Caterpillars: A Context Specification Technique*

Anne Brüggemann-Klein† Derick Wood‡

January 14, 2000

Abstract

We present a novel, yet simple, technique for the specification of context in structured documents that we call caterpillar expressions. Although we are primarily applying this technique in the specification of context-dependent style sheets for HTML, SGML and XML documents, it can also be used for query specification for structured documents, as we shall demonstrate, and for the specification of computer program transformations.

From a conceptual point of view, structured documents are trees, and one of the oldest and best-established techniques to process trees and, hence, structured documents are tree automata. We present a number of theoretical results that allow us to compare the expressive power of caterpillar expressions and caterpillar automata, their companions, to the expressive power of tree automata. In particular, we demonstrate that each caterpillar expression describes a regular tree language that is, hence, recognizable by a tree automaton.

Finally, we employ caterpillar expressions for tree pattern matching. We demonstrate that caterpillar automata are able to solve tree-pattern-matching problems for some, but not all, types of tree inclusion that Kilpeläinen investigated in his PhD thesis. In simulating tree pattern matching with caterpillar automata, we reprove some of Kilpeläinen's results in a uniform framework.

*The work of both authors was supported partially by a joint German-HK grant. In addition, the work of the second author was supported under a grant from the Research Grants Council of Hong Kong. Earlier and shorter versions of some of the ideas in this paper were presented at PODDP '98 [6] and at DLT '99 [10].

†Institut für Informatik, Technische Universität München, Arcisstr. 21, 80290 München, Germany. E-mail: brueggem@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.

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. In the Designer project (a typesetting project) that we have been working on for a number of years, we were faced with the problem of the specification of context-dependent style rules for structured documents. At first, we expected to use the traditional approach from the compiler and programming-languages community: attribution [3, 22, 33]. In addition, we also expected to be able to adopt and modify previous approaches to style specification such as suggested by the DSSSL [18] and XSL [26] documents and by Lie's work [24, 25] for SGML [17] and XML [12] style specification. Alternatively, we considered Munson's approach in the Proteus system [31] and Murata's more general approach [32] which is based on tree automata. Murata's approach is the closest technique to ours, although more traditional and quite different. But, we were faced with an additional constraint that changed our thinking. We wanted to provide a system that graphic designers could easily use to specify style rules and style sheets [5]. The point is that such designers would be, to a large extent, computer naive. They would almost certainly find the manipulation of attributes difficult. Therefore, we decided to separate, somewhat, the specification of context from the more general issue of style specification. One observation about this separation is in order: we need provide a mechanism that tests only 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 [4, 9]. We have also investigated style-sheet specification for tables [37] and some basic decidability questions for style sheets [8].

Before discussing further the contextual specification technique that we introduce and study, we make some comments on contextual style rules and provide some typical examples. General design rules suggest that the elements of a 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 designers 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 in Fig. 1. In the five examples described in Fig. 1, 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 as it happens to be the case in Examples 2 and 4. 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

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".

Figure 1 Examples of context.

the structure of the element it refers to (Example 5).

The contextual technique we introduce is also applicable to the compilation of computer programs, but has little appeal since compiler designers and writers do not usually allow users to modify a compiler according to new context dependencies. Our techniques may, however, be used when developing code optimizers or other program transformation tools since, in both cases, there may be a number of individuals collaborating on the development [30].

Once we have isolated the specification of contexts from the more general specification of style sheets, we are able to provide naive 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. Since regular expressions are understood by many people who are not programmers *per se,* and they are a simple specification technique, we decided to use them for context specification. There is a body of somewhat related work, which we discuss in Section 2, in which a similar decision was made.

We make the well-accepted assumption that a set of similar documents are modeled by syntax trees or abstract syntax trees of a given grammar (an SGML document grammar, an XML document grammar or HTML) that generates the set of all such documents. From now on we will no longer mention SGML and HTML but restrict ourselves to XML and XML document grammars. Indeed, for this paper it is irrelevant which specific grammar mechanism is used to define classes of documents.

We introduce and motivate, in Sections 2 and 3, the notions of caterpillars and context and establish a basic complexity result for the evaluation of caterpillar automata on document trees. In Section 4, we investigate the expressive power of caterpillar automata in comparison with tree automata. In particular, we demonstrate that each caterpillar expression describes a regular tree language that is, hence, recognizable by a tree automaton. Finally, in Section 5, we demonstrate that caterpillar automata can be used to solve tree-pattern-matching problems, before we conclude with some open questions and general remarks.

2 Caterpillars and context

It is natural to write of a context of a node ν in a document tree; for example, if ν is a paragraph, we may wish to determine whether ν is the first paragraph of a chapter or of a section. In this setting, the property "is the first paragraph of a chapter or of a section" is the context, and each first paragraph of a chapter or section satisfies it. Hand in hand with this intuitive notion is the notion of a *context set* that consists of all first paragraphs in chapters and sections of a document, the context set "first paragraph." Thus, for a specific abstract syntax tree, the set of first-paragraph-in-a-chapter-or-in-a-section nodes is the context set "first paragraph." In other words, for a given tree t, each set S of nodes of t may be a context set in the sense that the nodes in S are all the nodes in t that, intuitively, have a specific context.

Since a real caterpillar crawls around a tree, we define a *(contextual) caterpillar* as a sequence of atomic movements and atomic tests. A caterpillar can move from the current node to its parent, to its left sibling, to its right sibling, to its first child or to its last child. To prevent a caterpillar from dropping off a tree it is allowed to test whether it is at a leaf (external node), at the root, at the first sibling or at the last sibling. Finally, a caterpillar can read the label at the current node of the document tree. Note that these navigational and testing operations define an abstract data type for trees; however, caterpillars are more than an abstract data type as they capture a specific sequence of abstract-data-type operations.

For example, given the partial document tree in Fig. 2, a first-paragraph-in-chapter caterpillar is

$$p \ isFirst \ up \ up \ up \ ch,$$

where a node label is an implicit test on the current node, and a first-paragraph-in-section caterpillar is

$$p \ isFirst \ up \ sect.$$

Similarly, a last-section caterpillar is

$$sect \ isLast.$$

In each of these examples, if we place the caterpillar at any node in a given tree, it crawls and evaluates until it has executed the sequence successfully, it cannot carry out the

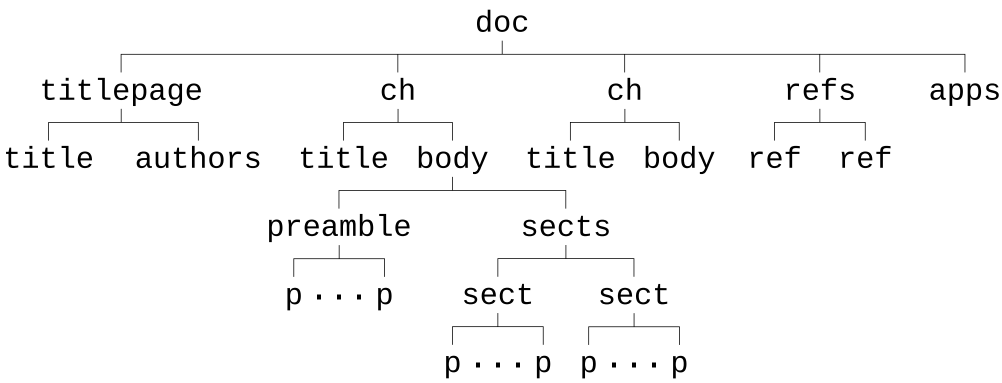

Figure 2 An example document tree.

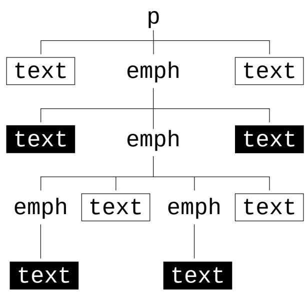

Figure 3 A document tree with nested emphasized text.

next move, or it has obtained a false evaluation of a test. This notion of a caterpillar's evaluation leads to a set of nodes in a tree that are the context set of the caterpillar, as we will make precise in the next section.

