Electronic Communications of the EASST
Volume 26 (2010)

Manipulation of Graphs, Algebras and Pictures

Essays Dedicated to Hans-Jörg Kreowski on the Occasion of His 60th Birthday

Towards the Tree Automata Workbench MARBLES

Frank Drewes

16 pages

Guest Editors: Frank Drewes, Annegret Habel, Berthold Hoffmann, Detlef Plump
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Towards the Tree Automata Workbench MARBLES*

Frank Drewes

Institutionen för datavetenskap
Umeå universitet, S-901 87 Umeå (Sverige)
drewes@cs.umu.se

Abstract: The conceptual ideas that are intended to become the basis for the tree automata workbench MARBLES[1] are sketched. The goal is to design and implement an extensible system that facilitates experiments with virtually any kind of algorithm on tree automata. Moreover, the system will be released with a library and an application programmer's interface to make it accessible to anyone who wants to apply tree automata algorithms in research and development.

Keywords: tree automaton; tree automata workbench; MARBLES

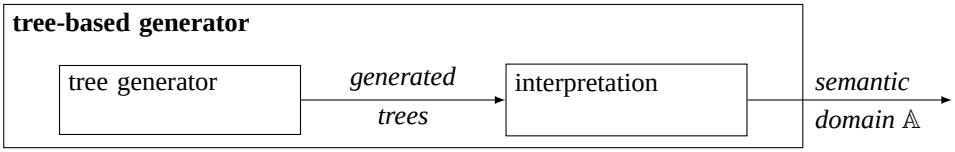
1 Introduction

Already in the 1960s, researchers realized that large parts of the theory of finite automata can be generalized by replacing strings with trees, retaining most of the positive algorithmic results and closure properties. This observation gave rise to a flourishing theory, including a large number of techniques and algorithms for the analysis, modification, and synthesis of various kinds of tree recognizers, tree grammars, and tree transducers [GS84, NP92, GS97, FV98, CDG+07]. Throughout the rest of this paper, all devices that fall into one of these categories will be called tree automata. Nowadays, probably more theoretical research than ever before is done in this area, a fact that is explained by a constantly growing number of applications in fields such as verification and model checking [GK00, AJMd02, Löd02, FGV04], natural language processing [KG05, GKM08], XML processing [Sch07], code selection in compilers [FSW94, Bor04], and generation of graphs and pictures [Eng94, Dre06].

The system TREEBAG[2] uses tree generators to generate sets of objects over arbitrary domains. The central data type of TREEBAG is the ranked and ordered tree, with nodes labelled by symbols taken from a ranked alphabet Σ. In other words, every symbol $f \in \Sigma$ comes with a rank $k \geq 0$, such that a node labelled with f is required to have exactly k children (which are totally ordered). This means that a tree in the sense of TREEBAG is a term, i.e., a well-formed expression composed of abstract (i.e., "meaningless") operation symbols, each having a specified rank that determines the number of subexpressions. TREEBAG deals with two types of tree automata on such trees, namely tree grammars and tree transducers. A tree grammar is a device that generates trees out of itself, whereas a tree transducer transforms input trees into output trees. A *tree generator* is a tree grammar composed with a (possibly empty) sequence of tree transducers.

* Dedicated to Hans-Jörg Kreowski on the occasion of this 60th birthday.
[1] Tree Automata Workbench = taw = a large marble, a game of marbles (Oxford New Amer. Dict.).
[2] Tree-Based Generator

In the well-known way, trees of the type described above can be assigned a semantics by choosing a domain $\mathbb{A}$ and associating an operation on $\mathbb{A}$ (of the appropriate arity) with each symbol in the ranked alphabet Σ. In other words, a Σ-algebra is specified that maps every tree to an element of $\mathbb{A}$. Together, a tree generator and an algebra constitute a *tree-based generator* whose generated language is a subset of $\mathbb{A}$:



TREEBAG makes it possible to assemble tree-based generators interactively. This makes TREEBAG very flexible, because arbitrary combinations of tree grammars, tree transducers, and algebras can be used. However, in another respect, TREEBAG is quite restricted. All that can be done when a tree-based generator has been assembled is to execute it. In contrast, the usefulness of tree automata in most application areas does not primarily lie in the fact that they can be executed. Their real advantage is that they are simple enough to be effectively analyzed and manipulated. For instance, in a model checking application, tree automata may be generated that model safety and liveness properties of a protocol to be verified. Analyzing these automata then corresponds to checking correctness criteria. A tool that is supposed to be useful in such situations must make it possible to assemble not only tree automata but also *algorithms on tree automata*. This means that tree automata are mainly perceived as objects to be analyzed and manipulated, rather than as executable algorithms. MARBLES is intended to become such a tool. Its major purpose is to provide researchers with a software environment and infrastructure that enables them to create, use, and experiment with algorithms on tree automata.

