HKUST Theoretical Computer Science Center Research Report HKUST-TCSC-2001-06

The Visual Specification of Context*

Anne Brüggemann-Klein† Stefan Hermann‡ Derick Wood§

April 14, 2001

Abstract

We introduce a new visual technique for the specification of context in hierarchically structured documents such as those defined by XHTML and XML. The technique is based on a formal graph model called T-graphs and on what we call T-configurations in (abstract) document trees. A T-configuration is a restricted substructure of a tree. The technique is implemented in the current version of DESIGNER. Although we apply this technique to the specification of context-dependent stylesheets for XHTML and XML documents, the technique has much wider applicability. For example, it can be applied to query specification for structured documents and to computer program transformations.

Previously, we introduced a context-specification method that used regular expressions to specify sets of paths in a tree that we called caterpillar expressions. The work on T-graphs is an attempt to provide a 90 percent solution to context-specification problems that solves, in practice, almost all context-specification problems. T-graphs are much easier to visualize and to construct than are caterpillar expressions; they are both a restriction of and a generalization of caterpillar expressions.

We compare T-graphs with the context specification techniques found in other stylesheet systems and we also provide examples of context that we can and cannot specify with T-graphs. Although T-graphs are restrictive, they lend themselves to visual construction and modification, our main requirement when we designed this context-specification method. We also investigate the time and space complexity of T-graph matching, a necessity for efficient implementation.

1 Introduction

Context-dependent processing and specification are not new topics; they surface in almost all computing activities. What is somewhat surprising is that the issue of context does not appear to have been studied *as a topic in its own right.* (It has, however, been a topic of much interest in formal language theory; in particular, for variants of grammars.) In the DESIGNER project, that we have been working on over a number of years, we were faced with the problem of the specification of context-dependent style rules. At first, we expected to use the traditional approach from the compiler and programming-languages community; namely, attribution [3, 22, 30]. In addition, we also expected to be able to use and modify previous approaches to style specification such as those

*The work of the three authors was supported partially by a joint DAAD-HK grant. In addition, the work of the third author was supported under a grant from the Research Grants Council of Hong Kong. Preliminary versions of this paper were presented at DL '99 [10] and at ICCC/EP '99 [11].

†Institut für Informatik, Technische Universität München, Arcisstr. 21, 80290 München, Germany. {brueggem,hermann}@informatik.tu-muenchen.de

§Department of Computer Science, Hong Kong University of Science & Technology, Clear Water Bay, Kowloon, Hong Kong SAR. dwood@ust.hk

suggested by the DSSSL standard [20], the XSL document [1], and the W3C work on cascading stylesheets (CSS) for both XHTML and XML documents [4] and Lie and Bos's text on the use of CSS for XHTML [23]. We also considered Munson's approach in the Proteus system [26] and Murata's more general approach [27] which is based on tree and hedge automata.

We wanted to provide a system that would support graphics designers when they are specifying style rules and style sheets [8, 19]. Since we expect such designers to be neophyte computer users we had earlier decided to use a declarative approach to stylesheet specification. Traditional stylesheet specification in publishing houses is declarative; therefore, designers would be on familiar ground. Surely, they would find macro programming, procedural specification and attribute manipulation nontrivial.

We decided to separate the specification of context from the issue of style specification since, in our experience, almost all style rule difficulties are caused by contextual issues. One observation about this separation is in order: it is sufficient to provide a mechanism that determines whether a specific context of a given part of a document is present or is not. The style rules may then incorporate conditional statements or expressions to express context-dependent choices. We can, based on this observation, isolate context determination from style-rule syntax to a large extent. The preliminary work on DESIGNER did just that [7, 13]. Once we have isolated the specification of context from the more general specification of stylesheets, we are able to provide inexperienced users with better support for this aspect of style specification. Indeed, it also frees us to consider different techniques (different from attribution, for example) for context specification.

We make the well-accepted assumption that a set of similarly structured documents is modeled by the (abstract) syntax trees of a given grammar (an XML DTD or the XHTML DTD) that generates the set of all such documents. Fig. 1 provides the tree view of a document instance that conforms to the DTD given in Fig. 2; Fig. 3 provides the XML source view of the document instance and Fig. 4 is a formatted view of the same document instance.

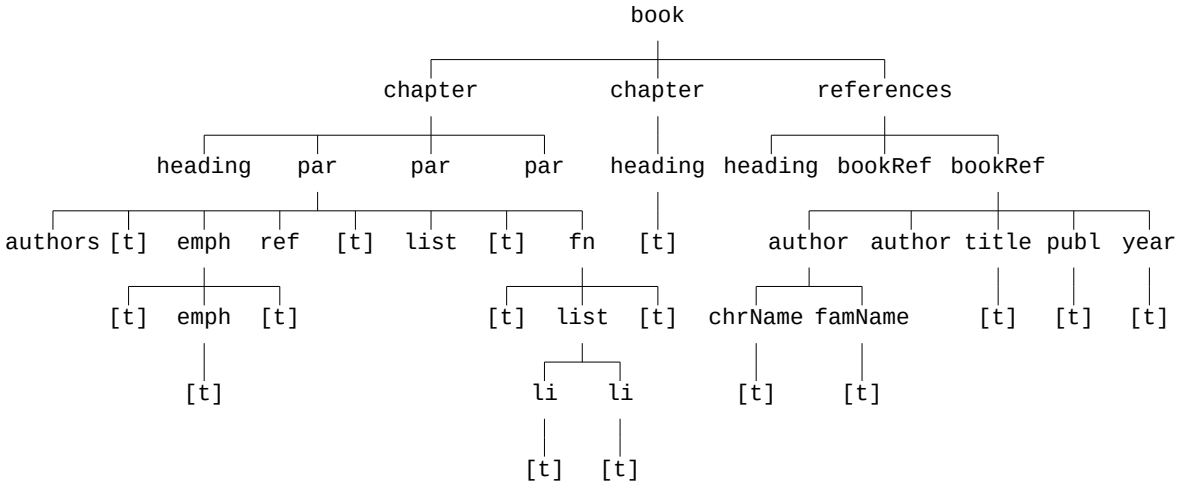


Figure 1: An example document tree for the XML DTD of Fig. 2.

The paper is organized as follows. After presenting a rapid overview of XML, DTDs and document instances, in Section 2, we present, in Section 3, some examples of how context is and can be used in style-rule specification. In Section 4, we briefly discuss the different approaches to style-rule and contextual specification that have been and are being used. In Section 5, the

```
<!ELEMENT book (chapter+,references)>
<!ELEMENT chapter (heading,par*)>
<!ATTLIST chapter id ID #IMPLIED>
<!ELEMENT heading (#PCDATA)*>
<!ELEMENT par (#PCDATA | authors | emph | ref | list | fn)*>
<!ELEMENT emph (#PCDATA | emph)*>
<!ELEMENT ref EMPTY>
<!ATTLIST ref refid IDREF #REQUIRED>
<!ELEMENT list (li)*>
<!ELEMENT li (#PCDATA | emph | list | fn | ref)*>
<!ELEMENT fn (#PCDATA | emph | list | fn | ref)*>
<!ELEMENT references (heading,bookRef*)>
<!ELEMENT bookRef (author*,title,publ,year)>
<!ATTLIST bookRef id ID #IMPLIED>
<!ELEMENT authors (author)+>
<!ELEMENT author (chrName,famName)>
<!ELEMENT chrName (#PCDATA)*>
<!ELEMENT famName (#PCDATA)*>
<!ELEMENT title (#PCDATA)*>
<!ELEMENT publ (#PCDATA)*>
<!ELEMENT year (#PCDATA)*>
```

Figure 2: An example XML DTD that defines books.

main section of the paper, we introduce the basic ideas underlying our approach to contextual specification. Specifically, we define T-configurations and T-graphs and the notion of T-graph matching. Also, we show how we can reduce T-graph matching to pattern matching with wild-card symbols in the pattern. We demonstrate a similar reduction for demonstrating that two T-graphs are conflict free, a notion that we define. In Section 6, we introduce define the kernel operations needed for a T-graph editor and we provide their semantics. We close with a brief discussion of the pros and cons of T-graphs, in Section 7, and, in Section 8, we discuss our results and possible future directions.

