x_i \in N \cup T$, corresponds to adding a node for each x_i and an arc from A to x_i. Algebraically, we see $V = N \cup T$ as a *ranked alphabet*, where each nonterminal $A \in N$ is an m-ary function symbol ($m \geq 1$), and terminals in T are 0-ary function symbols. Trees can be considered as *terms* that are defined recursively as follows: Let V be a set of function symbols, each having arity $n \geq 0$. (i) Each 0-ary symbol of V is a term, a *leaf.* (ii) If $t_1, \ldots, t_n$ are terms and $f \in V$ is an n-ary function symbol, then $f(t_1, \ldots, t_n)$ is a term. The set of terms so defined is denoted by $T(V)$. It is useful to extend the definition of term to allow variables. Let X be a set of variables. Extend now (i) to let each 0-ary symbol and each variable be a term. Then we get the set of terms with variables $T(V, X)$, where leaves are 0-ary functions (terminals) or variables. Thus, $T(V) = T(V, V^0)$ and $T(V, X) = T(V, X \cup V^0)$, where V^0 is the set of 0-ary symbols.

A *finite state tree transducer* $M = (Q, V_1, V_2, X, q_0, \delta)$ consists of a finite set Q of *states*, a ranked *input alphabet* V_1, a ranked *output alphabet* V_2, a set of variables X, the *initial* state $q_0 \in Q$, and the *transition function*

$$\delta : Q \times T(V_1, X) \to T(V_2, Q \times T(V_2, X)).$$

In other words, δ is a finite set of rules of the form

$$q : t(X_1, \ldots, X_m) \to t'(q_1 : X_{i_1}, \ldots, q_n : X_{i_n}).$$

where $X_1, \ldots, X_m$ are the variables of a term $t(X_1, \ldots, X_m)$, and $X_{i_j} \in \{X_1, \ldots, X_m\}$.

The transducer M induces a relation $Q \times T(V_1) \to^* T\big(V_2, Q \times T(V_1)\big)$ as follows.

If $t = t(t_1, \ldots, t_m)$, $t_i = \delta(q, t(X_1, \ldots, X_m)) \to t'(q_1 : X_{i_1}, \ldots, q_n : X_{i_n})$, and $q_j : t_{i_j} \to^* t'_{i_j}$, $t'_{i_j} \in T(V_2, Q \times T(V_1))$, then $q : t \to^* t'(t'_{i_1}, \ldots, t'_{i_n})$.

Finally, M transforms $t \in T(V_1)$ to $t' = t' \in T(V_2)$, if $q_0 : t \to^* t'$.

Example 2 Consider the tree transducer

```
q0:article(W,[date](D),title(text),content(A,S)) ->
   article(q1:W,title(q0:text),keywords(q0:text),q0:content(A,S))
q1:author+(author(text),W) -> writers(writer(q0:text),q1:W)
q1:author+(author(text)) -> author+(author(q0:text)))
q0:content(abstract(text),S) -> content(summary(q0:text),q2:S)
q2:section+(section(heading(text),P),S) ->
   section+(section(heading(q0:text),q3:P),q2:S)
q2:section+(section(heading(text),P) ->
   section+(section(heading(q0:text),q3:P))
```

```
q3:paragraph+(paragraph(X),P) -> paragraph+(paragraph(q4:X),q3:P)
q3:paragraph+(paragraph(X)) -> paragraph+(paragraph(q4:X))
q4:paragraph(textpara(text)) -> paragraph(textpara(q0:text))
q4:paragraph(itemlist(I))) -> paragraph(itemlist(q5:I))
q5:item+(item(text),I) -> item+(item(q0:text),q5:I)
q5:item+(item(text)) -> item+(item(q0:text))
q0:text -> text
```

After one step of the transducer, the derivation tree of Figure 2 becomes the tree in Figure 3. The final output of the transducer is given in Figure 4.