In addition to TREEBAG, there are several other systems that implement certain types of tree automata or algorithms on them.

AutoWrite (http://dept-info.labri.fr/~idurand/autowrite) is a system that allows the user to check properties of term rewrite systems by means of tree automata constructions. In particular, it allows to load, save, and combine bottom-up tree recognizers. Using the graphical user interface, one can build and manipulate bottom-up tree recognizers related to the term rewrite systems whose properties one wants to check.

Forest FIRE (http://www.loekcleophas.com) is a toolkit focusing on recognition, pattern matching, and parsing algorithms in connection with regular tree languages. The system has been developed on the basis of detailed taxonomies, with the major purpose of gaining a deeper conceptual understanding of how the ideas and techniques used in various tree automata constructions are related to each other.

MONA (http://www.brics.dk/mona) is a tool for checking the validity of formulas in the weak second-order theory of one successor (WS1S) or of two successors (WS2S). For deciding WS2S, the decision procedures convert a given formula into a so-called guided tree automaton, a variant of a bottom-up tree recognizer, and analyze this automaton.

Tiburon (http://www.isi.edu/licensed-sw/tiburon) is a command-line based package of algorithms on weighted regular tree grammars, context-free string grammars, and tree transducers, including various analyzers, modifiers, and synthesizers. Tiburon has mainly been developed for applications in Natural Language Processing, but can be used for other purposes as well.

Timbuk (http://www.irisa.fr/lande/genet/timbuk) is a toolkit for reachability proofs in term rewrite systems, among other techniques by manipulating nondeterministic bottom-up tree recognizers. It is intended to be used for the verification of programs and cryptographic protocols.

The proposed system MARBLES differs from each of these systems in several respects. Most notably, the systems above have all been developed with a particular application or problem area in mind. The one that is probably closest to MARBLES is Tiburon. The devices and algorithms implemented in Tiburon are typical even for MARBLES, and it is conceivable that Tiburon could, in principle, be extended to include most of the intended functionality of MARBLES. However, its design was not guided by MARBLES' emphasis on a general concept that is reflected in both the graphical user interface and the application programmers interface and that allows the system to be adapted and extended by researchers with different needs. This distinguishes MARBLES from the other systems. The intention behind it is to support tree automata research in general, by providing researchers with a suitable platform and infrastructure for their own extensions.

Of course, the list of systems above could be extended by mentioning various implementations of general term rewrite systems, (functional) programming languages based on term rewriting, theorem provers, and systems for executable algebraic specifications, because most of them include tree automata as special cases (at least in the unweighted case). However, the point is that such systems and languages are *too* general to provide support for the kind of problems MARBLES is supposed to address.

This article is a revised version of [Dre09b]. Its remainder is structured as follows. The next section presents some aspects of TREEBAG that have, in one way or the other, inspired the intended characteristics of MARBLES. In Section 3, some of the different types of trees, tree automata, and tree automata algorithms that should, in principle, be covered by MARBLES, are discussed. Section 4 presents initial ideas regarding some of the concepts needed for making this possible. Finally, Section 5 concludes the paper.

2 TREEBAG

Let us now have a slightly closer look at the concepts and design principles of TREEBAG. The following description is intentionally kept at a rather abstract level. Concrete classes of, e.g., tree grammars and algebras in TREEBAG are sometimes mentioned as examples, mainly for readers who happen to be familiar with tree automata theory. Readers who want to inform themselves in more detail should consult the TREEBAG user manual (see http://www.cs.umu.se/~drewes/treebag) or [Dre06] for the theory behind.

The work on TREEBAG was started during the second half of the 1990s, when the author was a member of Hans-Jörg Kreowski's research group at the University of Bremen. Around this time, context-free graph and collage grammars were two of the major research topics of the group; see, e.g., [HKV91, HKL93, HKT93, DHKT95, DK96, DHK97, DK99]. Their generative power can be characterized by combinations of a certain type of tree grammar (namely the regular tree grammar) with suitable algebras in the style of Mezei and Wright [MW67], i.e., the grammars can be viewed as tree-based generators. For graphs, this has been made explicit by Engelfriet in [Eng94], and for collages by the author in [Dre96, Dre00]. See also [DEKK03], where this characterization was used to establish certain decidability results for collage languages. In [Eng80],

Engelfriet discusses symbolic computation by tree transductions, which is essentially the same idea, applied to transformation rather than generation: a tree transduction, together with algebras interpreting the input and output trees, is considered as a symbolic algorithm that performs a computation on abstract trees rather than on the concrete objects of the two domains in question.