2 An XML primer

We briefly review the notions of XML, DTDs and document instances.

An XML document consists of an XML prolog and a **marked-up document instance.** The prolog contains a **document type definition (DTD)** that is an extended context-free grammar in which the right-hand sides of productions are both generalized and restricted regular expressions. In this regard, DTDs are similar to grammars in extended Backus–Naur form. An example of an XML DTD is given in Fig. 2; XHTML [28] is an example of a more complex XML DTD.

The productions of a DTD are called **element type definitions.** The left-hand sides of element type definitions are, essentially, nonterminals in grammar terminology. The right-hand sides of element type definitions are extended and restricted regular expressions called **content models.**

The DTD in Fig. 2 defines a simple document type for books. It defines a book as a sequence of chapters followed by references. The element `chapter` consists of subelements `heading` and `par` that must appear in the order `heading` followed by a sequence of `par`. The element `reference`

```
<?xml version="1.0"?>
<!DOCTYPE book SYSTEM "book.dtd">
<book>
<chapter id="scanParseCH">
<heading>Scanning and Parsing</heading>
<par><authors><author><chrName>A.V.</chrName><famName>Aho</famName></author>
<author><chrName>R.</chrName><famName>Sethi</famName></author>
<author><chrName>J.D.</chrName><famName>Ullman</famName></author></authors>
in <emph>their <emph>highly popular</emph>
textbook on compiler construction</emph> <ref refid="dragonBook"/>
introduce
<list><li>scanning and</li>
      <li>parsing</li></list>
as the first two phases in compiler construction.<fn>The other
three phases are <list><li>semantic analysis,</li><li>code
generation, and</li><li>code optimization</li></list></fn></par>
<par><authors><author><chrName>J.E.</chrName><famName>Hopcroft</famName></author>
<author><chrName>J.D.</chrName><famName>Ullman</famName></author></authors>
<ref refid="loomBook"/> provide the theoretical foundation for
scanning and parsing.</par>
<par>In <ref refid="laterPhasesCh"/>
we are looking at the later phases.</par>
</chapter>

<chapter id="laterPhasesCh">
<heading>The Later Phases</heading>
</chapter>

<references>
<heading>List of References</heading>
<bookRef id="dragonBook">
<author><chrName>A.V.</chrName><famName>Aho</famName></author>
<author><chrName>R.</chrName><famName>Sethi</famName></author>
<author><chrName>J.D.</chrName><famName>Ullman</famName></author>
<title>Compilers: Principles, Techniques, and Tools</title>
<publ>Addison-Wesley</publ><year>1986</year>
</bookRef>
<bookRef id="loomBook">
<author><chrName>J.E.</chrName><famName>Hopcroft</famName></author>
<author><chrName>J.D.</chrName><famName>Ullman</famName></author>
<title>Introduction to Automata Theory, Languages and Computation</title>
<publ>Addison-Wesley</publ><year>1979</year>
</bookRef>
</references>
</book>
```

Figure 3: A document instance of the XML DTD of Fig. 2.

consists of subelements `heading` and `bookRef` that must appear in the order `heading` followed by a sequence of `bookRef`. The element `par` consists of subelements `authors, emph, ref, list, fn` and `[t]` in any order, whereas the element `bookRef` consists of subelements `author*, title, publ` and `year` in exactly that order. Observe that the elements `emph` (emphasis), `list` and `fn` (footnote) are recursive. Also notice that we have replaced the keyword #PCDATA with the string `[t]` in Fig. 3; both notations denote unstructured strings.

The structural elements of a document instance are indicated by enclosing them in matching

Chapter 1
Scanning and Parsing

Aho et al. in *their **highly popular** textbook on compiler construction* [1] introduce

(1) scanning and
(2) parsing

as the first two phases in compiler construction.*

Hopcroft and Ullman [2] provide the theoretical foundation for scanning and parsing.

In Chapter 2 we examine the later phases.

Chapter 2
The Later Phases

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman: *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] J.E. Hopcroft and J.D. Ullman: *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

* The other three phases are (3) semantic analysis, (4) code generation, and (5) code optimization

Figure 4: A formatted version of the document instance Fig. 3.

pairs of **start tags** and **end tags.** One possible instance of the book DTD given in Fig. 2 is shown in Fig. 3.

Although we do not give a complete formal definition of a document tree, we need to introduce some minimal terminology and notation for document trees that we make use of when formalizing T-graph matching in Section 5.

A **document tree** is a node-labeled, rooted, ordered tree. It is usually called an abstract syntax tree in the programming languages community.

A document tree is specified by a set N of nodes, a set E of directed edges and a labeling function DLab : $N \longrightarrow \Sigma$, where Σ is a label alphabet, that associates a label with each node of the tree.

3 Context examples

General design rules suggest that elements of the document that are logically or structurally identical should also be laid out identically. There are, however, exceptions to this rule, due to tradition or aesthetics, or because the context of an element requires nonstandard treatment. Thus, graphic artists need the facilities to make the visual semantics of an element type conditional on the context of its instantiation in the document. Fig. 4 exhibits a number of design aspects that call for context-dependent processing:

1. All paragraphs (element `par`) are indented, with the only exception being paragraphs immediately preceded by a heading.

2. The chapter headings are numbered, whereas the heading of the reference section is not.

3. Three coauthors (or coworkers) such as Aho, Sethi and Ullman are referred to as "Aho et al.", whereas two coauthors such as Hopcroft and Ullman are referred to as "Hopcroft and Ullman".

4. In footnotes (element `fn`), list items are run in instead of being placed on a new line.

5. The wording of cross references depends on the type of the object to which they refer; that is, "Chapter 2" and "Aho et al.".

6. In a bibliographic item (element `bookRef`) in a list of references, what text is inserted after each author's name depends on the number of preceding and following authors. For example, whether ":", ",␣", "␣and␣", or "␣and" is inserted after an author's name (element `author`).

In these examples, the style rule that is to be applied to a specific element depends on the position of the current element instance within the structure of the document. Often this position is solely characterized by the element types higher up in the document hierarchy (Examples 2 and 4). Sometimes, however, it is necessary to take the left or right siblings (Example 1 and 6) or the internal structure of the current element instance (Example 3) into account or even to consider the structure of the element it refers to (Example 5).

Given a node in a document tree (that is, a specific element in a document), its context is a condition on the other nodes, their labels, and their structural relationship.