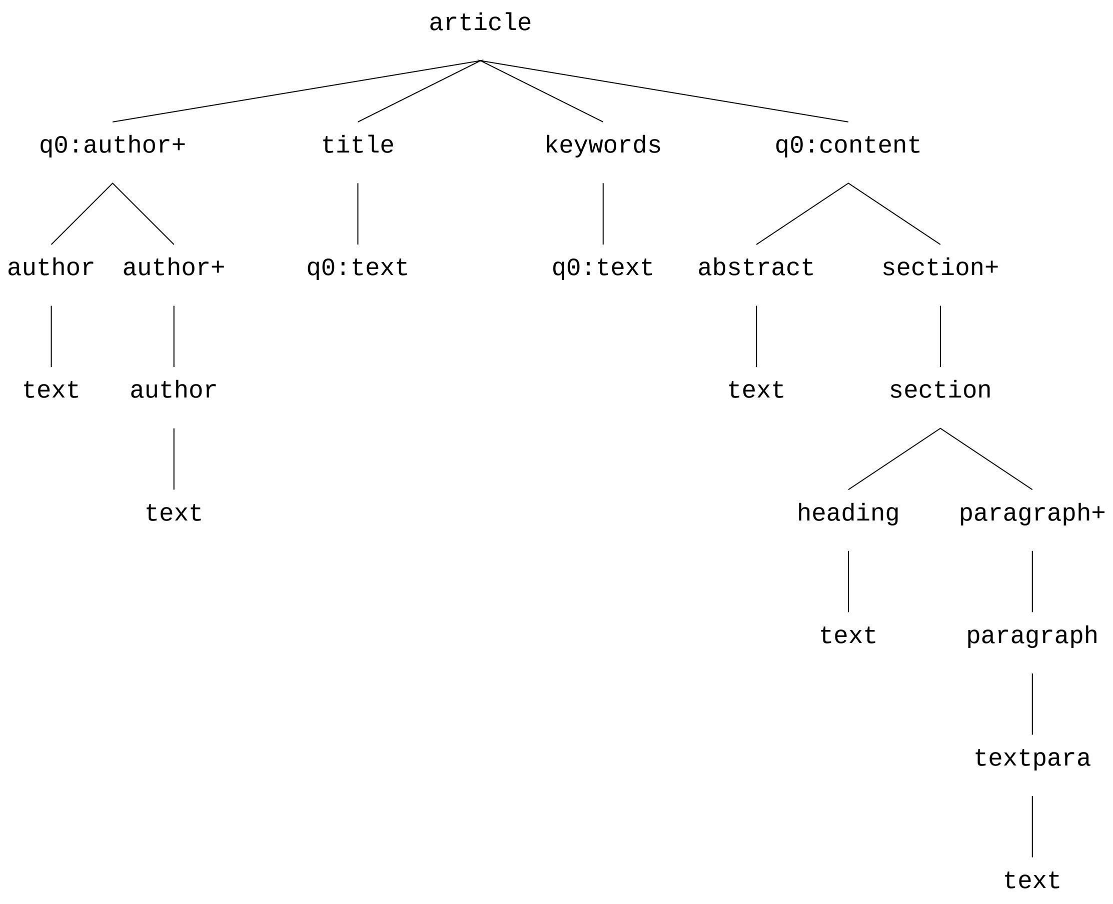

Figure 3: *Derivation tree after one transformation step.*

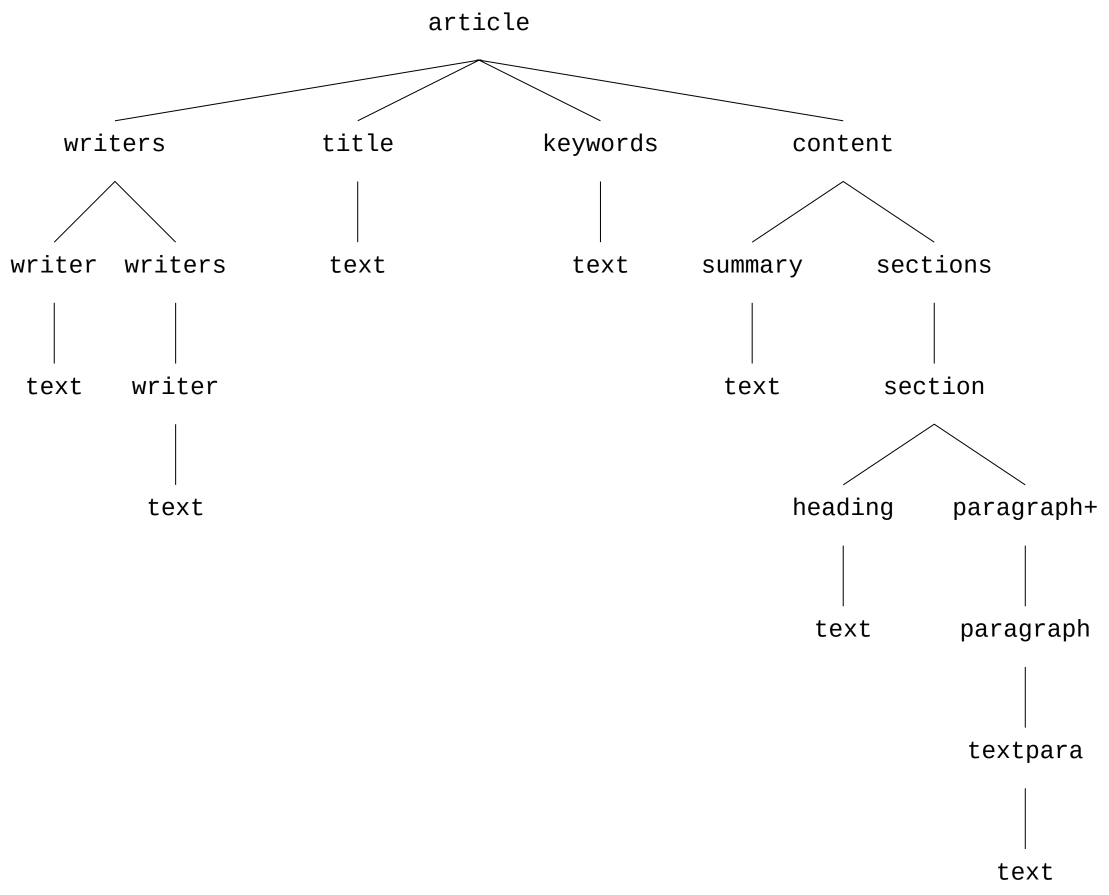

Figure 4: *Derivation tree after complete computation.*

3 Hierarchic, local, and dense transformations

For two context-free grammars G_1 and G_2, a *transformation* is a relation from the set of the derivation trees of G_1 into the set of the derivation trees of G_2. We require that transformation is defined for all derivation trees of G_1, but do not require that output of transformation is unique, or that all derivation trees of G_2 occur as a result of a transformation.

The reason for transforming a document is that there is some useful information stored somewhere in the document and there is a new need to use it for a similar or a different purpose, and for some reason it cannot be reused in identical form. A requirement for the successful reuse of a document is that the content is clearly enough marked up, so that content elements can be picked for reuse. If there is nothing in common, or the contents is not marked up, automatic transformation for reuse is impossible. If the structure is well marked and the new need is not very different, the transformation may be easy to automate. The purpose of this study is to characterize a class of transformations that are possible to automate, or at least to semi-automate.