Although the results mentioned above use only regular tree grammars, it should be obvious that one may in fact combine arbitrary kinds of tree generators with any sort of algebra, yielding a large number of different grammatical formalisms with comparatively little effort. Being a rather straightforward implementation of this idea (in Java), TREEBAG allows its user to assemble tree-based generators of various kinds. There are four major abstract classes, namely *tree grammars*, *tree transducers*, *algebras*, and *displays*. The first three represent the corresponding formal concepts, whereas displays show the results of the generating process. Concrete subclasses of the four abstract classes implement particular types of tree grammars, tree transducers, algebras, and displays. For example, the classes `generators.ET0LTreeGrammar` and `generators.mtTransducer` implement ET0L tree grammars and macro tree transducers, resp. (See [Eng80, CF82, EV85, FV98] for the latter.) The user can define specific instances (usually in ordinary ASCII text files) of such concrete classes and use them in assembling tree-based generators. Such instances are called components in the following.

Figure 1 shows a typical situation when working with TREEBAG. Window 1 is the main window of the system, the so-called worksheet. When the user loads a component, it is represented on the worksheet as a blob. These blobs represent the nodes of a directed acyclic graph whose edges determine the data flow between components. The data-flow edges are interactively established by the user, subject to a few rather obvious rules: The output of a tree grammar or tree transducer can become the input of any number of tree transducers and algebras, and the output of an algebra can become the input any number of displays. The configuration in Figure 1 consists of a regular tree grammar, a free term algebra with a corresponding tree display, a top-down tree transducer, and two copies of a collage algebra, each with its corresponding collage display. With each display component, a window is associated, namely the windows numbered 3–5. These windows show the tree generated by the regular tree grammar, its interpretation by the collage grammar, and the interpretation of the transformed tree by (another instance of) the same collage algebra.

An additional window (numbered 2 in the figure) contains buttons that provide access to the user commands of the regular tree grammar. Double clicks on the other components on the worksheet would open similar sections in this window, each one being populated by the individual commands understood by the respective component.

Let us now discuss two aspects of the design of TREEBAG which are expected to have some influence on MARBLES. In fact, these two aspects are quite closely related and can be seen as the two sides of the same coin.

From the point of view of the user, the way in which components can be interconnected depends only on their types, i.e., whether they are tree grammars, tree transducers, algebras or displays. In other words, if the user wants to connect a tree grammar and a tree transducer, this can be done regardless of whether the tree grammar at hand is a regular tree grammar, ET0L tree grammar, context-free tree grammar or whatever type of tree grammar might at some point in time be implemented in TREEBAG. Of course, users must interconnect the “right” components to achieve the desired effect. Every concrete component class provides the user with a set of commands that can be used to interact with components of this class (recall Figure 1, where

