# The Visual Specification of Context*

Anne Brüggemann-Klein† Stefan Hermann‡ Derick Wood§

December 11, 1998

**Abstract**

We introduce a simple and novel visual technique for the specification of context in structured documents called T-graphs that we base on T-configurations in document trees. The technique is implemented in the current version of Designer. Although we are applying this technique in the specification of context-dependent style sheets for HTML, XML, and SGML documents, it is clear that it can be used in other environments such as query specification for structured documents and for computer program transformations.

We compare T-graphs with the context specification techniques found in other style-sheet 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 as a topic in its own right does not appear to have been studied. (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 [1, 17, 27]. In addition, we also expected to be able to use and modify previous approaches to style specification such as suggested by the DSSSL draft document [2], the XSL draft document [20] and Lie's work [18, 19] for SGML [14] and XML [10] style specification.

*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.

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

§Department of Computer Science, Hong Kong University of Science & Technology, Clear Water Bay, Kowloon, Hong Kong. E-mail: dwood@cs.ust.hk.

We also considered Munson's approach in the Proteus system [23] and Murata's more general approach [24] which is based on tree and hedge automata.

We wanted to provide a system that will support graphics designers when they are specifying style rules and style sheets [6]. Since we expect such designers to be inexperienced computer users we are using a declarative approach to style-sheet specification. Traditional style-sheet specification in publishing houses is declarative; therefore, designers will be on familiar ground. Surely, they would find macro programming, procedural specification and attribute manipulation nontrivial. Therefore, 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: we need provide only a mechanism that tests whether a specific context of a given part of a document is present or not. Based on this observation, the style rules may now incorporate conditional statements or expressions to express context-dependent choices. Thus, we can isolate context determination from style-rule syntax to a large extent. The preliminary work on Designer did just that [5, 9]. Once we have isolated the specification of context from the more general specification of style sheets, 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 similar structured documents are modeled by syntax trees or abstract syntax trees of a given grammar (an SGML DTD, an XML DTD or the HTML DTD) that generates the set of all such documents. For example, see Fig. 1.

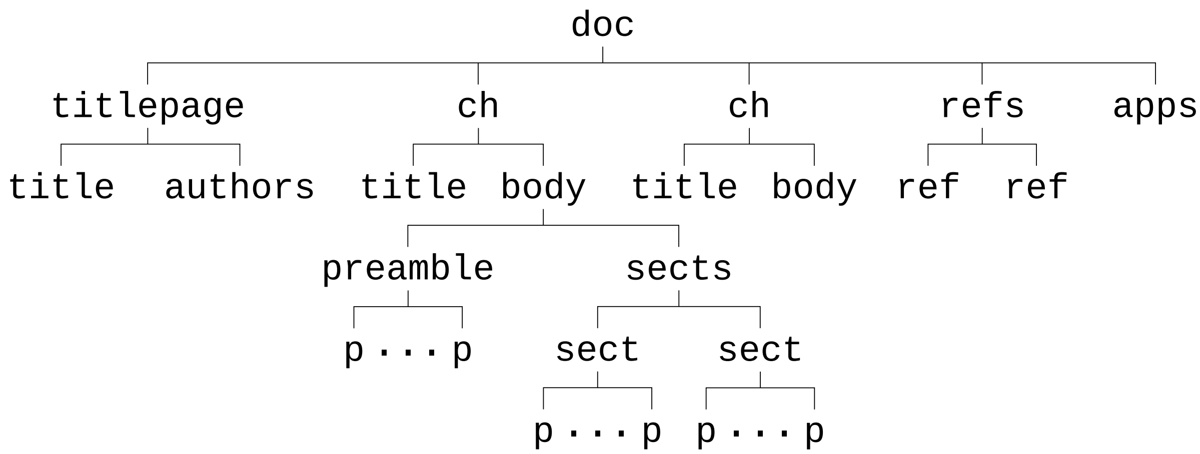


Figure 1: An example document tree.