In general, we want to be able to specify a given context for all trees of a given XML DTD (document type definition aka document grammar) and we may not be able to do so with a single caterpillar. For example, consider document trees in which emphasized text can be nested (as it can be in LaTeX); see Fig. 3. Typically, we emphasize text in a roman environment by setting it in italic whereas we emphasize text in an italic environment by setting it in roman. Thus, we need to determine the parity of the nesting to be able to specify the appropriate typeface. The caterpillar

text up p

and the caterpillar

text up emph up emph up p

specify contexts with even parity whereas the caterpillars

> *text up emph up p*

and

> *text up emph up emph up emph up p*

specify odd-parity contexts in the tree of Fig. 3. We need, however, to specify caterpillars of any length as the depth of emphasis nesting is not bounded, even though it is finite. Our solution is simple, yet powerful.

Rather than allowing only a finite number of caterpillars for each context, we allow infinite sets of caterpillars. Since we can consider a caterpillar to be a string over the set of positional tests, movements, and nonterminals (or elements) of the grammars, a set of caterpillars is a language in the usual language-theoretic sense.

We use regular expressions, *caterpillar expressions,* to specify such languages and we use finite-state automata, *caterpillar automata,* to model their execution. For example, we can specify all even-parity-emphasis contexts for trees of the form given in Fig. 3 with the caterpillar expression

$$\textit{text}\ [\textit{up emph up emph}]^*\ \textit{up p}$$

and all odd-parity-emphasis contexts with

$$\textit{text up emph}\ [\textit{up emph up emph}]^*\ \textit{up p}.$$

Our method of specifying contexts singles out those nodes in the trees of a grammar for which the execution of one of the caterpillars in the language of a given caterpillar expression succeeds when it is started on these nodes. One immediate implication of this model is that we must separately specify even-parity contexts and odd-parity contexts since we need to specify different actions in each case.

The novelty of our approach is that we use strings of node labels, tests and navigational operations rather than strings of only node labels. Readers of drafts of this paper have pointed out earlier work that uses caterpillar-like ideas. The first and larger body of work uses the idea of a regular expression, a *path expression,* to determine a set of paths in a labeled graph. Mendelzon's research [13, 27] has shown, with his graph-theoretic query language G++ for databases, that this approach is not only powerful but can also be visualized well [13]. (In Mendelzon's project, users provide restrictions of a graph-theoretic view of a database by graph-theoretic means. This visual process provides, essentially, a subgraph as the query.) More recently, Abiteboul and his coworkers [2] have used similar ideas to specify paths in graphs for semi-structured document queries.

COSY [23], a language for the specification of concurrent processes developed by Lauer and his codesigners, also uses path expressions as its central specification tool. In all cases, path expressions do not include any explicit navigational operations; they are very similar to standard regular expressions.

The second body of related work is the programming language developed in the Logo Group of the MIT Artificial Intelligence Laboratory [1]. It was designed to explore mathematics by students who ranged in age from preschool to postdoctoral. One of the cute aspects of the language is the use of turtle geometry to investigate geometrical notions. It was called turtle geometry because a user manipulated a "turtle" (a representation of the cursor's current position) with a simple command-based language. In this language, programs were sequences of movements and actions.

Last, Klarlund and Schwartzbach [21] proposed a new approach to recursive data structure specification that allows the resulting structures to be graph-like. Essentially, their work is closest in spirit to ours, although the domain and usage are very different. The central idea in their work is that recursive tree structures are enhanced by using additional routing expressions. The expressions add extra edges (or links) to a tree-structure instance when it is instantiated. The routing expressions also include navigational and testing operations.

3 Evaluating caterpillar expressions

We formally define caterpillars as well as caterpillar expressions and their languages. We also demonstrate that we can model the execution of caterpillar expressions with finite-state automata. Merk [28] has investigated some of the basic properties of caterpillar expressions.

Document trees (or abstract syntax trees) have node labels from an alphabet Σ. Since we view 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 Σ. Each XML DTD (or document grammar) defines a set of trees T. Not every set of trees we are discussing needs to be defined by an XML DTD though. Given a set T, a context mapping $\mathcal{C}$ for T maps any tree t in T to a subset $\mathcal{C}(t)$ of $nodes(t)$. Note that $\mathcal{C}(t)$ forms a context set in the sense of the previous chapter. Hence, a context mapping $\mathcal{C}$ identifies which nodes of a tree satisfy the context.

Let Δ denote the alphabet of moves and tests; that is,

$$\Delta = \{up, left, right, first, last, isFirst, isLast, isLeaf, isRoot\}.$$

A string x over $\Sigma \cup \Delta$ is a caterpillar and it denotes a context mapping $\mathcal{C}_x$ for the set of Σ-labeled trees as follows. For any tree t and any node ν of t:

- if $x = \lambda$, then ν belongs to $\mathcal{C}_\lambda(t)$.
- if $x = aw$, where $a \in \Sigma \cup \Delta$ and $w \in (\Sigma \cup \Delta)^*$, then ν belongs to $\mathcal{C}_{aw}(t)$ if and only if one of the following conditions holds:
  1. a is in Σ, ν has label a, and ν belongs to $\mathcal{C}_w$(t).
  2. $a = up$, ν has a parent ν' in t, and ν' belongs to $\mathcal{C}_w(t)$.
  3. $a = left$, ν has a direct left sibling ν' in t, and ν' belongs to $\mathcal{C}_w(t)$.
  4. $a = right$, ν has a direct right sibling ν' in t, and ν' belongs to $\mathcal{C}_w(t)$.
  5. $a = first$, ν has children in t, ν's leftmost child is ν', and ν' belongs to $\mathcal{C}_w(t)$.
  6. $a = last$, ν has children in t, ν's rightmost child is ν', and ν' belongs to $\mathcal{C}_w(t)$.
  7. $a = isFirst$, ν is the leftmost node among its siblings in t, and ν belongs to $\mathcal{C}_w(t)$.
  8. $a = isLast$, ν is the rightmost node among its siblings in t, and ν belongs to $\mathcal{C}_w(t)$.
  9. $a = isLeaf$, ν is an external node in t, and ν belongs to $\mathcal{C}_w(t)$.
  10. $a = isRoot$, ν is the root node of t, and ν belongs to $\mathcal{C}_w(t)$.

Note that this formal definition of the meaning of a caterpillar corresponds to the informal notion we used in the previous section.

We can extend the context mapping of a caterpillar to the context mapping of a language L over the alphabet $\Sigma \cup \Delta$ in the usual way; namely,

$$\mathcal{C}_L(t) = \bigcup_{x \in L} \mathcal{C}_x(t).$$

Hence, a node ν of a tree t satisfies the context denoted by a language L if, starting from ν, it is possible to perform at least one sequence x in L of moves and tests in t. Note that, for each context mapping $\mathcal{C}$ for a set of trees T, there is a language L over $\Sigma \cup \Delta$ such that $\mathcal{C} = \mathcal{C}_L$.

We call $\Sigma \cup \Delta$ the *caterpillar alphabet* of Σ. A regular expression and a finite-state automaton over a caterpillar alphabet are called a caterpillar expression and a caterpillar automaton, respectively. We now restrict our attention to regular languages over a caterpillar alphabet and to context mappings defined by such languages. We can define how caterpillar automata operate on trees by defining sequences of configurations in the standard manner. A configuration, in this case, consists of a node of a tree (the current node), a state of the automaton (the current state), and a string x over the caterpillar alphabet (the remaining input string). We are now in a position to state and prove a basic time-complexity result for the computation of a caterpillar automaton on a given tree.

Theorem A *Given a tree t with m nodes and a caterpillar automaton M with n transitions, we can compute the set of nodes in t that are in the context set $\mathcal{C}_{L(M)}(t)$ in worst-case* *time* $O(m \times n)$.

PROOF We assume without loss of generality that the automaton M has at most as many states than it has transitions.

Our goal is to compute the set R of all pairs (ν, s) such that the automaton M with s as its initial state has an accepting computation on t when starting from node ν. We can then determine the nodes in the context set $\mathcal{C}_{L(M)}(t)$ by reporting all pairs (ν, s_I) in R, where s_I is M's initial state.