To give a formal definition for the idea of "common", or "corresponding", structure elements we introduce the concept of label association. For any alphabets V_1 and V_2, *label association* is a relation in $\lambda \subseteq V_1 \times V_2$. In a special case, label association may be a function, but it needs not be. For example, labels of the tree in Figure 2 can be associated with the labels of the tree in Figure 4 by mapping `author+` to `writers`, `author` to `writer`, `abstract` to `summary`, and other labels to itself. However, we do not want to associate `[date]` in V_1 and `keywords` in V_2 because they do not have the same semantic meaning. It is better to leave them unassociated, because they do not have a counterpart in the other document.

The difficulty of the transformation depends on how the "corresponding" elements are situated in the transformed document tree. To speak about that, we introduce the concept of node association. For trees $t_1 \in T(V_1)$, $t_2 \in T(V_2)$, we call a *node association* a relation ν from the nodes of t_1 to the nodes of t_2. The node association ν from t_1 to t_2 *respects* a label association λ from V_1 to V_2, if the following three conditions are fulfilled. (i) If $(n_1, n_2) \in \nu$ and the labels of the nodes are $X \in V_1$ and $Y \in V_2$, then, $(X, Y) \in \lambda$. (ii) If a node n_1 of t_1 has a label X such that $(X, Y) \in \lambda$ for some $Y \in V_2$, then there is node n_2 in t_2 such that $(n_1, n_2) \in \nu$. (iii) If a node n_2 of t_2 has a label Y such that $(X, Y) \in \lambda$ for some $X \in V_1$, then there is node n_1 in t_1 such that $(n_1, n_2) \in \nu$.