4 Context-dependent methods

Stylesheet-driven authoring and layout systems support logical markup to separate form and content. They have two characteristic properties: (1) A style rule specifies the layout for all elements of one type; and (2) a stylesheet (a collection of style rules) is applicable to a whole class of documents. Current style-driven systems fall into a spectrum that extends from the purely **substitutional** to the fully **programmable** approach. Substitutional systems replace an element's logical markup tags with formatting instructions that affect the typesetting of only the marked-up element. Programmable systems, on the other hand, replace an element's logical markup tags with programs that operate on the marked-up element. These programs can eventually also produce formatting instructions. In addition, the programming language that these programs are written in need not be computationally restricted.

An essential feature of DESIGNER is that style rules can be context dependent. We review briefly how some other systems and stylesheet languages support context dependencies.

4.1 Systems and context dependence

One method for supporting context dependence that has often been used is to use variant element names in different contexts. Such an ad-hoc solution is called **illogical logical markup** by Hayashi et al. [18]. Macro packages such as LaTeX have some context dependencies hard-wired into them in an ad-hoc manner. Other systems such as Bravox [21] support subordinate style rules; that is, they allow the user to locally redefine a style rule within the scope of another element type. Subordinate style rules are also provided by `USELINK` declarations in SGML [17]. Subordinate style rules, although more powerful than Scribe's parameter-inheritance scheme [29], are too weak to express all the context dependencies listed in Section 5. They fail, for example, for the context-dependent indentation of paragraphs and for the citations of two and more authors.

In Designer, we use what is called a **structure-based stylesheet model;** that is, we assign style rules to element types. If we disregard Designer's context predicates for a moment, then a more general approach is to assign style rules to *appearances* of element types in a document grammar; that is, to use a **grammar-based stylesheet model** [25, 17, 31]. To illustrate the grammar-based model, consider the following two productions of a document grammar with style rules, rule 1 and rule 2. The two style rules are assigned to the two appearances of *paragraph* in the grammar:

$$article = (paragraph[rule\ 1] + quotation)^*.$$
$$quotation = (paragraph[rule\ 2])^*.$$

One effect that we may wish to achieve is that paragraphs at the outermost level of articles are not indented (rule 1) whereas paragraphs inside quotations are indented (rule 2). Therefore, grammar-based systems are capable of specifying some context-dependent processing. The weak point of this approach is that a document grammar may have to be adapted to the context dependencies that are needed in specific processing. For example, let us assume that we have a production

$$section = heading\ paragraph^*.$$

If we want all paragraphs with the exception of the first paragraph after the heading to be first-line indented, we have to change the grammar to separate the first appearance of *paragraph* from its subsequent appearances:

$$section = heading(paragraph[rule\ 3])(paragraph[rule\ 4])^*.$$

Now the document grammar depends on the layout we want to achieve although, from a logical or structural point of view, the first paragraph should be no different from the others.

Thus, the structure-based model and the grammar-based model both have to be enhanced to handle context dependencies competently. For grammar-based systems, attributed grammars [16] and grammar-based coordination between attributed grammars [6, 30] have been used.

As graphic artists usually explain their work by example, it is hard to argue which formal model comes closest to their work. We believe that they follow a structure-based approach, rather than a grammar-based one, since the notion of a document grammar does not come up at all, even if the layout for a whole series of books such as the Penguin books [24] is defined.

In principle, Designer's stylesheet mechanism does not depend on document grammars; therefore, Designer essentially ignores a document's grammar. Hence, it is appropriate to ask whether document grammars are irrelevant for design specifications. There are two answers: First, graphic artists can profit from knowing the document grammar when they formulate context predicates. For, if a stylesheet applies to only documents of restricted structure, then a context specification is often easier to define and is also shorter. Second, it is an interesting research question whether document grammars can be used to simplify the evaluation of context predicates.

4.2 Style-sheet languages and contextual dependence

We now briefly review the contextual methods available in some current stylesheet languages and approaches.

CSS

Cascading Style Sheets [4, 23] provide a simple stylesheet mechanism that is oriented towards fast implementation for XML documents [5]. A CSS context expression, the link between a document and a CSS stylesheet, is called a **selector.** CSS selectors can test for ancestral and left sibling relationships in a document tree. Selectors are strings that are interpreted as search patterns on the stack of open elements (the ancestors) during the processing of a contextual node; the selectors also include combinators. As a result the selectors can also check for the parent and first child, for example.

XSL

Extensible Style Language [1] is a stylesheet language for XML [5] that provides context-dependent style rules. Context expressions in XML are called patterns that are strings that are matched against contextual elements. As in CSS, the simplest pattern that matches the contextual node consists of only the element-type name. One can also use an any-element-name that matches any, but only one, element name. It is also possible to check whether an element has a child element of a particular type. In addition, positional qualifiers may be used to test an element's position or uniqueness among its siblings. More complex patterns may be specified in XSL than in CSS by checking for ancestral and parental relationships as well as checking for an element being the root node of a document tree.

DSSSL

The Document Style Semantics and Specification Language [20] defines the most complex style-specification system. It offers two different mechanisms to handle context-dependent processing.

To express simple context-dependent style rules one can use a string, as in CSS, that is interpreted as a search pattern on the stack of open elements when a contextual node is processed.

To express more complex conditions that may include access to attributes or any other information that can be deduced from the document itself one has to write code in DSSSL's expression language. The expression language is a full programming language that is derived from Scheme; therefore, it is possible to specify every possible context in this language as well as its semantics.

Proteus

The Proteus system [26] is a prototype document processing system. It does not offer any simple mechanism to specify ancestral or parental relationships. Instead all context expressions have to be constructed as an if-then-else cascade that can include queries of the logical structure of a document using the typical ADT tree operations.

Tree transformations

Murata [27] defines a general method of processing document trees that is based on a tree- and hedge-oriented view of documents, where a hedge is an ordered sequence of trees. The ideas stem

from tree automata theory; hence, they are rigorously grounded. He defines a tree view of context and defines transformations on trees that can be used for layout semantic specification as well as for document retrieval, document views and docubase queries. Murata's notion of context is much broader than that of Brüggemann-Klein et al. [9] and of Brüggemann-Klein and Wood [14] who use caterpillar expressions to specify context. There is however some overlap with the two approaches that has led to joint work on an appropriate theory of tree and hedge automata [12].

5 T-configurations and T-graphs

We now define and discuss T-configurations and T-graphs, and formally define and investigate T-graph matching.

5.1 T-configurations: Definition

Document trees have node labels from some element alphabet Σ. They satisfy the local constraint that the label of a parent node and the sequence of labels of its children conform to a production of the tree's DTD.

Since we view XML documents as trees, element names (or nonterminals) are the node labels. The content of a document is represented at an external node (or leaf) of such a tree whose label is also in Σ (in XML the label is #PCDATA; we have used `[t]` in Fig. 1). Each XML DTD defines a set of trees. Note though that not every set of trees we discuss needs to be defined by a DTD.

We assume that we can access a document tree using the basic navigational and checking operations that define an ADT for trees. Specifically, given a node v in a document tree, we can move up one node in the tree to v's parent. Similarly, we can move left or right from v to v's immediate left sibling or right sibling, respectively. Conceptually, for each node v in each document tree D, we assume that there are endmarker nodes l, r and t that are the leftmost sibling of v, the rightmost sibling of v and the topmost ancestor of v (which is the root). The endmarker nodes are labeled with the special endmarker symbol ∎; we also assume that ∎ is in Σ, the label alphabet. The endmarker nodes allow a user of the ADT to detect when a move falls off a document tree. We use the endmarker nodes when matching context specifications.