The set R can be computed as a transitive closure in a graph G: Let the vertices of G be the pairs (ν, s) of t-nodes ν and M-states s. Let there be an edge from (ν, s) to (ν', s') in G if and only if M, starting from node ν' in state s', can move to node ν and into state s. More precisely, for each transition $s' \xrightarrow{a} s$ of M,

1. If a is a move in Δ and M can move from ν' to ν using move a, then G has an edge leading from (ν, s) to (ν', s'). (Note that when a is *up*, all children of ν satisfy the condition.)
2. If a is a test in Δ or a symbol in Σ and the node ν satisfies the test a or has the label a, then G has an edge leading from (ν, s) to (ν', s').

Each path in G from (ν, s) to (ν', s') reverses a computation of M, starting on node ν' in state s' and leading to node ν in state s. Hence, R is the closure of the set of vertices (ν, s), s being a final state of M, in G.

The graph G has at most $m \times n$ edges, so the transitive closure R and, hence, the context set $\mathcal{C}_{L(M)}(t)$, can be computed in worst-case time $O(m \times n)$. $\square$

4 Caterpillar-regular and regular tree languages

Finite-state (string) automata are a well-established model for regular languages of strings, or regular string languages, as we will call them from now on. In analogy, finite-state tree automata have been investigated since the early sixties to model regular languages of trees or regular tree languages. We now introduce tree languages that are recognized by caterpillar automata and investigate how they relate to regular tree languages.

A tree t is *recognizable* by a caterpillar automaton M if and only if its root is in the context set $\mathcal{C}_{L(M)}(t)$. In other words, if M has at least one accepting computation on t when starting at t's root in the initial state. The tree language $\mathrm{T}(M)$ that is *recognizable* by a caterpillar automaton M is the set of trees that are recognizable by M. A tree language is *caterpillar regular* if it is recognizable by some caterpillar automaton.

The main result in this section is that each caterpillar-regular tree language is also regular

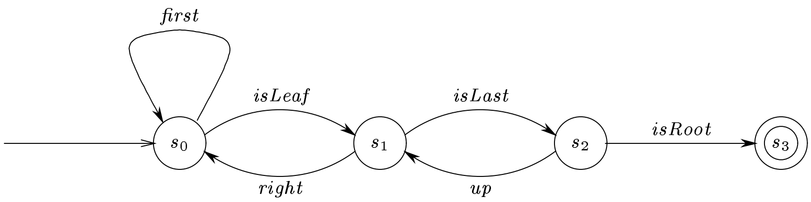

Figure 4 The tree-traversing caterpillar automaton M_T.

(Theorem F). The proof requires a new characterization of regular tree languages in terms of congruences and local views [7].

It is an open question whether regular tree languages are also always caterpillar regular. We do not believe so, but at least we provide evidence in this section that the caterpillar-regular languages form a rich language class. We demonstrate that the finite tree languages, the so-called local tree languages and the so-called path-closed tree languages all form proper subclasses of the caterpillar-regular languages and that the caterpillar-regular tree languages are closed under union (Theorems B–E).

The core part of some of the constructions in this section is the tree-traversing caterpillar automaton M_T defined in Fig. 4. When started at the root of a tree, M_T carries out a depth-first traversal of the tree and terminates at the root in the final state.

The depth-first tree traversal has three phases which continuously alternate: going down, going right, and going up. The automaton goes down until it reaches a leaf. Then, it goes right. A right-going phase is interrupted as soon as it reaches an internal node and, hence, a downward move is possible; at the point of interruption, a new down-going phase starts. A right-going phase ends when the automaton cannot move to the right, in which case, the automaton goes up. The up-going phase ends as soon as the automaton can move to the right. At this point, the automaton starts or resumes a right-going phase. Fig. 5 illustrates a state trace of M_T on an example tree.

The tree-traversing automaton M_T is deterministic in a very strong sense, which we call strongly deterministic: In any state and at any node of the tree, M_T executes at most one transition. For example, if M_T is in state s_0, it can either move down or execute an *isLeaf* transition, depending on whether M_T is at an internal or an external node, respectively. The next transition is in each case unambiguously determined by the position of the current node in the tree. Hence, when starting from t's root in state s_0, M_T has exactly one computation that cannot be continued any further.

Lemma 4.1 *The unique computation of M_T on a tree t that cannot be continued any further has the following properties:*

1. *M_T visits each node of t at least twice, the first time in state s_0 and the second time in state s_1.*

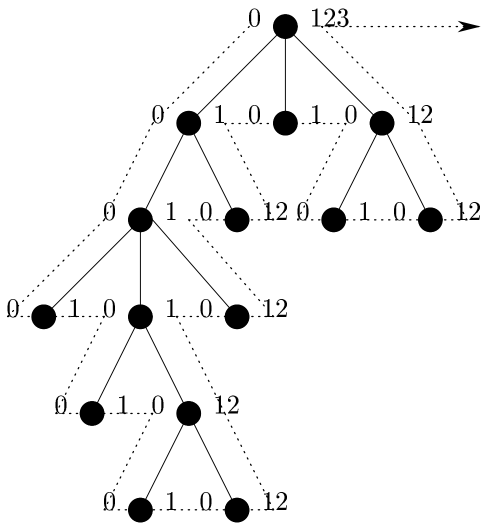

Figure 5 The state trace of M_T on an example tree.

2. *Between the first two visits of each node ν, M_T visits other nodes only in ν's subtrees.*
3. *After visiting a node ν the second time, M_T immediately visits ν again in state s_2 if ν has no right sibling, and then again in the final state if ν is the root. Apart from these additional visits, neither ν nor any of its descendants are ever visited again.*

It is straightforward to prove the three claims of Lemma 4.1 by induction on the tree structure. It follows that M_T does indeed perform a depth-first tree traversal and it terminates at the tree's root in the final state.

Note that M_T visits each node of a tree exactly once in state s_0 and these visits are in preorder. Hence, we can (and will) enhance M_T with other caterpillar automata that carry out some local testing at each node.

We now establish that caterpillar automata are able to recognize an important subset of the regular tree languages known as *tree-local tree languages*. Takahashi [35] has demonstrated the cetral rôle of tree-local languages in the theory of regular tree languages. Since the sets of syntax trees of (extended) context-free grammars are tree local, many document-grammar mechanisms, XML's among others, define tree languages that are tree local. Hence, document tree languages are tree local. After some preliminaries, we prove, in Theorem B, that caterpillar automata are an appropriate mechanism to check whether document instances conform to a given grammar.

A *tree grammar* G over an alphabet Σ is a tuple (Σ, P, I), where P is a subset of $\Sigma \times \Sigma^*$; for each a in Σ, the set $\{w \mid a \longrightarrow w \text{ is in } P\}$ is a regular string language; and I is a nonempty subset of Σ. We refer to P as the set of productions and I as the set of sentence symbols. A tree grammar is similar to an extended context-free grammar with just two differences.

First, a tree grammar can have more than one sentence symbol and, second, there is no distinction between terminal and nonterminal symbols.

The *derivation trees* of a tree grammar $G = (P, I)$ with root label a, $a \in \Sigma$, are defined inductively: For each production $a \longrightarrow w$, $w = a_1 \cdots a_n$, $n \geq 0$, and for any derivation trees t_1 with root label $a_1, \ldots, t_n$ with root label a_n, the tree $a(t_1, \ldots, t_n)$ is a derivation tree of G with root label a. A derivation tree of G is a derivation tree with a root label in I. Note that a-labeled leaves in a derivation tree correspond to productions $a \longrightarrow \epsilon$ in the grammar and only labels that have such a production can label a leaf.

A tree language is *tree local* if and only if it is the set of derivation trees for a tree grammar.

Theorem B *Every tree-local language is caterpillar regular.*

PROOF Letting $G = (\Sigma, P, I)$ be a tree grammar, we construct a caterpillar automaton that checks whether a tree t is a derivation tree of G.

Our approach is to enhance the tree-traversing caterpillar automaton M_T of Fig. 4 to also check, for each node ν in t, whether the sequence of the labels of ν's children conforms to the rules of the grammar G. The new automaton checks for local conformance of each node before it visits it for the first time in state s_0 of M_T.