The paper is organized as follows. We present, in Section 2, some examples of how context is and can be used in style-rule specification. In Section 3, we briefly discuss the different approaches to style-rule and contextual specification that have been and are being used. In Section 4, the 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 don't care 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 5, we introduce the kernel operations needed for a T-graph editor and we sketch their semantics and, in Section 6, we discuss the results and future directions.

# 2 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. We give some examples of designs that call for context-dependent processing:

1. In some designs, all paragraphs are indented, with the only exception being paragraphs immediately preceded by a heading.
2. The headings in an appendix might be labeled A), B), and C), whereas the headings in the main part are labeled 1., 2., and 3.
3. Three coauthors 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, list items are run-in instead of being placed on a new line.
5. Cross references are automatically prefixed with the name of the object to which they refer; that is, "see Theorem 1.2" and "see Lemma 1.1".

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. Its position, however, is in general not solely characterized by the element types higher up in the document hierarchy. Rather, it is often necessary to take the siblings (Examples 1 and 3) or the internal structure of the current element instance 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.

# 3 A review of context-dependent methods

Style-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 style sheet (or 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 logical markup tags with formatting commands that affect the typesetting directly. Programmable systems replace logical markup tags with programs that operate on the element they are applied to in an arbitrary manner; they may eventually also expand to direct formatting commands.

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

## 3.1 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. [13]. Macro packages such as LaTeX have some context dependencies hard-wired into them in an ad-hoc manner. Other systems such as Bravox [16] 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 [12]. Subordinate style rules, although more powerful than Scribe's parameter-inheritance scheme [26], are too weak to express all the context dependencies listed in Section 4. 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 style-sheet 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 style-sheet model** [22, 12, 28]. 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:

$$\begin{aligned} &\mathit{article} = (\mathit{paragraph}[\mathit{rule}\ 1] + \mathit{quotation})^*. \\ &\mathit{quotation} = (\mathit{paragraph}[\mathit{rule}\ 2])^*. \end{aligned}$$

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

$$\mathit{section} = \mathit{heading}\ \mathit{paragraph}^*.$$

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:

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

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 [11] and grammar-based coordination between attributed grammars [4, 27] 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 [21] is defined.

In principle, DESIGNER's style-sheet mechanism does not depend on document grammars. Therefore, it is legitimate to ask whether document grammars are redundant for design specifications. There are two answers: First, graphic artists profit from knowing the document grammar when they formulate context predicates, for, if a style sheet applies to only documents of restricted structure, then a context specification is in many cases easier and shorter. Second, it is an interesting research question whether document grammars can aid in the evaluation of context predicates.

## 3.2 Style-sheet languages and contextual dependence

We now briefly review the contextual methods available in some current style-sheet languages and approaches.

**CSS:** Cascading Style Sheets [18, 19] provide a simple style sheet mechanism that is oriented towards fast implementation for XML documents [3]. A CSS context expression, the link between a document and a CSS style sheet, 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 [20] is a style sheet language for XML [3] 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 [15] 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 [23] 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 [24] 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 set 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. [7] 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 [8].

# 4 Context, T-configurations and T-graphs

Document trees (or abstract syntax trees) have node labels from an element alphabet $\Sigma$. They satisfy the local matching requirement that the label of a parent node and the labels of its children conform to a production of the document grammar or of the DTD.

Since we view SGML, XML and grammatical documents as trees, element names (or nonterminals) are the node labels. The content of a document is represented as an external node or leaf of such a tree whose label is also in $\Sigma$ (in SGML the label is #PCDATA). Each SGML DTD (or document grammar) defines a set of trees. Note though that not every set of trees we discuss needs to be defined by an SGML DTD.