The underlying idea behind T-configurations is that each node v in a document tree D defines a set

$$\{w \in D : w \text{ is an ancestor of } v \text{ or } w \text{ is a sibling of } v\}$$

of nodes that we denote by C_v, where we call v the **contextual node** and C_v a **T-configuration.** The motivation for the name is that each set C_v can be visualized as an up subset of ancestors, a left subset of left siblings and a right subset of right siblings. When we embed these nodes in the plane such that we preserve their natural ordering, connecting adjacent nodes gives an inverted T; see the example in Fig. 5. Note that the endmarker nodes (not shown in Fig. 5) belong to the T-configuration. A T-configuration C_v contains the nodes that are most often used to evaluate a contextual specification for a node v. For example, is a section element v a descendant of one or more section elements? Does a paragraph element have any left siblings (in other words, is it the first paragraph)?

5.2 T-graph matching

Given a node v in a document tree, when we process v with a context-dependent rule, we need to determine whether v satisfies the corresponding context requirement. We translate this approach

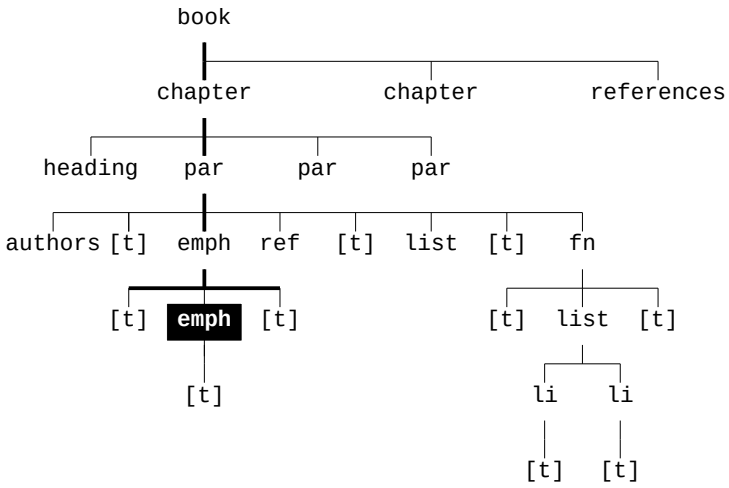


Figure 5: An example T-configuration in the document tree of Fig. 1. Note that we have omitted the endmarker nodes for reasons of clarity.

into an algorithmic framework by restricting where context requirements can occur—in the T-configuration of v—and by specifying context requirements generically with **T-graphs.** If a T-graph matches a T-configuration, in a sense we will make precise, then the T-configuration satisfies the context requirement specified by the T-graph. Examples of T-graphs are given in Figs. 6 and 7. Intuitively, we can think of T-graph matching as stretching the T-graph to anchor it at the contextual and endmarker nodes of the T-configuration. Then, we stretch the stems to locally match the nodes and edges in the T-configuration.

5.3 Formal definition of T-graphs

We define T-graphs more precisely as follows. A T-graph $G = (V, E, \mathrm{TLab}, l, r, t, u)$ is a digraph that has a set V of vertices such that V contains four designated and distinct vertices l, r, t and u: the **left vertex,** the **right vertex,** the **top vertex** and the **contextual vertex,** respectively. We use the terminological convention that "node" always refers to document trees and T-configurations, and "vertex" always refers to T-graphs.

Each T-graph G also has a set $E \subseteq V \times V$ of directed edges and a labeling function TLab. Now, the T-graph G can be decomposed into three subgraphs that we call **stems** (each of which is a line digraph). The three stems have only vertex u in common and all other vertices in V appear in exactly one stem; that is, apart from u the stems are mutually disjoint and together they exhaust G.

The **up stem** is the digraph G_U specified by (V_U, E_U), where $V_U \subseteq V$, $u, t \in V_U$ and E_U defines a line digraph from u to t. The **left stem** G_L, which is specified by (V_L, E_L), and **right stem** G_R, which is specified by (V_R, E_R), are defined in a similar manner. The labeling function TLab labels both the vertices and the edges. A vertex can be labeled with exactly one of three possible types of symbols: A symbol from the document-tree alphabet Σ (it matches only that symbol); Σ itself (the wild-card symbol that matches any symbol from Σ); or ▮, the endmarker symbol (only endmarker vertices can be labeled with ▮). Pictorially, we use solid circles to denote vertices labeled with Σ; empty circles labeled with a symbol from Σ to denote labeled vertices; and ▮ to denote endmarker vertices.

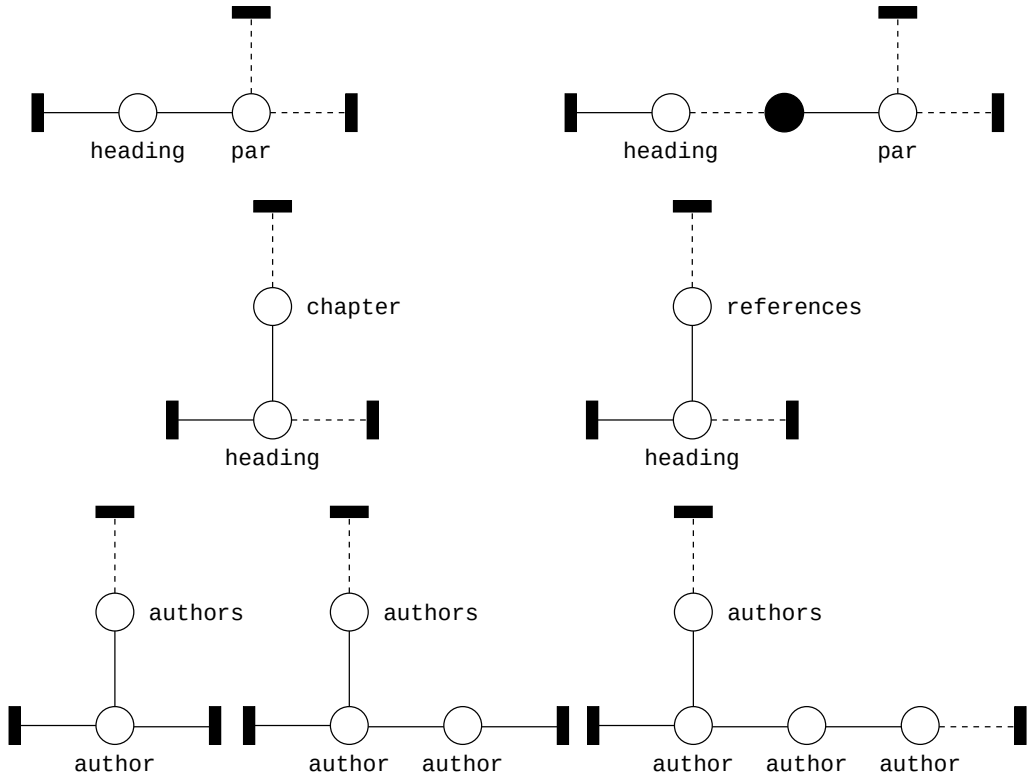


Figure 6: Example T-graphs that illustrate contexts for the document tree in Fig. 1, where a solid circle denotes a wild-card vertex that matches any labeled node. (a) A T-graph that matches the first paragraph, `par`, after a heading (Context Example 1). (b) A T-graph that matches any paragraph, `par`, but not the first, after a heading (Context Example 1). (c) A T-graph that matches a chapter heading, `heading` (Context Example 2) (d) A T-graph that matches a references heading, `heading` (Context Example 2) (e) A T-graph that matches the only author, `author,` of a sequence of authors, `authors.` (f) A T-graph that matches the first author, `author,` of exactly two authors of a sequence of authors, `authors.` (g) A T-graph that matches the first author, `author,` of at least three authors of a sequence of authors, `authors.`

An edge is labeled either with 1 (it matches exactly one edge in a T-configuration) or with + (it matches one or more edges, that is, a path in a T-configuration). Pictorially, we use solid lines to denote edges labeled with 1 and dashed lines to denote edges labeled with +.