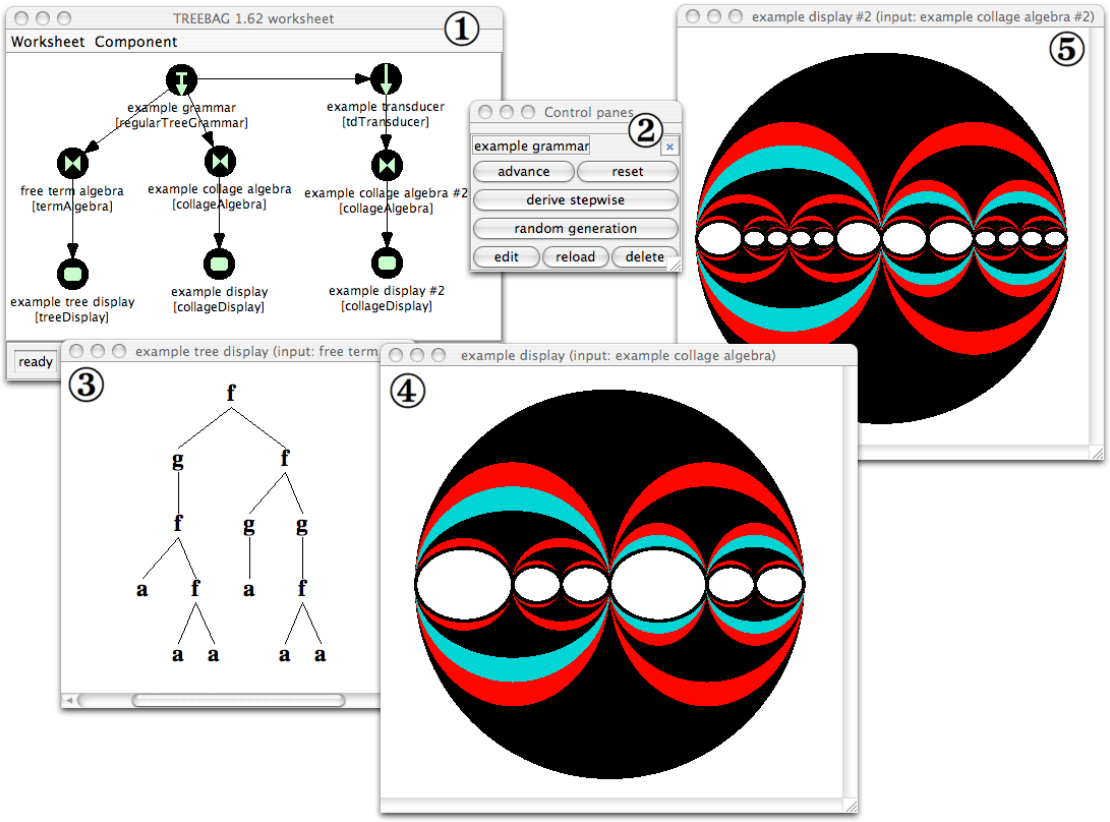


Figure 1: A typical configuration of TREEBAG

Window 2 contains buttons for the commands provided by the implementation of regular tree grammars). While the commands would be different for, e.g., ET0L tree grammars, this has no influence on the way in which regular tree grammars or ET0L tree grammars can be connected to other components.

The person who implements new classes of tree grammars, tree transducers, algebras or displays will find out that the properties mentioned in the previous paragraph simply reflect properties of the implementation. The core of TREEBAG does not make any distinction between, e.g., different classes of tree grammars. In fact, consider the file defining the regular tree grammar used in Figure 1:

```
generators.regularTreeGrammar("example grammar"):
( { S, A },
  { f:2, g:1, a:0 },
  { S -> f[S,S],
    S -> g[A],
    A -> f[A,A],
    A -> a },
  S )
```

When the user instructs TREEBAG to load this component, TREEBAG parses only the first line, to discover that the user wishes to load an instance of `generators.regularTreeGrammar`. The rest of the file uses a syntax which is specific to the implementation of this class and therefore unknown to (the core of) TREEBAG. To handle this, TREEBAG dynamically tries to load the class `generators.regularTreeGrammar` and, upon success, creates an (uninitialized) object of this class. Now, it lets this object, which is required to contain a method called `parse`, initialize itself by parsing the remainder of the file. Each of the four abstract component types of TREEBAG requires its concrete subclasses to implement such a parsing method. To handle component-specific user commands, each concrete subclass provides two further methods. The first returns, at any point in time, the list of user commands available at that moment (which means that the list of commands may change), while the second executes a given command.

This structure makes it possible to extend TREEBAG by new classes of tree grammars, tree transducers, algebras, and displays in an easy way, without having to change existing parts of the system. One only has to implement it as a subclass of the appropriate abstract component class and place it in the appropriate directory. Immediately afterwards (provided that everything has been done correctly), it is possible to load instances of this class onto the worksheet, interconnect them with other components, and work with them.

It may be interesting to note that the implementations of some of the classes currently available in TREEBAG make use of decomposition results from the literature. For example, a so-called branching synchronization tree grammar of nesting depth n can be decomposed into a regular tree grammar and a sequence of n top-down tree transducers (see [DE04]). During the parsing step, the implementation of this class in TREEBAG performs this decomposition and writes the $n+1$ components onto the hard disk (in the syntax required by the respective classes). Afterwards, it uses TREEBAG's loading mechanism to load them as internal variables hidden from the user (i.e., so that they do not appear on the worksheet). Every user command is basically forwarded to these internal components, and whichever output tree they produce is returned. In this way, the implementation of the class becomes considerably easier and less error prone than a direct one.