We assume that we can access a document tree using the basic navigational and testing 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. Conceptually, we assume that there are endmarker nodes $l$, $r$ and $t$ in each document tree that are the leftmost sibling of $v$, the rightmost sibling of $v$ and the topmost ancestor of $v$. They allow the user of the ADT to detect when a move falls off a document tree. We will 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. Once 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. 2. Note that the endmarker nodes belong to the T-configuration. A T-configuration $C_v$ contains the nodes that are most

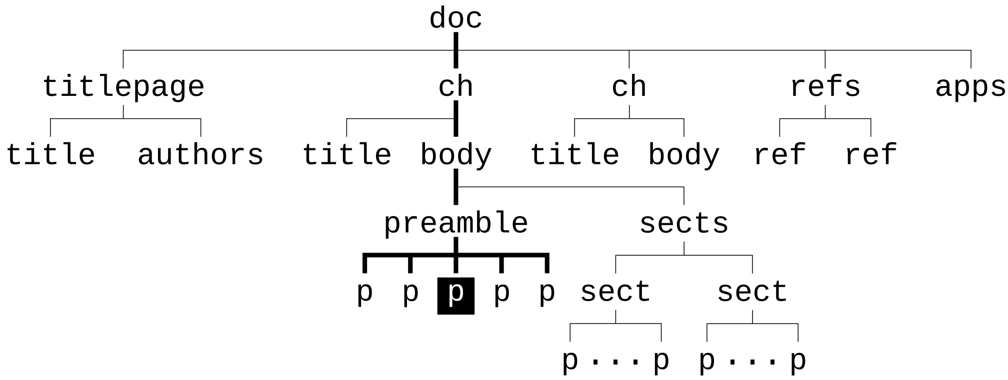


Figure 2: An example T-configuration in the document tree of Fig. 1.

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?

Given a node $v$ in a document tree, when we are processing $v$ with a context-dependent rule, we need to determine whether $v$ satisfies the corresponding context requirement. We translate this approach into an algorithmic framework by restricting where context requirements can occur—in the T-configuration of $v$—and by specifying context requirements generically using what we call **T-graphs.** If a T-graph matches a T-configuration, in a sense we make precise, then the T-configuration satisfies the context requirement specified by the T-graph. Examples of T-graphs are given in Fig. 3. Intuitively, we can think of T-graph matching as stretching the T-graph to anchor it at the contextual and endmarker nodes of the document tree. Then, we stretch the stems to locally match the nodes and edges in the document tree.

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

1. The T-configuration of a node in a document tree includes all nodes that are commonly used for context determination—the 90 percent solution.

2. A T-graph 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.

On the other hand, one obvious disadvantage of T-graphs is that they cannot capture all reasonable contexts. For example, they cannot handle the nested emphasis problem, even though only ancestors of the contextual node need be examined; see Fig. 4. Similarly,

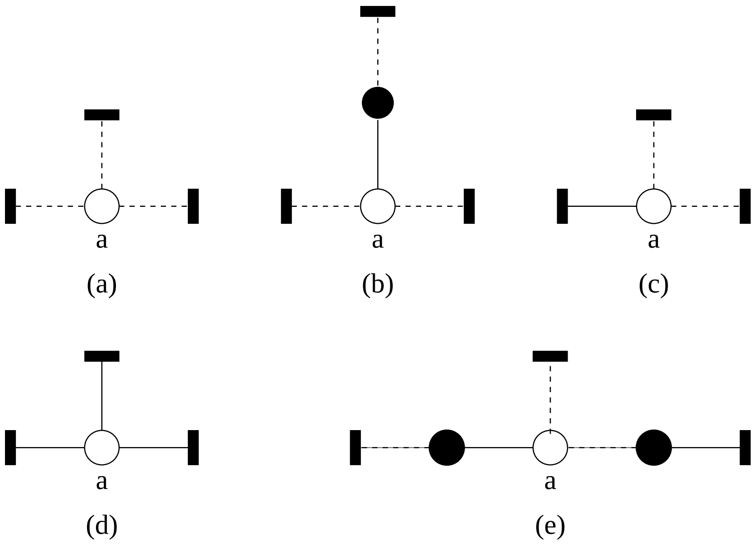


Figure 3: Example T-graphs. a. A T-graph that matches any contextual node. b. A T-graph that matches any contextual node that is not the root. c. A T-graph that matches a contextual node that is the leftmost sibling. d. A T-graph that matches only a contextual node that is the root. e. A T-graph that matches a contextual node that is neither the leftmost nor the rightmost sibling.