Informally, given a document tree D, D's nodes N, D's labeling function DLab, a node v in D and a T-graph $G = (V, E, \mathrm{TLab}, l, r, t, u)$, G **matches** the T-configuration C_v determined by v if the following conditions hold: *There is an injection $\iota : V \longrightarrow C_v$ from the vertices of G to the nodes C_v of D such that $\iota(u) = v$ and $TLab(u) = DLab(v)$, and the endmarker vertices l, r and t of G are mapped by ι to the corresponding endmarker nodes of C_v.*

The injection must also satisfy the following local conditions:

1. If x is in V_U, then $\iota(x)$ is an ancestor of v.
2. If x is in V_L, then $\iota(x)$ is a left sibling of v.
3. If x is in V_R, then $\iota(x)$ is a right sibling of v.

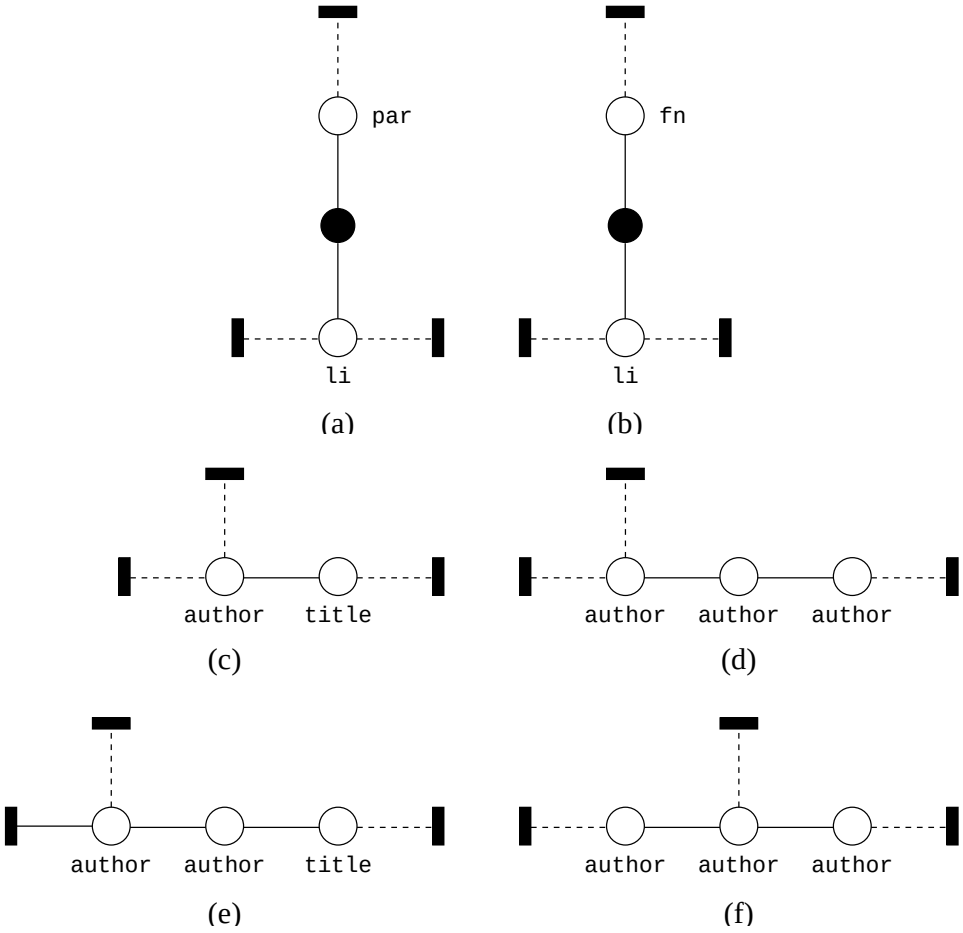


Figure 7: Example T-graphs that illustrate contexts for the document tree in Fig. 1, where a solid circle denotes a wild-card vertex that matches any labeled node. (a) A T-graph that matches any list item, `li`, that appears in a paragraph, `par` (Context Example 4). (b) A T-graph that matches any list item, `li`, that appears in a footnote, `fn` (Context Example 4). (c) A T-graph that matches the last author in a book reference, `bookRef` (Context Example 6). (d) A T-graph that matches any author who is followed by at least two authors in a book reference, `bookRef` (Context Example 6). (e) A T-graph that matches the first author of a two-author book in a book reference, `bookRef` (Context Example 6). (f) A T-graph that matches any author who is neither the first nor the last author in a book reference, `bookRef` (Context Example 6).

4. If (x, y) is in E_U, then either $\mathrm{TLab}((x, y)) = 1$ and $\iota(y)$ is the parent of $\iota(x)$ or $\mathrm{TLab}((x, y)) = +$, $\iota(y) \neq \iota(x)$ and $\iota(y)$ is an ancestor of $\iota(x)$.

5. If (x, y) is in E_L, then either $\mathrm{TLab}((x, y)) = 1$ and $\iota(y)$ is the immediate left sibling of $\iota(x)$ or $\mathrm{TLab}((x, y)) = +$, $x \neq y$ and $\iota(y)$ is a left sibling of $\iota(x)$.

6. If (x, y) is in E_R, then either $\mathrm{TLab}((x, y)) = 1$ and $\iota(y)$ is the immediate right sibling of $\iota(x)$ or $\mathrm{TLab}((x, y)) = +$ and $\iota(y)$ is a right sibling of $\iota(x)$.

7. If x is in V and x is not a wild-card, then $\mathrm{DLab}(\iota(x)) = \mathrm{TLab}(x)$. Note that this condition automatically holds for endmarker vertices since they must be mapped to the endmarker

nodes.

5.4 A T-graph matching algorithm

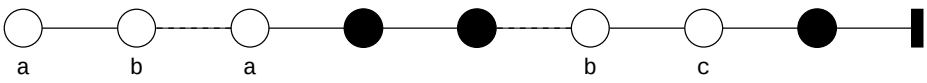
The basic algorithmic result is that T-graph matching can be carried out efficiently. We carry out a sequence of straightforward reductions to simplify the T-graph matching problem.

Reduction 1: Clearly, we can solve the matching problem for each stem of the T-graph independently.

As a result of this claim, we discuss the matching problem for only a right stem, the other two stems can be dealt with in a similar manner.

Reduction 2: We can reduce the right-stem matching problem to a more traditional pattern-matching problem. We translate a right stem into a **pattern expression** as follows: *Replace each labeled vertex with its label, omit each* 1*-edge, replace each* +*-edge with* Σ^* *and replace the right endmarker with* ∎.

Observe that we are, essentially, keeping only the labeled vertices. Now, since there is an edge between each pair of adjacent vertices, 1-edges are redundant. On the other hand, +-edges correspond to a sequence of wild-card vertices, possibly none at all; hence, they are captured by the translate Σ^* in the corresponding position of the pattern expression. We have translated a right stem R into a regular-like expression p_R. For example, the right stem shown in Fig. 8(a) is translated into the regular-like expression p_R in Fig. 8(b). A pattern expression never begins or ends with Σ^*.