The enhancement involves adding to M_T new caterpillar automata, one for each symbol in Σ. For each a in Σ, there is, by the definition of tree grammars, a finite-state string automaton M_a that recognizes $\{w \,|\, a \longrightarrow w \text{ is in } P\}$. We first convert M_a into a caterpillar automaton M'_a as follows. Initially, M'_a has the same states as M_a and has no transitions. Then, for each transition (p, b, q) in M_a, add a new state r to M'_a and add two new transitions (p, b, r) and $(r, right, q)$ to M'_a.

Second, we construct the caterpillar automaton of Fig. 6 that combines all the caterpillar automata M'_a and includes additional testing and tidying up after a local check has terminated successfully.

Third, we join M_T and the automaton of Fig. 6 at state s_0. The state s'_0 is the new initial state. The dotted transition from s_a to s_0 on *isLeaf* is present only if M_a recognizes the empty string. Note that the *isLast* transition leaving M_a in Fig. 6 denotes a collection of *isLast* transitions, one for each final state of M_a. There is also a copy of s_a and M_a in the enhanced automaton for each a in Σ. Finally, we redirect the two transitions on *first* and *right* of M_T into the new start state s'_0.

As a result, the enhanced automaton does everything the tree-traversing automaton M_T does, and in addition it checks before it enters a node of the tree in state s_0 whether the labels of that node and its children conform to grammar G. Since M_T visits each node exactly once in state s_0, the enhanced automaton checks the whole tree for conformance with G.

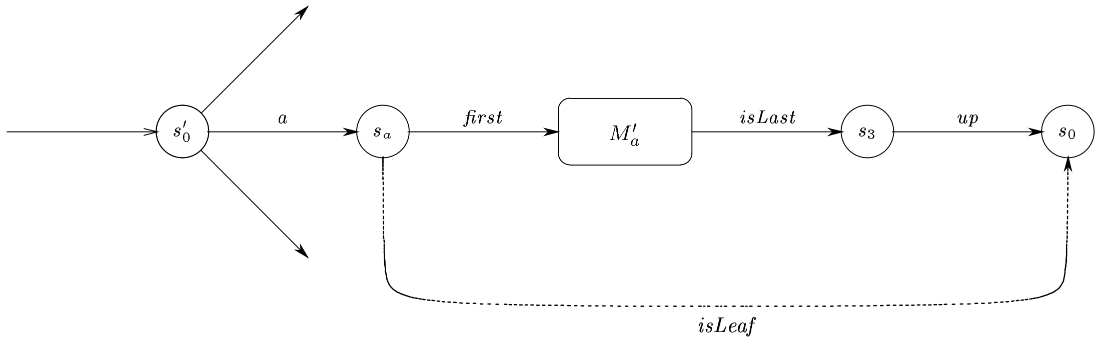

Figure 6 Testing for local grammar conformance.

To complete the construction of the enhanced automaton we still have to ensure that the root label of the given tree is in the set of G's sentence symbols. For this purpose we add another state to the enhanced automaton, make this new state the sole final state, and add a transition from the original final state of M_T to its new final state on each sentence symbol. □

We establish that tree-local languages form a proper subfamily of the caterpillar-regular languages in two steps. First, we prove that every finite tree language is caterpillar regular and, second, we exhibit a finite tree language that is not tree local.

Theorem C *First, the family of caterpillar-regular tree languages is closed under union. Second, every finite tree language is caterpillar regular. Third, every co-finite tree language is caterpillar regular.*

PROOF First, given two finite-state caterpillar automata M_1 and M_2, we define their "sum" $M_1 + M_2$ as usual by building the disjoint union of the state sets and the transition sets of M_1 and M_2 and then adding a new initial and a new final state and empty-string transitions from the new initial state to each of the initial states of M_1 and M_2 and from each of the final states of M_1 and M_2 to the new final state. Obviously, $\mathrm{L}(M_1 + M_2) = \mathrm{L}(M_1) \cup \mathrm{L}(M_2)$. Hence, $\mathrm{T}(M_1 + M_2) = \mathrm{T}(M_1) \cup \mathrm{T}(M_2)$.

Second, for each tree t, we can construct a caterpillar that recognizes t and only t. For example, a caterpillar for the tree $a(ba)$ is

$$a \, \mathit{first} \, b \, \mathit{isLeaf} \, \mathit{right} \, a \, \mathit{isLeaf} \, \mathit{isLast}.$$

Since the family of caterpillar-regular tree languages is closed under union, all finite tree languages are caterpillar regular.

Third, starting from a caterpillar x that recognizes the tree t and no other tree, we construct a strongly-deterministic caterpillar automaton M_x that recognizes any tree but

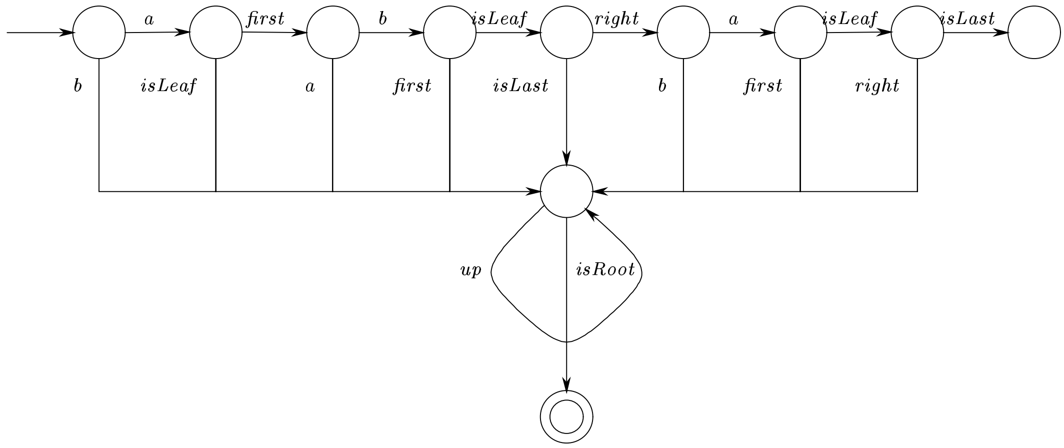

Figure 7 Recognizing trees different from a(ba).

t_x and ends each computation where it began, namely at the root of the tree it attempts to recognize. Fig. 7 demonstrates the automaton M_x for the caterpillar

$$x = a\,\mathit{first}\,b\,\mathit{isLeaf}\ \mathit{right}\,a\,\mathit{isLeaf}\ \mathit{isLast},$$

which recognizes $t = a(ba)$. The automaton M_x looks for a reason why x might fail; only if it finds one does it return to the root in an accepting state.

Given a number of trees $t_1, \dots, t_n$ and caterpillars $x_1, \dots, x_n$ that recognize the singleton tree languages $\{t_1\}, \dots, \{t_1, \dots, t_n\}$, respectively, we catenate $M_{x_1}, \dots, M_{x_n}$. The resulting automaton recognizes the complement of the tree language $\{t_1, \dots, t_n\}$. □

Theorem D *There are caterpillar-regular tree languages that are not tree local.*

Proof The tree language $L = \{a(a)\}$ is finite and, hence, is caterpillar regular, but it is not tree local. If L were tree local, then the productions $a \to a$ and $a \to \lambda$ must be in its tree grammar. Immediately, the trees a and $a(a(a))$, for example, must also be in L, which is not the case; hence, we have obtained a contradiction. □

Given a string language L, we define the tree language $\mathrm{PL}(L)$, the *path-closed tree language of* L, as follows: A tree t is in $\mathrm{PL}(L)$ if and only if the labels of each root-to-leaf path in t spell out a string in L. Our interest in such languages is that they shed light on the power of caterpillars as we see in Theorem E. The tree languages $\mathrm{PL}(L)$ are similar to, but different from, the branch-tree languages of Courcelle [14]. The difference is that in Courcelle's work each path in a tree is encoded by both its label string and its branching string (the indexes of the specific children on the path). As a result, the label and branching strings of a tree characterize the tree uniquely which is not the case when we have only label strings.