3 Trees, Tree Automata, and Tree Automata Algorithms

As mentioned in the introduction, the major intended purpose of MARBLES is to make it possible to apply and experiment with algorithms on tree automata. The aim is to design MARBLES in such a way that it accommodates virtually all kinds of tree automata algorithms. While this does not mean that all such algorithms should readily be implemented in the system, the design of MARBLES should enable researchers (and application programmers) interested in a particular type of algorithm on tree automata to make the necessary extensions. As in the case of TREEBAG, this should be possible without changes to existing parts. However, compared to TREEBAG, the design challenge is considerably bigger for MARBLES, because its intended coverage is much wider. It seems to be reasonable to distinguish between (at least) three central categories of objects: trees, tree automata, and tree automata algorithms. Each of them may, in principle, have any number of subcategories one may wish to implement in MARBLES. In the following, some of the possible subcategories of each will be discussed to illustrate this point.

3.1 Trees

In the traditional setting (and in TREEBAG), tree automata work on trees over ranked alphabets, as explained above. This is appropriate, because trees are supposed to be evaluated by algebras by associating with every symbol of rank k a k-ary function on some domain. However, tree automata on unranked trees have received a lot of attention during recent years. In this setting, symbols are unranked, and the number of children of a node does not depend on its label. It turns out that this variant is well suited for applications in connection with XML, because XML documents can appropriately be viewed as unranked trees. (For example, a node corresponding to a list structure in HTML may have any number of children of type *list item*.) Thus, an XML document type corresponds to a tree language of unranked trees, and a tree transducer on unranked trees corresponds to a transformation between XML document types.

While the two types of trees mentioned are the only ones that play a major role in contemporary research on tree automata, this situation may change in the future. Thus, MARBLES should allow programmers to implement other classes of trees than just these.

3.2 Tree Automata

Tree automata can be classified according to various criteria. An important observation is that the resulting classifications are, to a rather large extent, orthogonal.

Perhaps the most obvious classification is the one that gave rise to the structure of TREEBAG, namely the distinction between tree grammars, tree recognizers (which are not directly available in TREEBAG), and tree transducers. From an abstract point of view, a tree grammar is a formal device that generates output trees without requiring input. As usual, the tree recognizer is the dual concept. It takes a tree as input and computes an output value, usually in the range $\{0,1\}$, indicating whether the tree is accepted or not. Finally, a tree transducer is a formal device transforming input trees into output trees.

The second classification distinguishes between tree automata according to the type of trees they act upon, i.e., tree automata on ranked or unranked trees. Each of the types of tree automata in the first classification can be ranked or unranked. In this sense, these two classifications are orthogonal. In fact, one may even wish to consider tree transducers that turn unranked trees into ranked ones, or vice versa.

Finally, one may consider weighted tree automata [FV09], which deal with tree series instead of tree languages. A tree series is a mapping $\psi\colon \mathrm{T}_\Sigma \to \mathbb{S}$, where T_Σ denotes the set of all trees over a given alphabet, and $\mathbb{S}$ is a semiring. In other words, weighted tree automata generalize the traditional case, which is obtained by choosing the Boolean semiring. Even this third classification is orthogonal to the two previous ones, provided that the used definition of tree automata is general enough to include the weighted case.

It is interesting to note that, from an abstract point of view as well as from the point of view of system design, weighted tree recognizers are very similar to algebras. Both take a tree as input and compute a value in some other domain. In fact, this observation yields one of the possible ways to define the semantics of weighted tree automata. The initial algebra semantics of a weighted tree automaton A over a semiring S associates a Σ-algebra $\mathscr{A}$ with A [FV09, pp. 322–323]. The domain of $\mathscr{A}$ is S^k, where k is the number of states of A. The evaluation of

a tree with respect to $\mathscr{A}$ yields the tuple of weights carried by the states at the root node of the tree.

3.3 Algorithms on Tree Automata

Many useful algorithms on tree automata have been described in the literature. For classification purposes, it is useful to distinguish between analyzers, synthesizers, and decomposition algorithms.

An analyzer for tree automata takes a tree automaton as input and analyses it with respect to certain properties. Well-known examples are algorithms that decide whether the language represented by a tree recognizer or tree grammar is empty or whether it is finite (cf., e.g., [DE98]).

A synthesizer is an algorithm that takes zero or more tree automata (and maybe some additional data) as input and yields a tree automaton as output. There are various important types of synthesizers:

- A generator is an algorithm that outputs tree automata without requiring other tree automata as input. A prominent example is given by grammatical inference algorithms for tree automata. These are algorithms whose purpose it is to "learn" tree languages. For this, the algorithm is provided with some source of information regarding the tree language (or tree series) to be learned, such as positive and negative examples. The algorithm is then expected to construct a tree automaton representing the tree language in question. See, e.g., the references in [Dre09a] for a variety of approaches.