(a)

$$a\,b\,\Sigma^*\,a\,\Sigma\,\Sigma\,\Sigma^*\,b\,c\,\Sigma\,\blacksquare$$

(b)

Figure 8: Translation of a right stem into a pattern expression. (a) A right stem. (b) The corresponding pattern expression.

Corresponding to a pattern expression p_R, we obtain the text string t_R that is the string of labels of the nodes in the right stem of the given T-configuration. Clearly, the text string begins with the label of the contextual node and ends with ∎.

Now, although there may be many matches of a pattern expression with the text string, it suffices to determine whether there is either no match at all or at least one match. The essence of the reduction of T-graph matching to regular-expression matching is captured as follows:

Claim 1: *Given a T-graph* G *and the T-configuration of a node* v *in a document tree* D, G_R *matches the right stem of node* v *in* D *if and only if* p_R *matches* t_R*; that is, if and only if* t_R *is in the language of* p_R.

Reduction 3: We now simplify the pattern-matching process even more by partitioning a pattern expression p_R into its **rigid components** that correspond to maximal star-free subexpressions of p_R. We write $p_R = r_1\Sigma^* \cdots \Sigma^* r_k$ for rigid components $r_1, \ldots, r_k$, where $k \geq 1$ (note that p_R must include at least the contextual vertex and the endmarker vertex).

For example, the pattern expression

$$a\,b\,\Sigma^*\,a\,\Sigma\,\Sigma\,\Sigma^*\,b\,c\,\Sigma\,\blacksquare$$

of Fig. 8(a) is partitioned into the rigid components

$$(a\,b),\ (a\,\Sigma\,\Sigma),\ (b\,c\,\Sigma\,\blacksquare).$$

Note that the rigid components of p_R are separated by Σ^*.

Assume that p_R matches t_R. Then, we claim that we can match the rigid components separately, greedily and without overlaps. Since p_R matches t_R, there must be a leftmost match of r_1 from the first symbol (the label of the contextual node) of t_R that leaves an unexamined suffix t_R^2 of t_R. Similarly, there must be a match of r_2 in t_R^2 and, again, we have an unexamined suffix t_R^3 of t_R^2. Continuing in this greedy manner, since p_R matches t_R, the rigid components $r_1, \ldots, r_k$ must match t_R in such a way that the matches of r_i and r_j do not overlap, for all i and j, $i \neq j$, and, in addition, they must match in the given order from left to right. Observe that because both r_k and t_R end with $\blacksquare$ and this symbol does not appear elsewhere in r_k or in t_R, greedy matching of r_k always fails with the possible exception of when the last symbol of r_k and the last symbol of t_R (endmarker symbols) are aligned. For this reason, in practice it makes sense to first match r_k, from right to left before matching $r_1, \ldots . r_{k-1}$ from left to right.

Claim 2: *Given a pattern expression $p_R = r_1\Sigma^* \cdots \Sigma^* r_k$, $k \geq 1$, and a text string t_R, we can reduce the corresponding pattern-matching problem to the greedy matching of the rigid expressions $r_1, \ldots, r_k$ in that order. That is, p_R matches t_R if and only if r_1 matches t_R, r_2 matches t_R^2, ..., r_k matches t_R^k, where t_R^i is the remaining unexamined suffix of t_R after $r_1, \ldots, r_{i-1}$ have been matched.*

We have only one remaining problem to solve; namely, how do we do rigid-component matching efficiently? We propose the use of the Aho–Corasick algorithm [2, 15] that computes occurrences in a given text of any string in a given finite set of pattern strings. The algorithm has two phases: In the first phase, the preprocessing phase, a deterministic finite-state automaton is efficiently computed for the given set of pattern strings and, in the second phase, the automaton processes the text string to find the first match of a pattern string in the text. In addition, the size of the automaton is linear in the total number of symbols in the pattern strings.

For our application, we can assume that preprocessing is done when a stylesheet is created. Thus, we need only be concerned about executing the automaton on a given text string.

Consider a rigid component r of a pattern expression p_R. We expand r into the set $L(r)$ of all strings of length $|r|$ obtained by replacing each symbol in $\Sigma \cup \{\blacksquare\}$ by itself and keeping each appearance of Σ. If there are s appearances of Σ in r, then $L(r)$ is a set of $|\Sigma|^s$ strings. The Aho–Corasick algorithm has a worst-case preprocessing time of $O(|r| \times |\Sigma|^s)$. Fortunately, given a text t_R, the automaton for a rigid component r runs in worst-case time $O(|t_R|)$ which is linear in the size of the T-configuration.

Because we use the greedy approach to matching all the rigid components $r_1, \ldots, r_k$ of p_R, we obtain a preprocessing time of

$$O(m_1 \times |\Sigma|^{s_1} + \cdots + m_k \times |\Sigma|^{s_k}),$$

where $m_i = |r_i|$ and s_i is the number of appearances of Σ in r_i, $1 \leq i \leq k$, for matching p_R with t_R. This time bound can be simplified to be

$$O(|p_R| \times |\Sigma|^s),$$

where s is the maximum of $s_1, \ldots, s_k$.

Thus, we can summarize the preceding reductions and claims in the following result.

Theorem 1 *Given a T-graph of size m and a T-configuration of size n, T-graph matching can be implemented to run in time linear in the size of the T-configuration. The algorithm requires a preprocessing time of $O(m \times |\Sigma|^s)$, where s is the maximum number of wild-card vertices in a rigid component of the T-graph.*

The execution of the automaton of the T-graph on the given T-configuration takes $O(n)$ time in the worst case.

Two comments about the Aho–Corasick algorithm are in order. First, we conjecture that in practice there will be few wild-card symbols; therefore, the number of pattern strings in a rigid component r_i will also be small. Hence, we expect the Aho–Corasick algorithm to run in almost linear time.

Second, rather than using the Aho–Corasick algorithm, it is easier to use the basic pattern matching algorithm [15] that, for a pattern string $p = p_1 \cdots p_l$ and a text $t = t_1 \cdots t_L$, compares p_1 with t_1, p_2 with t_2, ..., p_l with t_l until either two symbols are different or p matches t at position 1. This description of the algorithm is incomplete in that it does not say what happens when it meets a wild-card symbol in p. In this case, p_i always matches t_i, whatever symbol t_i is.

In general, if p does not match t at position j, then we repeat the matching process at position $j+1$ of t; that is, we compare p_1 with t_{j+1}, p_2 with t_{j+2}, ..., p_l with t_{j+l}. This algorithm guarantees to find a match if there is one.

Its running time, in the worst case, is $O(l \times L)$. (A worst-case example is $p = a \cdots a$ which at every position i in t, $1 \leq i \leq |t| - |p| + 1$, makes $|p|$ comparisons.) Thus, for a rigid component r and a stem t, the basic algorithm gives a worst-case run time of $O(|r| \times |t|)$. The basic algorithm runs in worst-case time $O(m \times n)$ but it uses only $O(m+n)$ space. We conjecture that, in practice, it runs in $O(m+n)$ time since we expect there will not be many wild-card symbols.