Theorem E *If L is a regular string language, then its path-closed tree language* $\mathrm{PL}(L)$ *is a caterpillar-regular tree language.*

Proof The idea behind the proof is that a regular string language L is recognized by some finite-state automaton M. We now construct from M a caterpillar automaton $\mathrm{PL}(M)$ such that a tree t is recognized by $\mathrm{PL}(M)$ if and only if $t \in \mathrm{PL}(L)$. The backbone of $\mathrm{PL}(M)$ is the tree-traversing caterpillar automaton M_T of Fig. 4.

Let M be a nondeterministic finite-state automaton for L that has no null-string transitions and has initial state s_I. Furthermore, we also assume, without loss of generality, that M is reverse deterministic; that is, for each symbol a in Σ and each state s' of M there is at most one state s in S such that $s \xrightarrow[M]{a} s'$. Such an automaton can be constructed from a deterministic automaton for the reverse of language L.

The caterpillar automaton $\mathrm{PL}(M)$ for $\mathrm{PL}(L)$ has states that are pairs of states as in the standard cross-product of two automata (here the two automata are M and M_T) but we also need some additional states as we shall see. The paired states of $\mathrm{PL}(M)$ are pairs (s_i, S), where s_i, $0 \leq i \leq 2$, is a state of M_T and S is a set of states of M.

While traversing a tree t, we ensure that the first component of $\mathrm{PL}(M)$'s state is M_T's state at the same point of the traversal and that the second component consists of all the states M can be in after processing the labels of the nodes on the path from the root to the current node, including the label of the current node.

Since M is reverse deterministic, given the label a of the current node and the second component of $\mathrm{PL}(M)$'s state, we can compute the second component of $\mathrm{PL}(M)$'s state at the current node's parent.

In addition to the paired states, $\mathrm{PL}(M)$ has an initial state $()$, a final state $[]$ and some auxiliary states. The auxiliary states are named $A(S)$, $A_1(a, S)$, $A_2(a, S)$, and $A_3(a, S)$, where $a \in \Sigma$ and S is a set of states of M.

Here are $\mathrm{PL}(M)$'s transitions:

$$() \xrightarrow{a} (s_0, S),\ \ S = \{s \mid s_I \xrightarrow[M]{a} s\}$$

$$(s_0, S) \xrightarrow{\mathit{first}} A(S) \xrightarrow{a} (s_0, S'),\ \ S' = \{s' \mid s \xrightarrow[M]{a} s',\ \ s \in S\}$$

$$(s_0, S) \xrightarrow{\mathit{isLeaf}} (s_1, S) \text{ if } S \text{ contains a final state of } M$$

$$(s_1, S) \xrightarrow{a} A_1(a, S) \xrightarrow{right} A_2(a, S) \xrightarrow{b} (s_0, S'),$$

$$S' = \{s' \mid \text{there are states } s \text{ and } s'' \text{ of } M \text{ such that } s'' \in S,\ s \underset{M}{\xrightarrow{a}} s'',\ s \underset{M}{\xrightarrow{b}} s'\}$$

$$(s_1, S) \xrightarrow{isLast} (s_2, S)$$

$$(s_2, S) \xrightarrow{a} A_3(a, S) \xrightarrow{up} (s_1, S'),\ S' = \{s' \mid s' \underset{M}{\xrightarrow{a}} s,\ s \in S\}$$

$$(s_2, S) \xrightarrow{isRoot} [\,]$$

We can verify from the transitions that a computation of PL(M) does indeed have the properties we have claimed. Furthermore, PL(M) can successfully complete a computation only if each label sequence of a the root-to-leaf path is recognized by M. Hence, PL(M) recognizes PL(L). □

We now compare caterpillar automata to tree automata and the tree languages they recognize, namely regular tree languages. We define tree automata and regular tree languages using the approach of Thatcher [36]. The key point about his approach is that the node labels are not ranked; any node label may label any node in a tree independently of the number of children that node has. We refer to our synopsis [7] on the state of the art on tree automata and regular tree languages over unranked alphabets; the more widely known literature on tree automata [11, 15, 16] addresses primarily the ranked-alphabet case.

We have characterized [7] the regular tree languages using congruences and local views. We now summarize the pertinent definitions and results before proving the following main result.

Theorem F *Every caterpillar-regular tree language is a regular tree language.*

A *pointed tree* (sometimes also called a tree with a handle or a handled tree) is a tree over an extended alphabet $\Sigma \cup \{X\}$ such that precisely one node is labeled with the variable X and that node is a leaf.

If t is a pointed tree and t' is a (pointed or nonpointed) tree, we can catenate t and t' by replacing the node labeled X in t with the root of t'. The result is the (pointed or nonpointed) tree tt'.

Let T be a tree language. Trees t_1 and t_2 are *top congruent* with respect to T ($t_1 \sim_T t_2$ or simply $t_1 \sim t_2$) if and only if for each pointed tree t the following condition holds:

$tt_1 \in T$ if and only if $tt_2 \in T$.

The top congruence for trees is the tree analog of the left congruence for strings.

Proposition 4.2 *The top congruence is an equivalence relation on trees; it is a congruence with respect to catenations of pointed trees and trees (pointed or nonpointed).*

The *top index* of a tree language T is the number of $\sim_T$-equivalence classes.

A string language is regular if and only if it has finite index; however, that a tree language has finite top index is insufficient for it to be regular. For example, consider the tree language

$$L = \{a(b^i c^i) : i \geq 1\}.$$

Clearly, L has finite top index, but it is not regular. A second condition, namely regularity of so-called local views, has to be satisfied as well.

Definition 4.1 Let T be a tree language, a be a symbol in Σ, t be a pointed tree and T_f be a finite set of trees. The local view of T with respect to a, t and T_f is the string language

$$V_{a,t,T_f}(T) = \{t_1 \cdots t_n \in T_f^* \mid ta(t_1, \ldots, t_n) \in T\}$$

over the alphabet T_f.

For the purposes of local views we consider the trees in a finite set T_f to be symbols of the alphabet T_f, so the trees in T_f are primitive entities that can be catenated to form strings over T_f.

Proposition 4.3 *[7] A tree language is regular if and only if it is of finite top index and all its local views are regular string languages.*

At first glance it may appear that the local-view condition for regular tree languages is a condition on an infinite number of trees. But, if we exchange a tree t_1 in a finite set T_f by an equivalent—with respect to top congruence—tree t_2, then

$$V_{a,t,T_f}(T) = V_{a,t,(T_f \setminus \{t_1\}) \cup \{t_2\}}(T).$$

Hence, if T has finite top index, we need to check the local-view condition for only a finite number of tree sets T_f.

We split the proof of Theorem F into the following two parts.

Lemma 4.4 *Every caterpillar-regular tree language has finite top index.*

Proof Let $M = (Q, s_I, \delta, F)$ be a caterpillar automaton that recognizes the tree language T. We assume without loss of generality that each accepting computation of M terminates at the root of t. We prove that T is of finite top index, using a technique that is familiar from string languages.

Let Q_t be the set of all pairs (s, s') of states such that M, when starting at the root of t in state s, performs a computation on t that ends at the root once more in state s' without doing any *isRoot*, *isFirst*, or *isLast* transitions at the root. Since Q is finite, there are only finitely many sets Q_t.

Now, defining t to be equivalent to t' if and only if $Q_t = Q_{t'}$, we obtain an equivalence relation of finite index on the set of all trees. Furthermore, for any pointed tree t and equivalent trees t_1 and t_2, any computation on tt_1 and on tt_2 that enters the root of t_1 in state s and leaves it is state s' also enters the root of t_2 in state s and leaves it in state s'. Hence, for an accepting computation of M, it does not matter whether t_1 or t_2 is appended to the pointed tree t. Therefore, the equivalence relation refines the top congruence, which consequently must also be of finite index. □

Lemma 4.5 *Every caterpillar-regular tree language has only regular local views.*

PROOF Let $M = (Q, s_I, \delta, F)$ be a caterpillar automaton that recognizes the tree language T. We assume that each accepting computation of M terminates at the root of t. Furthermore, let a be a symbol in Σ, t be a pointed tree, and T_f be a finite set of trees. We need to prove that the local view of T with respect to a, t and T_f, namely the string language

$$V_{a,t,T_f}(T) = \{t_1 \cdots t_n \in T_f^* \mid ta(t_1, \ldots, t_n) \in T\},$$

is regular.