  Conceptually, a tree automaton A may be considered as a generator that outputs the constant value A.

- Conversion algorithms take a tree automaton as input and yield another tree automaton as output, usually with the same semantics as the input automaton. Well-known examples are conversions between regular tree grammars and finite-state tree recognizers and algorithms that minimize tree automata, make them deterministic, remove useless states or nonterminals, etc (see, e.g., [CDG+07]). There are also conversion algorithms that do not retain the semantics of the tree automaton they are applied to. For example, a macro tree transducer *mtt* may be turned into a finite-state tree recognizer that accepts the pre-image of the tree transformation computed by *mtt*. A conversion algorithm that inverts suitable types of top-down tree transducers would be another example.

- Composition algorithms turn n tree automata ($n > 1$) into one. A wealth of such algorithms can be found in the literature. One type of example is, of course, given by composition in the strict sense. For instance, certain types of tree transductions are known to be closed under composition. Another example is the main result of [DE04], which provides an algorithm for converting a regular tree grammar g and n top-down tree transducers $td_1, \ldots, td_n$ into a branching synchronization tree grammar generating the image of $L(g)$ under $td_n \circ \cdots \circ td_1$. Composition algorithms in a more general sense may not perform mathematical composition, but combine tree automata in a different way. For example, two finite-state tree recognizers can be turned into one that recognizes the intersection of the tree languages recognized by the two individual automata.

Finally, decomposition algorithms are the conceptual inverse of composition algorithms, turning one tree automaton into several others. For example, for $\{x,y\} = \{\text{top-down}, \text{bottom-up}\}$, every x tree transducer may be decomposed into two y tree transducers [Eng75]. A similar example is given by the result that every deterministic total macro tree transducer may be decomposed into a top-down tree transducer followed by a YIELD mapping [EV85].

Of course, algorithms on tree automata may additionally be classified by the types of tree automata they work on, similarly to the way in which tree automata can be classified.

4 A Proposed Attribute Type System for MARBLES

As mentioned earlier, the goal behind the development of MARBLES is that it should allow its user to assemble configurations of tree automata algorithms in a similar way as TREEBAG allows its user to assemble various sorts of tree-based generators. In particular, there should be a way to load components representing (tree automata and) tree automata algorithms, establish a data-flow relation between them, and execute them. However, while TREEBAG comes with a fixed set of component types, something like this is neither possible nor desirable for MARBLES. In contrast, users should be given the possibility to define and implement their own classes of tree automata algorithms and experiment with them. The following two fictitious scenarios try to illustrate this.[7]

Scenario 1: Test Environment for Minimization Algorithms.

Doctoral student X works in a research group using bottom-up tree recognizers for model checking purposes. A typical example is the verification of a process communication protocol P by generating a tree recognizer A_P that models the system behavior P causes, and then analyzing A_P to establish P's correctness. The problem is that A_P tends to be unnecessarily huge, so that its analysis takes too much time. Unfortunately, A_P is also nondeterministic, which means that it cannot efficiently be minimized.

Therefore, in her thesis, X proposes and studies a number of efficient heuristics for reducing nondeterministic tree recognizers A in size (called minimization, for simplicity). The general technique used is to compute a suitable equivalence $\equiv$ on the state set of A, such that the quotient automaton $A/\equiv$ accepts the same language as A. The various heuristics studied differ only in the concrete definition (and computation) of $\equiv$. Besides studying the minimization algorithms theoretically to establish their correctness and worst case complexity, X wants to study empirically how they behave on real examples arising in the model checking context, in terms of size reduction and efficiency. However, X does not have the time to implement a test environment for her algorithms from scratch, in addition to her theoretical studies. Therefore, she decides to use MARBLES.

First, she notices that there is a type of tree automata algorithm called generator, a special type of synthesizer. She defines and implements a simple generator which lets the user choose the name of a protocol (from a fixed set of possible choices) and possibly some other parameters.

[7] While being fictitious, the scenarios have a real background, as they are inspired by [Kaa08] and ongoing work in our own group, resp.

The generator will then output nondeterministic bottom-up tree recognizers of increasing size, whenever the user presses a certain button.

Next, X discovers that there are so-called converters, and decides to implement a new type of converter as an abstract class. A concrete implementation is obtained by providing a method that, for a given bottom-up tree recognizer A, computes an equivalence relation $\equiv$ on the states. The converter will then return $A/\equiv$.