6 T-graph editing

As we have mentioned in Section 5, we can think of T-graph matching as follows: Stretch the T-graph to anchor it at the contextual and endmarker nodes of the T-configuration and then stretch the stems to match the nodes and edges in the T-configuration. Given this scenario, the kernel T-graph editing operations are not surprising. We need to be able to navigate in a T-graph, to edit vertex and edge labels and to update a T-graph. Navigationally, we need to be able to traverse a T-graph by vertices and edges. We need to be able to change the label of a vertex to a different symbol including the wild-card symbol, and to change a fixed edge into a path, or a path into a fixed edge. Lastly, we need update operations: insertion and deletion of vertices and edges. We choose to define the insertion and deletion only for vertices that implicitly add and remove edges.

At each editing step, we assume that there is always either a current vertex or a current edge—the editing operations always operate on the current vertex or edge. We specify the basic editing operations for T-graphs and also define their semantics as follows:

Navigational operations:

1. Initialize current object to be either the contextual vertex or one of the endmarker vertices.
2. Change the current object to be either the next or the previous vertex of the current vertex. The previous vertex is undefined if the current object is the contextual vertex and the next vertex is undefined if the current object is an endmarker vertex. This operation is undefined if the current object is an edge.

3. Change the current object to be the outgoing or the ingoing edge of the current vertex. This operation is undefined if the current object is an edge.
4. Change the current object to be the source or the target vertex of the current edge. This operation is undefined if the current object is a vertex.

Vertex-label operations:

1. $\mathrm{TLab}(x) = a \implies \mathrm{TLab}(x) = b$, where $a, b \in \Sigma$.
2. $\mathrm{TLab}(x) = \Sigma \implies \mathrm{TLab}(x) = a$, where $a \in \Sigma$.
3. $\mathrm{TLab}(x) = a \implies \mathrm{TLab}(x) = \Sigma$, where $a \in \Sigma$.

Note that endmarker vertices are always labeled ▮; hence, they cannot be relabeled.

Edge-label operations:

1. $\mathrm{TLab}((x, y)) = 1 \implies \mathrm{TLab}((x, y)) = +$; an edge is replaced by a path.
2. $\mathrm{TLab}((x, y)) = + \implies \mathrm{TLab}((x, y)) = 1$; a path is replaced by an edge.

Vertex operations:

1. Delete(x), where x is the current object. This operation is undefined if x is an edge or if x does not have two adjacent edges (that is, if x is either an endmarker or the contextual vertex). If the operation is legal, $V \implies V - \{x\}$, $E = E - \{(w, x), (x, y)\} \cup \{(w, y)\}$, $\mathrm{TLab}((w, y)) = 1$ and the current vertex becomes y (which was the next vertex).
2. Insert(e), where e is the current object. This operation is undefined if e is a vertex. If the operation is legal, then $V \implies V \cup \{x\}$, where x is a new vertex, $E = E - \{e\} \cup \{(w, x), (x, y)\}$, where $e = (w, y)$, $\mathrm{TLab}((w, x)) = \mathrm{TLab}(e)$, $\mathrm{TLab}((x, y)) = \mathrm{TLab}(e)$ and (w, x) is the new current object.

Note that these rules prevent the insertion and deletion of endmarker and contextual vertices.

7 T-graphs: Pros and cons

Some advantages of the T-configuration and T-graph approach are:

1. The T-configuration of a node in a document tree consists of the nodes that are commonly used for context determination—a 90 percent solution.
2. T-graphs can be visualized easily, by humans and by user interfaces.
3. T-graph matching can be implemented efficiently.
4. We can easily specify the complete semantics of a T-graph editor.

The obvious disadvantage of T-graphs is that they do not capture all reasonable contexts. We discuss four specific context problems that T-graphs cannot handle. The first two problems are caused by going beyond T-configurations and T-graphs. The second two problems, however, demonstrate that within T-configurations there are contexts that T-graphs cannot specify.

1. T-graphs cannot specify descendant contexts. For example, is there a node labeled a properly in the subtree of the contextual node?

   One possible extension of T-graphs to accommodate such specifications is to use what we call **fork graphs.** A fork graph is a T-graph with tines from the siblings; it looks like a dinner fork (or an octopus). The price we pay for such an extension is that it *is more complex* and, therefore, harder to understand and to use.

2. T-graphs cannot specify contexts that involve neighbors; that is, that cut across the document-tree structure as required by the cross-reference problem (see Context Example 5 in Section 3).

   We would like to be able to automatically add the appropriate boilerplate text to a cross reference. Cross references are, however, such a special case that perhaps they should be handled separately. Indeed, DESIGNER provides a concept, called **beamer**, that can be used to copy (beam up) a referenced item to the referring element. Consequently, context-dependent style rules using T-graphs can be employed to format each copy as appropriate. DSSSL [20] and XSL [1] support structure transformations *before applying style rules* to handle the cross referencing problem.

3. T-graphs cannot specify **nested-emphasis contexts.** For example, in the document tree in Fig. 1, the node labeled `emph` has another node labeled `emph` as its descendant. The corresponding DTD in Fig. 2 shows that the nesting of `emph` in `emph` can continue to any depth. The semantics of `emph` is that if a `[t]` node has an even number of `emph` ancestors, then it is typeset in Roman; if it has an odd number of `emph` ancestors, then it is typeset in Italic.

   Even though only the ancestors of a contextual node need be examined (see Figs. 2 and 9), T-graphs cannot determine parity. The reason is simple: Each T-graph can include only a

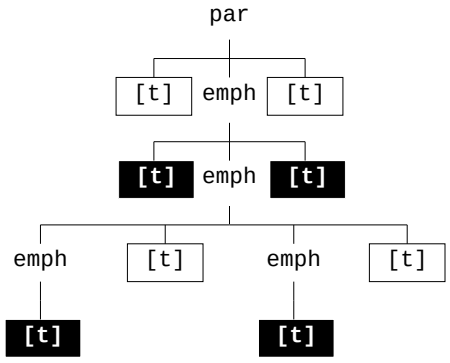


Figure 9: A document tree with nested emphasized text.

   finite number of `emph` vertices. We have not considered, in the use of T-graphs, the possibility of allowing an infinite set of T-graphs.

4. T-graphs cannot handle what we call the **two-ancestor problem:** *Does a contextual node v have exactly two ancestors labeled a?*

   An obvious candidate for such a T-graph is given in Fig. 10. Unfortunately, this T-graph will also match a contextual node v that has three or more ancestors labeled a.

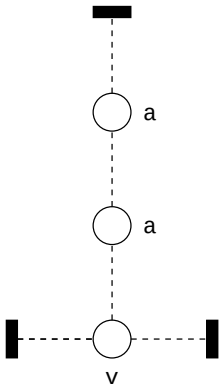


Figure 10: A candidate T-graph for the two-ancestor problem.

8 Closing remarks

As always with implementations, the implemented version of T-graphs in DESIGNER [8, 19] differs somewhat from the T-graphs we have defined. The major difference is that the full T-graph editor has not been implemented. Nevertheless, we have demonstrated the viability of the T-graph approach. More important, from the graphic designers' viewpoint, we have also implemented a more visual version of T-graph specification and editing. Although we have not yet carried out appropriate cognitive experiments to test the learnability and usability of the T-graph concept, we believe that such tests will not falsify the approach.

In addition, it appears that T-graphs are indeed a 90 percent solution in that we can capture most, although not all, real-world contexts with them.