A tree transformation τ is *hierarchic* with respect to a label association λ, if for any $t_2 \in \tau(t_1)$, there is a node association ν respecting λ such that whenever $(x, u) \in \nu$, $(y, v) \in \nu$, y being a descendant of x implies v being a descendant of u.

The tree transformation τ is $(c, d)-$*local* for constants c and d and label association λ, if there is a node association ν such that whenever the distance of nodes x and y in t_1 being less than c and $(x, u) \in \nu$, $(y, v) \in \nu$, the distance of u and v in t_2 is less than d.

Finally, τ is *e-dense* with respect to constant e and label association λ, if whenever a node has label X associated in λ (i.e. having Y such that $(X, Y) \in \lambda$), it has a descendant associated in λ within distance e, or it does not have any associated descendants.

Intuitively, these concepts are important for the following reasons. Most of the documents have an hierarchically organized content and it is not easy to imagine a case, where a part becomes a collection and vice versa. From the technical point of view, a hierarchic transformation can be processed by progressing from root to leaves (or alternatively from leaves to root). Being local means that transformation rule can be decided by a bounded lookahead in the document. Denseness means that transformable content elements are so closely situated in the document that the next transformable part can be found by a finite lookahead.

Theorem 1 *Let $G_1 = (N_1, T_1, P_1, S_1)$ and $G_2 = (N_2, T_2, P_2, S_2)$ be context-free grammars and λ be a label association from $V_1 = N_1 \cup T_1$ to $V_2 = N_2 \cup T_2$. If τ is a transformation from $T(V_1)$ to $T(V_2)$ that respects λ and is hierarchic, (c, d)-local and e-dense for some constants c, d, e, then there is a finite tree transducer M such that $M(t_1) = \tau(t_1)$ for all derivation trees of t_1.*

Proof. Let τ be a hierarchic, (c, d)-local and e-dense transformation. Consider a derivation tree t_1 and $t_2 \in \tau(t_1)$. By Algorithm 1 one can construct rules of a tree transducer producing t_2 from t_1.

By Lemma 1 all derivation trees of grammar G_1 can be be presented as a composition of a finite set of trees. By applying Algorithm 1 to all of these, we get pairs of subtrees to be used as rules to transform all derivation trees generated by G_1. □

Algorithm 1 *Construction of tree transducer rules*

Input. *A derivation tree t_1 by grammar G_1, a derivation tree t_2 by grammar G_2 such that $t_2 = \tau(t_1)$, a label association λ between G_1 and G_2, and a $(c, d)-$local, $e-$dense node association respecting λ.*

Output. *Tree transducer rules implementing τ.*

1. *Associate the roots of t_1 and t_2. and mark them as* open *nodes.*

2. *If there are no more open nodes in t_1, stop. Otherwise choose an open node n_1 and its associated node n_2 in t_2.*

3. *Expand the subtree whose root is n_1 until in each branch a node m_1 with a nonempty association m_2 or a node without associated descendants in its subtree is found. (Due to denseness assumption, these nodes are found in the bounded depth.) The nodes in frontier having a nonempty association are marked open, all other generated nodes and n are marked closed.*

4. *Expand the node n_2 in t_2 so that the open nodes generated in 3 can be associated with nodes in t_2. (Such nodes can be generated in finite time, because there is a hierarchic and local association.)*

5. *Go to 2.*

4 Document transformation system

In last section, we proved that hierarchic, local and dense transformations can be implemented by finite state tree transducers, and described how transducer rules are constructed from the pairs of source tree and target tree. However, it is more likely that a target tree is not available, and even if it is, describing the node association in the whole tree might be cumbersome. Therefore, in this section we discuss, at a less formal level, how the transformation is can be developed by an interactive procedure.

4.1 Setup of the transformation task

Assume a situation, where there is given a structured document tree t_1, a grammar G_1 by which t_1 was generated, and a grammar G_2 that defines the target structure to which t_1 should be transformed. The output of the transformation t_2, that is a derivation tree by G_2, and corresponds to t_1 by contents.