T-graphs cannot handle the two problem: A contextual node $v$ has exactly two ancestors labeled $a$. An obvious candidate for such a T-graph is given in Fig. 5 and it should also be clear that this T-graph will also match a contextual node $v$ that has three or more ancestors labeled $a$.

We define T-graphs more precisely as follows. A T-graph $G = (V, E, L, u)$ has a set $V$ of vertices such that $V$ contains four designated vertices $l$, $r$, $t$ and $u$, the **left vertex,** the **right vertex,** the **top vertex** and the **contextual vertex.** We use the terminological convention that "node" refers to document trees and T-configurations, and "vertex" refers to T-graphs. $G$ also has a set $E \subseteq V \times V$ of directed edges and a labeling function $L$. $G$ has three subdigraphs called **stems** (each of which is a line digraph) that exhaust $G$. The three stems have node $u$ in common; apart from $u$ they are mutually disjoint. The **up stem** is a digraph $(V_U, E_U)$, where $V_U \subseteq V$, $u, t \in V_U$ ($l, r \notin V_U$) and $E_U$ defines a line from $u$ to $t$. The **left stem** $(V_L, E_L)$ and **right stem** $(V_R, E_R)$ are defined in a similar manner. The labeling function $L$ labels the vertices, apart from the vertices $l$, $r$ and $t$, and edges. A vertex can be labeled either with a symbol from the document-tree alphabet $\Sigma$ (it matches only that symbol) or with $\bullet$ (it matches any symbol). An edge can be labeled either with 1 (it matches exactly one edge) or with + (it matches one or more edges).

Given a document tree $D$, a node $v$ in $D$ and a T-graph $G$, informally $G$ corresponds to the T-configuration $C_v$ determined by $v$ if there is an injection $\iota$ from the vertices of $G$ to the nodes of the T-configuration such that the contextual vertex of $G$ is mapped to $v$ and the endmarker vertices $l$, $r$ and $t$ of $G$ are mapped to the corresponding endmarker nodes of

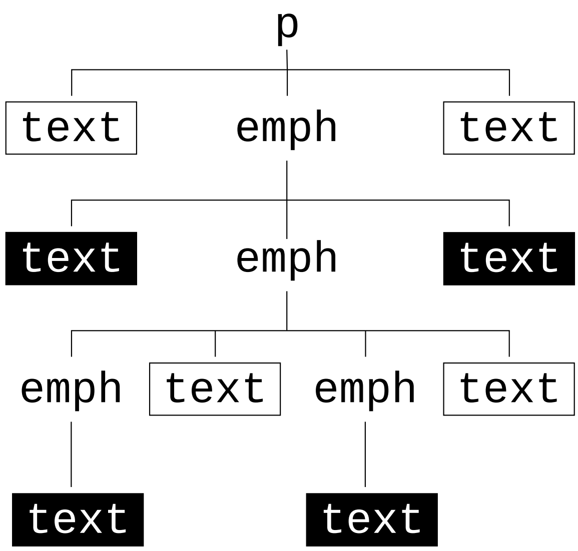


Figure 4: A document tree with nested emphasized text.