Fortunately, X finds out that someone else has already implemented two useful auxiliary components. One of them is a wrapper for arbitrary converters that simply executes them, but also reports how much time the execution takes. The other one takes bottom-up tree recognizers as input and saves some statistics about them to a file, such as the number of states and transitions. Now, X has everything needed to make the desired tests. All she has to do is to implement the different algorithms yielding the equivalence relations $\equiv$, load and interconnect the required components, and execute them.

Scenario 2: Simulation of Minimal Adequate Teachers Using Corpora.

The research group in which researcher Y is working has previously studied grammatical inference algorithms that, within Angluin's learning model of a *minimal adequate teacher* (MAT), construct a bottom-up tree recognizer for a recognizable tree languages L. Now, they want to find out whether such an algorithm can be used to learn the syntax of natural languages reasonably well, where the necessary data is taken from a corpus.[8]

The major obstacle is the MAT, an oracle capable of answering two types of queries, namely membership queries (*Is the tree t in L?*) and equivalence queries (*Does the bottom-up tree recognizer A satisfy $L(A) = L$? If not, return a counterexample.*) Clearly, a MAT is not available in the situation sketched above. The research question is whether it can (imperfectly) be simulated on the basis of a corpus, so that the inference algorithm as a whole runs with reasonable efficiency and yields acceptable results.

Y decides to try out some approaches and to use MARBLES for that purpose. Thus, she defines two new types of algorithms, namely MATs and learners. A learner is a generator that must be connected to a MAT to create a tree automaton. During the first phase, she only wants to test different realizations of the MAT, to see whether the results are promising enough to continue. Therefore, she implements a single learner (e.g., any of those in [Dre09a]). In contrast, a variety of different MATs are implemented, using different approaches for answering membership and equivalence queries based on a corpus.

To find out how good the various approaches are, Y implements a component that has access to a sufficiently large sample of positive and negative examples. It takes a tree recognizer as input, runs it on the samples, and returns statistics regarding its sensitivity and specifitivity. In a second phase of her research work, Y even wants to study other variants of the learner, which can be done in the same setting by replacing one learner with another.

In scenarios such as those above, the researcher who wants to use MARBLES must implement certain extensions, new types of tree automata and algorithms that become components of MARBLES. The central idea behind MARBLES is that the developer can easily tell the system

[8] A corpus is a manually analyzed and annotated database of sentences in a natural language.

how the implemented extensions can be used and, in particular, how the new components can be combined with others. Thus, there must be a possibility to describe the types of components in an easy and flexible way. Rather than defining a strict typing system, the goal is to enable developers to communicate the major properties of new components to both MARBLES and its users. Thus, one needs a way to name the basic data types and relate them with each other. The solution proposed here is an extensible hierarchy of attributes. To see where the hierarchy comes in, consider the case of binary ordered trees. The property of being binary should be expressed by an appropriate attribute, for example, by giving the attribute *uniformRank* the value *binary*. However, since binary trees are essentially a special case of ranked trees, one may wish to reserve this attribute for ranked trees, which means that one should be allowed to specify *uniformRank* only under the premise that the tree is ranked, which may be indicated by the attribute *ranked* having the value *true*. Independently of whether or not the tree is ranked, it may be ordered or not. The latter could be expressed by assigning the attribute *ordered* the value *true* or *false*. However, both *ranked* and *ordered* make sense only for trees. Thus, their premise could be that the attribute *class* associated with the data structure in question has the value *tree*. This attribute may be an "outermost" one, meaning that it does not have a premise and can, thus, be the root of the hierarchical structure. In summary, binary ordered trees could be designated a tree of attribute assignments of the form

$$tree^{class}[true^{ranked}[binary^{uniformRank}], true^{ordered}],$$

designating a basic type in MARBLES. The following definition formalizes this notion.

Definition 1 (attribute trees and basic types)

1. Let *ATTR* be a finite set of *attributes a*, each having a finite set $V(a) = \{v_1, \ldots, v_n\}$ of possible values. For $v \in V(a)$, let v^a denote the assignment of v to a. The set of all such assignments is denoted $ASS(ATTR)$.

2. For all $a \in ATTR$, assume that an attribute assignment $prem(a) \in ASS(ATTR) \cup \{\bot\}$, called the *premise* of a, is specified. The set of all *attribute trees* is defined inductively, as follows.

   (a) Every $v^a \in ASS(ATTR)$ is an attribute tree with root attribute a.