We have proposed [9, 14] a different approach to context specification based on tree walking and paths that uses **caterpillar expressions.** Each stem in a T-graph can be described by a caterpillar expression; however, the T-graph as a whole cannot, in general, be described by a single caterpillar expression. The reason is simple: caterpillar expressions, as currently defined, cannot simultaneously distinguish between "to the left of a node" and "to the right of a node". For example, if we want to capture contextual nodes that have a node labeled a to their left and a node labeled b to their right, we cannot specify this relationship with a caterpillar expression. (Note that it is easy to construct a T-graph that checks this relationship.)

We have implicitly assumed that we never access more than one T-graph at the same time. But, we may wish to generalize a context specification by combining two or more T-graphs to give the union of the contexts they specify. Alternatively, we may wish to break a complex context specification into partial specifications for which T-graphs are easier to construct. We then need to combine the resulting T-graphs. One obvious question to ask in both these scenarios is whether two or more T-graphs can all match a given T-configuration; indeed, whether there is a T-configuration that has this property. This question can be reduced to asking whether the right stems of two T-graphs can match the right stem of some T-configuration. This problem has been studied in pattern matching when both the text and pattern have wild-card symbols; however, the difference here is that we are not given a text, we are given only two "patterns". It is an interesting open problem whether we can solve this T-graph-matching problem efficiently.

References

[1] Sharon Adler, Anders Berglund, Jeff Caruso, Stephen Deach, Paul Grosso, Eduardo Gutentag, Alex Milowski, Scott Parnell, Jeremy Richman, and Steve Zilles. Extensible stylesheet language (XSL) 1.0. http://www.w3.org/TR/2000/CR-xsl-20001121/, November 2000.

[2] A.V. Aho and M.J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18:333–340, 1975.

[3] H. Alblas and B. Melichar. *Attribute Grammars, Applications and Systems.* Springer-Verlag, Heidelberg, 1991. LNCS 545.

[4] Bert Bos, Håkon Lie, Chris Lilley, and Ian Jacobs. Cascading Style Sheets, Level 2: CSS2 Specification. http://www.w3.org/TR/1998/REC-CSS2-19980512/, May 1998.

[5] T. Bray, J. P. Paoli, C. M.Sperberg-McQueen, and Eve Maler. Extensible markup language (XML) 1.0 (second edition). http://www.w3.org/TR/2000/REC-xml-20001006/, October 2000.

[6] A. L. Brown, Jr., T. Wakayama, and H. A. Blair. A reconstruction of context-dependent document processing in SGML. In C. Vanoirbeek and G. Coray, editors, *EP92*, The Cambridge Series on Electronic Publishing, pages 275–290, Cambridge, 1992. Cambridge University Press.

[7] A. Brüggemann-Klein. Formal Models in Document Processing, 1993. Habilitationsschrift, Institut für Informatik, Universität Freiburg.

[8] A. Brüggemann-Klein and S. Hermann. Design by example. In F. Rowland and J. Meadows, editors, *Electronic Publishing 1997: New Models and Opportunities*, pages 223–236, 1997. Proceedings of an ICCC/IFIP Conference held at the University of Kent at Canterbury, England, 14.–16. April 1997.

[9] A. Brüggemann-Klein, S. Hermann, and D. Wood. Context and caterpillars and structured documents. In E. Munson, C. Nicholas, and D. Wood, editors, *Principles of Digital Document Processing, PODDP 98*, pages 1–9, Heidelberg, 1998. Springer-Verlag. Lecture Notes in Computer Science 1481.

[10] A. Brüggemann-Klein, S. Hermann, and D. Wood. The visual specification of context. In *Proceedings of the Sixth Conference on Advances in Digital Libraries*, pages 28–36, Los Alamitos, California, 1999. IEEE Computer Society.

[11] A. Brüggemann-Klein, S. Hermann, and D. Wood. Visually specifying context. In J.W.T. Smith, A. Ardo, and P. Linde, editors, *Proceedings of the Third ICCC/IFIP Conference on Electronic Publishing*, pages 103–118, Washington, DC, 1999. International Council for Computer Communications.

[12] A. Brüggemann-Klein, M. Murata, and D. Wood. Regular tree and regular hedge languages over unranked alphabets: Version 1, April 3, 2001. Technical Report HKUST-TCSC-2001-05, Theoretical Computer Science Center, Hong Kong University of Science & Technology, Clear Water Bay, Kowloon, HK SAR, 2001. http://www.cs.ust.hk/tcsc/RR/2001-05.ps.gz.

[13] A. Brüggemann-Klein and D. Wood. Electronic style sheets. Interner Bericht 45, Institut für Informatik, Universität Freiburg, January 1992.

[14] A. Brüggemann-Klein and D. Wood. Caterpillars: A context specification technique. *Markup Languages*, 2(1):81–106, 2000.

[15] Maxime Crochemore and Wojciech Rytter. *Text Algorithms.* Oxford University Press, New York, 1994.

[16] P. Franchi-Zannettacci. Attribute specifications for graphical interface generation. In G. X. Ritter, editor, *Information Processing 89*, volume 11 of *IFIP Congress Series*, pages 149–155, Amsterdam, 1989. North Holland.

[17] C. F. Goldfarb. *The SGML Handbook.* Clarendon Press, Oxford, 1990.

[18] N. Hayashi, K. Saito, H. Nakatsuyama, Y. Suzuki, and M. Murata. Some ODA layout control problems revealed by a prototype editor. In J. André and J. Bézivin, editors, *Woodman '89*, pages 79–90, Rennes, 1989. Bigre. Technical Report No. 63-64.

[19] Stefan Hermann. *Design Specifications in The Digital Publication Process.* PhD thesis, Fakultät für Informatik, Technische Universität München, July 2000.

[20] ISO/IEC 10179: Information technology—Processing languages—Document Style Semantics and Specification Language (DSSSL), 1996. International Organization for Standardization.

[21] J. Johnson and R. J. Beach. Styles in document editing systems. *IEEE Computer*, pages 32–43, January 1988.

[22] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[23] H. Lie and B. Bos. *Cascading Style Sheets: Designing for the Web.* Addison-Wesley Publishing Company, Reading, MA, 1997.

[24] R. McLean. *Jan Tschichold: Typographer.* Lund Humphries, London, 1975.

[25] N. Meyrowitz and A. van Dam. Interactive editing systems (Part I and II). *Computing Surveys*, 14(3):321–415, September 1982.

[26] E. V. Munson. *Proteus: An Adaptable Presentation System for a Software Development and Multimedia Document Environment.* PhD thesis, Computer Science Division, University of California, Berkeley, 1994.

[27] M. Murata. Transformation of documents and schemas by patterns and contextual conditions. In C. Nicholas and D. Wood, editors, *Proceedings of the Third International Workshop on Principles of Document Processing (PODP 96)*, pages 153–169, Heidelberg, 1997. Springer-Verlag. Lecture Notes in Computer Science 1293.

[28] Steven Pemberton and many others. XHTML 1.0: The Extensible HyperText Markup Language: A Reformulation of HTML 4 in XML 1.0. http://www.w3.org/TR/2000/REC-xhtml1-20000126/, January 2000.

[29] B. K. Reid. A high-level approach to computer document formatting. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 24–30, January 1980.

[30] W. Schreiber. *Generierung von Dokumentverarbeitungssystemen aus formalen Spezifikationen von Dokumentarchitekturen*. PhD thesis, Institut für Informatik, Technische Universität München, 1996.

[31] J. Warmer and H. van Vliet. Processing SGML documents. *Electronic Publishing—Origination, Dissemination and Design*, 4(1):3–26, March 1991.