$C_v$. The injection should 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 $v$.
4. If $(x, y)$ is in $E_U$, then either $L((x, y)) = 1$ and $(\iota(y)$ is the parent of $\iota(x)$ or $L((x, y)) = +$ and $\iota(y)$ is an ancestor of $\iota(x)$
5. If $(x, y)$ is in $E_L$, then either $L((x, y)) = 1$ and $\iota(y)$ is the immediate left sibling of $\iota(x)$ or $L((x, y)) = +$ and $\iota(y)$ is a left sibling of $\iota(x)$.
6. If $(x, y)$ is in $E_R$, then either $L((x, y)) = 1$ and $\iota(y)$ is the immediate right sibling of $\iota(x)$ or $L((x, y)) = +$ and $\iota(y)$ is a right sibling of $\iota(x)$.
7. If $x$ is in $V$ and is not an endmarker vertex, then label$(\iota(x)) = L(x)$ if $L(x)$ is a document-tree symbol.

The basic algorithmic result is that T-graph matching can be carried out efficiently.

**Theorem 1** *T-graph matching can be implemented to run in time linear in the sum of the sizes of the T-graph and a T-configuration.*

Observe that we can solve the problem for each stem of the T-graph separately. We will discuss the matching problem for a right stem, the other two stems can be dealt with in a similar manner. We first break the stem into **rigid parts,** each of which is a substem

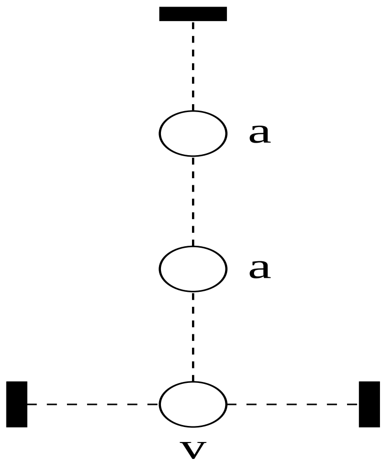


Figure 5: A candidate T-graph for the two problem.

that begins and ends with a vertex and has only rigid edges; that is, edges labeled 1. The idea is that we first try to match the rigid parts (which may or may not include don't-care vertices; that is, vertices labeled $\bullet$) one at a time beginning at the contextual vertex. The first rigid part must match the contextual node and a fixed number of nodes in this part of the stem. If it does not match, we know there is no T-graph match. Otherwise, we try to match the second rigid part at the earliest possible node after the last matched node. We continue this process as long as matching succeeds. The last part must match at the endmarker node; therefore, this match is also easy to carry out as the candidate matching nodes are predetermined.

But how do we match a rigid part that includes neither the contextual vertex nor the endmarker vertex? We use Pinter's algorithm [25] that takes into account the don't-care symbols. His algorithm can be implemented to run in time linear in the sum of the sizes of the right stem in the T-graph and the right stem in the T-configuration. We note that most contextual specifications are simple enough that it is easy to implement them to run in linear time without resorting to Pinter's algorithm.

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 by forming 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 problme has been studied in pattern matching when both the text and pattern have don't care symbols; however, the difference here is that we are not given a text, we are only given two "patterns". It is an interesting open problem whether we can answer this question efficiently.

## 5 Editing T-graphs

As we mentioned, 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 and then stretching 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 want to be able to change a vertex's label and an edge's label. In the case of a vertex, we want to be able to change its label to a different symbol including the don't-care symbol. In the case of an edge we want to be able to change a fixed edge to be a path, or a path to be a fixed edge. The last operations that we need are the insertion and deletion operations which we define as vertex operations that implicitly add and remove edges.

We specify the basic editing operations for T-graphs and also define their semantics.

**Vertex-label operations:**

1. $L(x) = a \;\rightarrow\; L(x) = b$.
2. $L(x) = \bullet \;\rightarrow\; L(x) = a$.
3. $L(x) = a \;\rightarrow\; L(x) = \bullet$.

**Edge-label operations:**

1. $L((x,y)) = 1 \;\rightarrow\; L((x,y)) = +$.
2. $L((x,y)) = + \;\rightarrow\; L((x,y)) = 1$.