First, for a pair (s, s') of states consider the set $X_{s,s'}$ of all strings $t_1 \cdots t_n$ over T_f such that M, starting at the root of $a(t_1, \ldots, t_n)$ in state s performs a computation that returns to the root in state s' but never visits the root in between.

Let us examine a computation of M on a tree $ta(t_1, \ldots, t_n)$ that starts and ends at the root of the tree. We can break the computation down into segments that happen within t and others that happen within $a(t_1, \ldots, t_n)$. The outcome of the complete computation—whether it recognizes the tree or not—does not depend on the exact subcomputations within $a(t_1, \ldots, t_n)$ but only on the pairs s and s' of states that M is in at the start and the end of each such subcomputation. So each computation on $ta(t_1, \ldots, t_n)$ corresponds to a "short-cut" computation on the pointed tree t that replaces the subcomputations within $a(t_1, \ldots, t_n)$ with a short-cut transition from some state s to another state s' on the handle X. On the one hand, for each string $t_1 \cdots t_n$ in $V_{a,t,T_f}(T)$, there is an accepting short-cut computation on t with short-cut state transitions $(s_1, s'_1), \ldots, (s_m, s_m)$; hence,

$$t_1 \cdots t_n \in \cap_{i=1}^m X_{s_i,s'_i}.$$

On the other hand, for any accepting short-cut computation on t with short-cut state transitions $(s_1, s'_1), \ldots, (s_m, s_m)$ and any string $t_1 \cdots t_n$, if

$$t_1 \cdots t_n \in \cap_{i=1}^m X_{s_i,s'_i},$$

then

$$t_1 \cdots t_n \in V_{a,t,T_f}(T).$$

Therefore, $V_{a,t,T_f}(T)$ is the union of intersections of sets $X_{s,s'}$. Since there is only a finite number of such sets, for $V_{a,t,T_f}(T)$ to be regular it suffices to show that each $X_{s,s'}$ is regular.

We can test whether $t_1 \cdots t_n \in X_{s,s'}$, for $n \geq 1$, by executing the caterpillar automaton M on $a(t_1, \ldots t_n)$. Indeed, after M moves down from the root into some state z on either t_1's root by $s \xrightarrow[M]{\mathit{first}} z$ or on t_n's root by $s \xrightarrow[M]{\mathit{last}} z$, the only transitions we need consider are ones within the trees t_i or between the roots of the t_i. The crucial question is whether M can then reach a state z' at the root of some t_i such that $z' \xrightarrow[M]{\mathit{up}} s'$.

We transform the caterpillar automaton M into a string automaton $\dot{M}$ over T_f by replacing a computation of M on some t_i with a single state transition of $\dot{M}$. Since M may move from left to right and from right to left among the sibling subtrees $t_1, \ldots, t_n$, the string automaton $\dot{M}$ is a two-way automaton. Since M may perform tests *isFirst* and *isLast* at the roots of the trees t_i, the automaton $\dot{M}$ is an automaton with endmarkers $\vdash$ and $\dashv$ for the left and right ends of the strings, respectively.

We now give a more detailed description of the transformation of M into a two-way string automaton $\dot{M} = (\dot{Q}, \dot{s}_I, \dot{\delta}, \dot{F})$ with endmarkers over T_f. The automaton $\dot{M}$ has three move operations: left, stay and right. It has endmarkers $\vdash$ and $\dashv$ for the left and right ends of strings. The transition relation $\dot{\delta}$ is a subset of $\dot{Q} \times (\Sigma \cup \{\vdash, \dashv\}) \times \dot{Q} \times \{\mathit{left}, \mathit{right}, \mathit{stay}\}$. For the input string $\vdash a_1 \cdots a_n \dashv$ the automaton $\dot{M}$ starts on the first symbol a_1 in state $\dot{s}_I$. An accepting computation must end on $\dashv$ in a final state. The transition relation $\dot{\delta}$ does not contain any transitions $s \xrightarrow{\vdash} s', \mathit{left}$ or $s \xrightarrow{\dashv} s', \mathit{right}$, so $\dot{M}$ can never fall off either end of its input string.

It is well known that two-way automata recognize precisely the regular string languages as proved by Shepherdson [34].

The states of M are states of $\dot{M}$, but $\dot{M}$ has some additional states. For each walk that M can perform on some tree t in T_f, starting in state z at the root and returning to the root in state z', without doing any *isRoot*, *isFirst*, or *isLast* transitions at the root, we include a transition $z \xrightarrow[\dot{M}]{t} z', \mathit{stay}$. Thus, we can reduce the whole computation of M within a tree t to a single transition of $\dot{M}$ on t.

For each transition $z \xrightarrow[M]{a} z'$ in M and each t in T_f with root label a, we add the transition $z \xrightarrow[\dot{M}]{t} z'$ to $\dot{M}$.

For each transition $z \xrightarrow[M]{\textit{left}} z'$ in M and each t in T_f, we add $z \xrightarrow[\dot{M}]{t} z', \textit{left}$ to $\dot{M}$. We treat the right transitions analogously.

For each transition $z \xrightarrow[M]{\textit{isFirst}} z'$ in M, $\dot{M}$ does a left move from state z on any symbol in Σ_{T_f}, then moves right again on $\vdash$. Again, we treat the test *isLast* analogously.

Finally, we have to ensure that $\dot{M}$ starts and finishes appropriately. Hence, we introduce new initial and final states $\dot{s}_I$ and $\dot{s}_F$ for $\dot{M}$. Then, $\dot{M}$ either moves from $\dot{s}_I$ into any state z such that $s \xrightarrow[M]{\textit{first}} z$ without changing its position or moves from $\dot{s}_I$ into any state z such that $s \xrightarrow[M]{\textit{last}} z$ while simultaneously changing its position to the last T_f symbol in the input string. Furthermore, for any state z such that $z \xrightarrow[M]{\textit{up}} z'$, $\dot{M}$ moves to the right endmarker and enters state $\dot{s}_F$.

By construction, $\dot{M}$ recognizes $X_{s,s'}$. $\square$

5 Caterpillars and tree pattern matching

Kilpeläinen [19] uses tree pattern matching and tree inclusion as a means of querying databases of structured documents. Although originally designed for context specification, we can also employ caterpillar expressions and automata to specify queries for document databases. It turns out that caterpillar expressions and automata are able to represent several variants of tree inclusion that Kilpeläinen has investigated. In particular, we reprove two of Kilpeläinen's time-complexity results [19, 20] using the notion of caterpillar automata. The proofs are simpler and more uniform than Kilpeläinen's original proofs.

Tree pattern matching and tree inclusion use a tree p to specify a pattern; the goal is to find occurrences of the pattern tree p in a target tree t (in general, in a set of target trees). We now introduce the various notions of tree inclusion studied by Kilpeläinen.

A mapping $f : \textit{nodes}(p) \longrightarrow \textit{nodes}(t)$ is a *tree inclusion* of p in t if and only if the following three conditions hold:

1. f is injective.
2. f preserves labels: for each node ν, the labels of ν and $f(\nu)$ are identical.
3. f preserves ancestorship: for any two nodes μ and ν of p, μ is an ancestor of ν if and only if $f(\mu)$ is an ancestor of $f(\nu)$.

A tree inclusion $f : \textit{nodes}(p) \longrightarrow \textit{nodes}(t)$ is *ordered* if and only if it preserves the postorder of nodes (or, equivalently, the preorder); that is, for any two nodes μ and ν of p, the node μ precedes ν in the postorder of nodes if and only if $f(\mu)$ precedes $f(\nu)$.

A pattern tree p specifies a context mapping $\mathcal{C}_p$ with respect to (ordered) tree inclusion as follows: For a tree t and a node ν in t, the node ν is in the context set $\mathcal{C}_p(t)$ if and only if there is an (ordered) inclusion $f : \mathit{nodes}(p) \longrightarrow \mathit{nodes}(t)$ that maps the root of p to ν.