4.2 Finding the label association

The transformation procedure starts with a comparison of grammars, and one should find the corresponding content elements. As a result of this comparison one gets a label association λ, as defined in last section.

It may, however, be difficult to get the label association right at once. Therefore, it would be good to make the transformation system so flexible that one can correct or complete the label association and save as much of the later work as possible.

4.3 Finding node associations

The most essential part of the transformation is finding out, what subderivations by grammar G_2 correspond to subderivations of the document tree derived by G_1. We know that start symbols must correspond to each others.

Assume that the matching procedure has advanced to a subtree t'_1 of t_1 and a leaf node n_1 of t is labeled with nonterminal A_1, see Figure 5. By this time a part of the target tree has already been constructed, call it t'_2. By earlier transformation (or by the assumption that start nonterminals are related in λ) we know its matching nonterminal node n_2 labeled with nonterminal A_2, which is a leave in t'_2. Now t'_1 is expanded at A_1 within t_1 by rules of G_1 so much that each leave either has an associated label in λ (like $(B_1, B_2) \in \lambda$ in Figure 5), or this leave has no descendant in t_1 that is associated with any label in λ. By denseness assumption this expansion t''_1 can be found in constant time. Now we should expand t'_2 respectively. We should find a derivation by G_2 that starts at nonterminal A_2 and should have associated leaves, like B_2 in Figure 5. If we know that a hierarchic and local transformation respecting λ exists, the expansion t''_2 can be found by a finite search, and $t''_1 \to t''_2$ defines the transformation step. In this way rules for the transformation of the whole tree are found.

In the above reasoning there may be several different t''_2 that fulfill the association requirement. Which one is correct? The user should say.

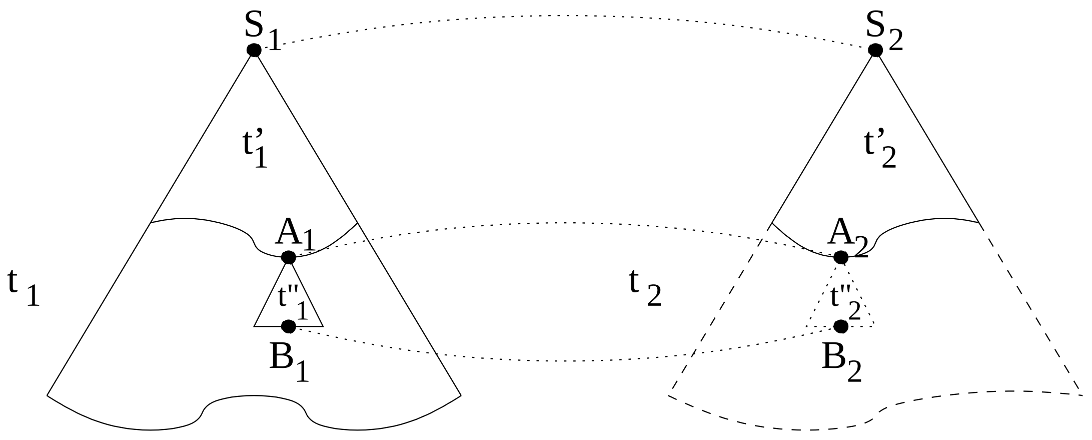

Figure 5: *Expansion of the transformation*

The system should offer alternatives generated by G_2 and the user accepts one of them. To speedup the finite search and selection, the system can use some optimality criteria, like (i) choose the smallest possible subtree, (ii) choose the subtree that keeps the order, etc. Sometimes it may be useful to allow transformations that do not completely match the label association criteria—in that case we could use the selection criterion (iii) choose the subtree with as many matching labels as possible.

4.4 Lazy vs eager transformation system

When tree transducer rules that are needed for transforming a single document tree are constructed, it is probable that they are not sufficient for transforming another document tree. There are strategies to construct "full" finite state tree transducer, that can perform all transformations from grammar G_1 to grammar G_2 respecting a label association λ. By *eager* strategy one first (automatically) generates all elementary trees and pumping factors, as stated in Lemma 1, an then constructs transformation rules for all of these. By Lemma 1 all trees can now be processed.