Note that endmarker vertices have no labels; hence, they cannot be relabeled.

**Vertex operations:**

1. delete$(x) \;\rightarrow\; V = V - \{x\}$,
   $E = E - \{(w,x),(x,y)\} \cup \{(w,y)\}$ and $L((w,y)) = +$ if $x$ has exactly two adjacent edges (that is, if $x$ is neither an endmarker nor the contextual vertex).
2. $Insert(x,e) \;\rightarrow\; V = V \cup \{x\}$,
   $E = E - \{e\} \cup \{(w,x),(x,y)\}$ and $L((w,x)), L((x,y)) = L(e)$ where $e = (w,y)$.

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

## 6 Closing remarks

As always when implementing an idea, the implemented version of T-graphs in Designer differs somewhat from the T-graphs we have defined in this paper. The major difference is that the full T-graph editor has not been implemented. Despite this incompleteness we have demonstrated the viability of the T-graph approach. More important, from the graphics 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.

One obvious weakness with T-graphs is that we cannot use them to specify descendant contexts. One possible extension 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. The price we pay for such an extension is that it is more complex and, therefore, harder to understand and use as well as harder to implement.

A second weakness is that we cannot use T-graphs to specify contexts that cut across the document-tree structure such as required by the cross-reference problem (see context example 5 in Section 2). We wish to be able to automatically add the appropriate boilerplate text to a cross reference. Cross references are such a special case that perhaps they should be handled separately anyway.

We have proposed [7] 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 and to the right of a node. For example, if we want to capture 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.

What are our next steps?

We intend to enhance the creation of T-graphs by using inferential techniques and an example document tree. For example, we may wish to replace a rigid edge with a path if the context examples that we are using satisfy a single T-graph rather than two or more.

# References


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

[2] J. Bosak. Web page on DSSSL online. http://sunsite.unc.edu/pub/sun-info/standards/dsssl/dssslo/do960816.htm, 1997.

[3] T. Bray, J.-P. Paoh, and C.-M. Sperberg-McQueen. Extensible markup language XML 1.0. http://www.w3.org/TR/1998/REC-xml-19980210/, February 1998.

[4] 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.

[5] A. Brüggemann-Klein. Formal models in document processing. Habilitationsschrift. Faculty of Mathematics at the University of Freiburg, 1993.

[6] A. Brüggemann-Klein and S. Hermann. Design by Example: A user-centered approach to the specification of document layout. In F. Rowland and J. Meadows, editors, *Electronic Publishing '97: New Models and Opportunities. Proceedings of an ICCC/IFIP Conference held at the University of Kent at Canterbury, England, 14-16 April 1997*, pages 223–236. ICCC Press, 1997. URL: `http://www11.informatik.tu-muenchen.de/proj/designer/canterbury.ps`.

[7] 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*, Heidelberg, 1998. Springer-Verlag. To appear in Lecture Notes in Computer Science 1481.

[8] A. Brüggemann-Klein, M. Murata, and D. Wood. Regular Tree and regular hedge languages over unranked alphabets. Working paper, 1998.

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

[10] D. Connolly. W3C web page on XML. http://www.w3.org/XML/, 1997.

[11] 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.

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

[13] 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.

[14] ISO 8879: Information processing—Text and office systems—Standard Generalized Markup Language (SGML), October 1986. International Organization for Standardization.

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

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

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

[18] H. Lie. W3C web page on Cascading Style Sheets. http://www.w3.org/Style/css/, 1997.

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

[20] C. Lilley. Extensible style language XSL. http://www.w3.org/Style/XSL/, September 1998.

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

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

[23] 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.

[24] 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.

[25] R.Y. Pinter. Efficient string matching with don't-care patterns. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume F 12 of *Computer and System Sciences*, pages 11–29. Springer-Verlag, New York, NY, 1985.

[26] 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.

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

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