An *(ordered) path inclusion* of p in t is an (ordered) tree inclusion f of p in t that preserves the parent–child relationship; that is, if μ is the parent of ν in p, then $f(\mu)$ is the parent of $f(\nu)$. An *(ordered) range inclusion* of p in t is an (ordered) tree inclusion f of p in t that maps the set of children of a node ν in p onto a continous range of children of $f(\nu)$. An *(ordered) child inclusion* of p in t is an (ordered) tree inclusion f of p in t that surjectively maps the set of children of a node ν in p onto *the* set of children of $f(\nu)$. This is the classical notion of tree pattern matching. An *(ordered) subtree inclusion* f of p in t is an (ordered) child inclusion f of p in t that maps each external node of p to an external node of t; that is, p is isomorphic to a subtree of t.

A pattern tree p specifies contexts $\mathcal{C}_p^{(\mathrm{O})\mathrm{T}}$, $\mathcal{C}_p^{(\mathrm{O})\mathrm{P}}$, $\mathcal{C}_p^{(\mathrm{O})\mathrm{R}}$, $\mathcal{C}_p^{(\mathrm{O})\mathrm{C}}$ and $\mathcal{C}_p^{(\mathrm{O})\mathrm{S}}$ with respect to (ordered) tree, path, region, child and subtree inclusion, respectively.

Theorem G *For each pattern tree p, the context mappings $\mathcal{C}_p^{(\mathrm{O})\mathrm{P}}$, $\mathcal{C}_p^{(\mathrm{O})\mathrm{R}}$, $\mathcal{C}_p^{(\mathrm{O})\mathrm{C}}$ and $\mathcal{C}_p^{(\mathrm{O})\mathrm{S}}$ are caterpillar context mappings. Furthermore, for each ordered type of inclusion, there are a caterpillar expression and a deterministic caterpillar automaton of sizes* $\mathrm{O}(p)$ *that denote the corresponding context mapping.*

Proof For each pattern tree p and each target tree t the context sets $\mathcal{C}_p^{\mathrm{P}}$, $\mathcal{C}_p^{\mathrm{R}}$, $\mathcal{C}_p^{\mathrm{C}}$ and $\mathcal{C}_p^{\mathrm{S}}$ of the unordered inclusion type are finite unions of some context sets $\mathcal{C}_p^{\mathrm{OP}}$, $\mathcal{C}_p^{\mathrm{OR}}$, $\mathcal{C}_p^{\mathrm{OC}}$ and $\mathcal{C}_p^{\mathrm{OS}}$ of the ordered inclusion type, respectively. Hence, we need to consider only the ordered inclusion type.

We first demonstrate the case of ordered path inclusion, constructing, by induction on the pattern tree p, a caterpillar expression E_p such that the caterpillar-regular language $\mathrm{L}(E)$ denotes $\mathcal{C}_p^{\mathrm{OP}}$:

If $p = a$, $a \in \Sigma$, then $E_p = a$.
If $p = a(p_1, \dots, p_n)$, $n \geq 1$, then

$$E_p = a\, \mathit{first}\ \mathit{right}^*\ E_{p_1}\ \mathit{right}\ \mathit{right}^*\ E_{p_2}\ \cdots\ \mathit{right}\ \mathit{right}^*\ E_{p_n}\ \mathit{up}.$$

The observation that $\mathrm{L}(E)$ denotes $\mathcal{C}_p^{\mathrm{OP}}$ is based on the following characterization of $\mathcal{C}_p^{\mathrm{OP}}$: Letting $p = a(p_1, \dots, p_n)$, $n \geq 0$, a node ν of a target tree t is in the context set $\mathcal{C}_p^{\mathrm{OP}}(t)$ if and only if ν is labeled a and ν has children $\nu_1, \dots, \nu_n$ such that ν_i is in the context set $\mathcal{C}_{p_i}^{\mathrm{OP}}(t_i)$, $1 \leq i \leq n$, and $t = a(t_1, \dots, t_n)$. The crucial property of E_p is that each caterpillar w in $\mathrm{L}(E_p)$, when started at a node ν of a pattern tree t, only visits ν and descendants of ν and if it is successful, then it terminates at node ν. This property guarantees $\mathcal{C}_{\mathrm{L}(E_p)} = \mathcal{C}_p^{\mathrm{OP}}$.

For the other three types of inclusion we provide only the inductive definitions of caterpillar expressions.

Ordered range inclusion:

$$E_a = a,$$
$$E_{a(p_1,\dots,p_n)} = a\,\mathit{first}\ \mathit{right}^*\ E_{p_1}\ \mathit{right}\ E_{p_2}\ \cdots\ \mathit{right}\ E_{p_n}\ \mathit{up}.$$

Ordered child inclusion:

$$E_a = a,$$
$$E_{a(p_1,\dots,p_n)} = a\,\mathit{first}\ E_{p_1}\ \mathit{right}\ E_{p_2}\ \cdots\ \mathit{right}\ E_{p_n}\ \mathit{isLast}\ \mathit{up}.$$

Ordered subtree inclusion:

$$E_a = a\ \mathit{isLeaf},$$
$$E_{a(p_1,\dots,p_n)} = a\,\mathit{first}\ E_{p_1}\ \mathit{right}\ E_{p_2}\ \cdots\ \mathit{right}\ E_{p_n}\ \mathit{isLast}\ \mathit{up}.$$

Obviously, the expressions E_p we have constructed are of sizes linear in the size of p. Furthermore, deterministic caterpillar automata of the same size are easily constructed from the expressions. □

If we combine Theorems A and G, we obtain new proofs of two of Kilpaläinen's time-complexity results [19, 20] on tree inclusion.

Theorem H *Given a pattern tree p with m nodes and a target tree t with n nodes, we can compute in time $\mathrm{O}(m \times n)$ all ordered path and range inclusions of p in t.*

The proof is simpler than Kilpeläinen's and uniform for both types of inclusion. The same technique also works for unordered child and subtree inclusions, but the time complexities are worse than those obtained by Kilpeläinen.

In the proof of Theorem G for unordered inclusions the sizes of the caterpillar expressions and automata we obtain are exponential in the sizes of the pattern trees. It is an open question whether polynomially sized expressions or automata can be found.

If we restrict the alphabet Δ to only movements, omitting the positional tests, Theorem G still holds for (ordered) path and region inclusion but no longer holds for (ordered) child or subtree inclusions [29].

Theorem I *Let $\Delta_{\mathrm{r}} = \{\mathit{up}, \mathit{left}, \mathit{right}, \mathit{first}, \mathit{last}\}$. For any pattern tree p, there are caterpillar expressions and deterministic caterpillar automata over the reduced alphabet $\Delta_{\mathrm{r}} \cup \Sigma$ that denote $\mathcal{C}_p^{(\mathrm{O})\mathrm{P}}$ and $\mathcal{C}_p^{(\mathrm{O})\mathrm{R}}$, respectively. For each of the two ordered types of inclusion, expressions and automata of sizes $\mathrm{O}(p)$ can be found.*

PROOF The proofs for the cases of (ordered) path and range inclusion in Theorem G also carry the stronger results of this theorem, since the expressions and automata that were constructed in the proofs of Theorem G make use of only the reduced alphabet. □

Theorem J *Let* $\Delta_{\mathrm{r}} = \{up, left, right, first, last\}$.

1. *For any pattern tree* p *with at least two nodes, there is no caterpillar-regular language over the reduced alphabet* $\Delta_{\mathrm{r}} \cup \Sigma$ *that denotes either* $\mathcal{C}_p^{\mathrm{OC}}$ *or* $\mathcal{C}_p^{\mathrm{C}}$.
2. *For any pattern tree* p*, there is no caterpillar-regular language over the reduced alphabet* $\Delta_{\mathrm{r}} \cup \Sigma$ *that denotes either* $\mathcal{C}_p^{\mathrm{OS}}$ *or* $\mathcal{C}_p^{\mathrm{S}}$.

PROOF The proofs use the surjectivity of the inclusion mapping to obtain contradictions.