However, if the grammar G_1 is big, the number of elementary trees and pumping factors may be high, and some of them may occur seldom. It is probable that most of the documents are quite similar in structure. Therefore, the *lazy* strategy may be more reasonable. First construct rules for one document, and use these rules as long as they are sufficient. When a new document cannot be transformed with existing rules, the system falls to the interactive mode and new rules are constructed. This approach can also be called *teaching by example.*

4.5 Experts and novices

As soon as the tree transducer is completely constructed, using it should be as simple for novice users as using any application program, by a suitable user interface.

In the construction phase, however, some expertise is required. The expert user should understand structure definition by grammars. He/she should also identify, which structure elements (or nonterminals) in two grammars refer to identical contents. The system should also allow easy changing of associations, because it is not easy to make the right associations at the first try.

4.6 Implementation of the transformation system

We have implemented a very rudimentary prototype of the transformation system. The system can automatically perform local transformations in small documents. It implements in some way all essential phases from label association to rule construction and application. However, our prototype is not sufficient for real use, because it is inefficient (written in Prolog and Tcl/Tk, without optimizations), and user interface is too difficult except for developers. Some more description about the system is found in [7].

5 Discussion

There is a need for structured document transformations. As the range of applications, where structured documents are used, there certainly remains cases, where one just has to use an "ad hoc" method, but when possible, a suitable methodology should be used.

We characterized a class of transformations, hierarchic, local and dense transformations, which can be automated by finite state tree transducers. We believe that these transformations cover a large part of structured document transformations. We also believe that finite state tree transducers are a good model for this task because of their cleanness, simplicity, and sufficient power. Structure transformation is basically replacing tree structures by others. Top-down progress and state control give rigidity to this replacement process. For other approaches, see [1] or [2].

In our model, the transformation is defined by a dialogue between the transformation system and an expert user. The first phase in the procedure is the definition of the label association, the correspondence of the nonterminals in the two grammars (or document type definitions). The second phase is the construction of the transformation rules, where the system suggests rules and the users accepts. In this way it is guaranteed that the result of the transformation obeys the target structure.

The transformation system must guarantee that the automatic transformation indeed transforms from the given source structure to the given target structure. It should also give the user intuitive support so that he/she can become convinced that will be semantically correct, and if not, an easy way to correct the rule. Our first prototype of transformation system can be called a transformation system, but is still far from being useful.

References

[1] E. Akpotsui, V. Quint, *Type transformations in structured editing systems.* In: C. Vanoirbeek, G. Coray (eds.): Proceedings of Electronic Publishing, Cambridge University Press, 1992, pp. 27–41.

[2] R. Furuta and P. D. Stotts, *Specifying structured document transformations.* In J.C. van Vliet, editor, Document Manipulation and Typography, The Cambridge Series on Electronic Publishing, pp. 109–120, Nice, France, 1988. Cambridge University Press.

[3] F. Gécseg and M. Steinby, *Tree Automata.* Académiai Kiadó, Budabest, 1984.

[4] ISO 8879, *Information processing—Text and Office Systems—Standard Generalized Markup Language (SGML).* ISO, Geneva, 1986.

[5] E. Kuikka, *Processing of Structured Documents Using a Syntax-Directed Approach.* PhD thesis, Kuopio University Publications C. Natural and Environmental Sciences 53, 1996.

[6] E. Kuikka, P. Leinonen, M. Penttonen, *An approach to document structure transformations.* In: Yulin Feng, David Notkin, Marie-Claude Gaudel (eds): Proceedings of Conference on Software: Theory and Practice, pp. 906–913, 16'th IFIP World Computer Congress 2000, Beijing, China.

[7] Eila Kuikka, Paula Leinonen, Martti Penttonen, *Overview of tree transducer based document transformation system.* In: Ethan V. Munson, Derick Wood (eds): Preliminary Proceedings of the Fifth International Workshop on Principles of Digital Document Processing (PODDP'00), 13 p., Munich, Germany, 2000.

[8] E. Kuikka and M. Penttonen, *Transformation of structured documents.* Electronic Publishing—Origination, Dissemination, and Design. 8(4):319–341, December 1995.

[9] E. Kuikka, M. Penttonen, and M.-K. Väisänen, *Theory and implementation of SYNDOC document processing system.* Proceedings of the Second International Conference on Practical Application of Prolog (PAP-94). London, UK, 1994, pp. 311–327.