1. Let p be a pattern tree with at least two nodes; that is, $p = a(p_1, \dots, p_n)$, $n \geq 1$. We construct a target tree $t = a(p_1, \dots, p_n, p_1, \dots, p_n)$ such that t contains two copies of each of p's subtrees p_i, $1 \leq i \leq n$.

   There is a natural relationship between the roots of p and t and between each nonroot node of p and its two copies in t. It is intuitive that each walk of a caterpillar on p corresponds to either one or two related walks of the same caterpillar on t, depending on whether the walk starts at the root or at an interior node, respectively. We omit a formal proof, which uses induction on the length of the walk since it is straightforward. The crucial observation is that when a walk on p moves down from the root, the corresponding walk on t moves down either with a *first* move, in the left copy of the forest $(p_1, \dots, p_n)$, or with a *last* move, in the right copy of the forest $(p_1, \dots, p_n)$. In both cases the caterpillar is at a node in t that corresponds naturally to the node the caterpillar is at when traversing p. As a consequence, each caterpillar over the restricted alphabet $\Delta_r \cup \Sigma$ that successfully completes its walk when started at p's root also successfully completes its walk when started at t's root.

   We now assume that there is a regular language L over $\Delta_r \cup \Sigma$ such that either $\mathcal{C}_L = \mathcal{C}_p^{\mathrm{OC}}$ or $\mathcal{C}_L = \mathcal{C}_p^{\mathrm{C}}$. Since the identity mapping is an (ordered) child inclusion of p in p, we can find a caterpillar w in L such that w succeeds when started at p's root; hence, w also succeeds when started at t's root. But there is no child inclusion of p in t that maps p's root to t's root since the two roots have different numbers of children. Therefore, we have obtained a contradiction.

2. Let p be any pattern tree. We construct a target tree t by appending a new singleton node to one of p's leaves. Observe that each caterpillar over the restricted alphabet $\Delta_r \cup \Sigma$ that successfully completes its walk when started at p's root also successfully completes its walk when started at t's root.

   Assume that there is a regular language L over $\Delta_r \cup \Sigma$ such that either $\mathcal{C}_L = \mathcal{C}_p^{\mathrm{OS}}$ or $\mathcal{C}_L = \mathcal{C}_p^{\mathrm{S}}$. Since the identity mapping is a subtree inclusion of p in p, there is a caterpillar w in L such that w succeeds when started at p's root; hence, it also succeeds when started at t's root. But there is no subtree inclusion of p in t that maps p's root to t's root. For, any such subtree inclusion would induce a inverse image for each of t's leaves and for t's root; hence, the inclusion mapping must be surjective. But t has one more node than p; therefore, we have obtained a contradiction.

□

6 Closing remarks

Our investigation of caterpillar-regular and regular tree languages leaves a number of questions unanswered. Among them are:

1. For all pattern trees p, are the context mappings $\mathcal{C}_p^{\mathrm{OT}}$ and $\mathcal{C}_p^{\mathrm{T}}$ caterpillar contexts?
   Our strong intuition is: No, they are not.
2. Is the family of caterpillar-regular languages closed under complement?
   Our strong intuition is: No, they are not.
3. Can each caterpillar-regular language be denoted by a strongly deterministic caterpillar automaton (in the sense that the tree-traversing caterpillar automaton of Fig. 4 is strongly deterministic?).
   Our tentative conjecture is: No, they cannot.
4. Is each regular tree language also caterpillar regular?
   Our strong intuition is: No, they are not.

We have mentioned our interest in document classes that are constrained by some type of grammars, for example by XML DTDs. The results in this paper hold for the classes of all documents that are given as trees over a fixed alphabet. Applications, particularly applications based on document databases, often come with non-trivial document grammars. It is therefore pertinent to generalize our results to document classes defined by non-trivial document grammars and to address the open questions in this light.

Children, and even adults, were able to draw complex figures very quickly in the Logo language [1] using the turtle as metaphor and guide. We hope that our use of caterpillars will garner a similar response from graphics designers.

References

[1] H. Abelson and A. A. diSessa. *Turtle Geometry: The Computer as Medium for Exploring Mathematics*. MIT Press, Cambridge, MA, 1980.

[2] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1:68–88, 1997.

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

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

[5] 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.

[6] 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.

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

[8] A. Brüggemann-Klein and T. Schroff. Grammar-compatible stylesheets. In C. Nicholas and D. Wood, editors, *Proceedings of the Third International Workshop on Principles of Document Processing (PODP 96)*, pages 51–58, Heidelberg, 1996. Springer-Verlag. Lecture Notes of Computer Science 1293.

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

[10] A. Brüggemann-Klein and D. Wood. Caterpillars, context, tree automata and tree pattern matching, 1999. Proceedings of the Fourth International Conference on Developments in Formal Language Theory (DLT '99).

[11] H. Comon, M. Daucher, R. Gilleron, S. Tison, and M. Tommasi. Tree automata techniques and applications, 1998. Available on the Web from l3ux02.univ-lille3.fr in directory tata.

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

[13] M. P. Consens, F. C. Eigler, M. Z. Hasan, A. O. Mendelzon, E. G. Naik, A. G. Ryman, and D. Vista. Architecture and application of the Hy+ visualization system. *IBM Systems Journal*, 33(3):458–476, 1994.

[14] B. Courcelle. A representation of trees by languages. *Theoretical Computer Science*, 7:25–55, 1978.

[15] F. Gécseg and M. Steinby. *Tree Automata*. Akadémiai Kiadó, Budapest, 1984.

[16] F. Gécseg and M. Steinby. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages, Volume 3, Beyond Words*, pages 1–68. Springer-Verlag, Berlin, Heidelberg, New York, 1997.

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

ization.

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

[19] P. Kilpeläinen. *Tree Matching Problems with Applications to Structured Text Databases.* PhD thesis, Department of Computer Science, University of Helsinki, 1992. Series of Publications A, No. A/11992-6.

[20] Pekka Kilpeläinen and Heikki Mannila. Ordered and unordered tree inclusion. *SIAM Journal on Computing*, 24:340–356, 1995.

[21] Nils Klarlund and Michael I. Schwartzbach. Graph types. In *Twentieth Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 196–205, New York, NY, 1993. ACM.

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

[23] P. E. Lauer, P. R. Torrigiani, and M. W. Shields. COSY: A system specification language based on paths and processes. *Acta Informatica*, 12:109–158, 1979.

[24] H. Lie. Cascading style sheets. http://www.w3.org/Style/css/, 1997.

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

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

[27] A. O. Mendelzon and P. T. Wood. Finding regular simple paths in graph databases. *SIAM Journal of Computing*, 24(6), December 1995.

[28] M. Merk. Spezifikation von Mustern als Kontexte. Master's thesis, Institut für Informatik, Universität Freiburg, 1994.

[29] M. Merk. Spezifikation von mustern als kontexte. Master's thesis, Institut für Informatik, Universität Freiburg, July 1994.

[30] J. D. Morgenthaler. *Static Analysis for a Software Transformation.* PhD thesis, University of California, San Diego, Department of Computer Science and Engineering, 1997. Also available as Technical Report CS97-552 and from URL: `http://www-cse.ucsd.edu/users/jdm/Papers/Dissertation.html`.

[31] 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.

[32] 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.

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

[34] J. C. Shepherdson. The reduction of two-way automata to one-way automata. *IBM Journal on Research and Development*, 3:198–200, 1959.

[35] M. Takahashi. Generalization of regular sets and their application to a study of context-free languages. *Information and Control*, 27(1):1–36, January 1975.

[36] J. W. Thatcher. Characterizing derivation trees of context-free grammars through a generalization of finite automata theory. *Journal of Computer and System Sciences*, 1:317–322, 1967.

[37] X. Wang and D. Wood. Xtable—A tabular editor and formatter. In A. Brown, A. Brüggemann-Klein, and A. Feng, editors, *EP96, Proceedings of the Sixth International Conference on Electronic Publishing, Document Manipulation and Typography*, pages 167–180, 1996. Special Issue, Electronic Publishing—Origination, Dissemination and Design 8(2 and 3).