   (b) For all attribute trees $t_1, \ldots, t_n$ whose root attributes are pairwise distinct and have the same premise v^a, $v^a[t_1, \ldots, t_n]$ is an attribute tree with root attribute a.

3. An attribute tree whose root attribute has the premise $\bot$ is a *basic type*. The set of basic types is denoted by *basic*.

It should be noted that attribute trees (and, thus, basic types) are loose specifications in the sense that they do not generally refer to a specific data type. The rationale behind this is that it should be possible to specify only those attributes that are of interest in a given situation. For example, consider a class of algorithms working on, e.g., tree grammars. If it is essential that the trees generated by these tree grammars are binary ordered trees, then the basic type discussed above may be appropriate. However, if the algorithms work on any type of ranked trees, ordered or not, then the more general basic type $tree^{class}[true^{ranked}]$ is more appropriate.

Note that no specific semantics or implementation is associated with the attributes. It should, however, be *possible* to do this in MARBLES by, e.g., associating an attribute with formal semantic requirements, or with an abstract class in the implementation. How this can be done in the most appropriate way is an interesting topic for future work. A similar remark applies to the types at the higher levels discussed next.

The next definition concerns automaton types. It takes a very general approach, where an automaton is a device that turns a finite number of input values of specified basic types and into a finite number of output values, also of specified basic types.

Definition 2 (automaton type) An *automaton type* is a pair (in, out) with $in \in basic^k$ and $out \in basic^l$ for some $k, l \geq 0$. Such a type will normally be written as $in \to out$. The set of all automaton types is denoted by AUT.

As an example, a tree grammar of the most general form could be described as being an automaton of type $() \to (tree^{class})$. In other words, it takes no input and yields any type of tree as output. Note that such an automaton type does not say much about how the actual implementation of an automaton behaves, which kinds of operations it provides, and so on. For example, reasonable types of tree grammars are always nondeterministic. Implementations should therefore provide a means to enumerate the generated trees or nondeterministically generate one. Automaton types may specify such details if necessary, but they need not. In the most specific case, an automaton type may be associated with a particular class in the implementation. The level of detail used may vary depending on the situation.

Slightly more specific than the type $() \to (tree^{class})$ would be the type of tree grammars generating ranked trees, namely $() \to (tree^{class}\langle true^{ranked}\rangle)$. For weighted tree automata over a semifield that work on ranked trees, the type $(tree^{class}\langle true^{ranked}\rangle) \to (semiring^{class}\langle hasInverses^{prop}\rangle)$ could be an appropriate description, and for tree transducers on unranked trees one could use $(tree^{class}\langle false^{ranked}\rangle) \to (tree^{class}\langle false^{ranked}\rangle)$. Though uncommon in the literature, one may also wish to consider, e.g., tree transducers that take two trees as input and produce one output tree, the corresponding type being $(tree^{class}, tree^{class}) \to (tree^{class})$.

Note that the concept is very general. For example, an algebra can be seen as an automaton of type $(tree^{class}\langle true^{ranked}\rangle) \to (any^{class})$, if any^{class} is assumed to be the most general basic type, standing for arbitrary data. Also weighted tree automata over multioperator monoids [Kui00] have this type. Clearly, the concept covers even devices that do not work on trees at all. Thus, in principle, MARBLES may even be extended to automata on other structures, such as graphs.

Finally, the next definition makes it possible to specify types of algorithms on tree automata.

Definition 3 (algorithm type) The set ALG of *algorithm types* is inductively defined to be the smallest set containing all triples (in, use, out) such that, for some $k, l, m \geq 0$, $in \in AUT^k$, $out \in AUT^m$, and $use \in ALG^l$. Such a triple is denoted by $in \xrightarrow{use} out$.

The intuitive interpretation of $in \xrightarrow{use} out$ is that of an algorithm which turns inputs according to *in* into outputs according to *out*, thereby possibly making use of other algorithms given by *use*. A typical example is the MAT learner in Scenario 2, which could be of the type $() \xrightarrow{MAT} (TA)$, where TA is the automaton type $tree^{class}\langle true^{ranked}\rangle \to bool^{class}$.

As mentioned earlier, one of the ideas behind MARBLES is that its GUI, similar to the one of TREEBAG, should allow the user to assemble configurations of tree automata in order to experiment with them. The basic (and still somewhat tentative) plan is that every implementation of a class of tree automata or tree automata algorithms comes with a specified type according to the definitions above. When the user loads an instance of such a component, this information is used in order to determine which connections between these components are possible. The idea is that an algorithm of the type in Definition 3 will, from the point of view of the user, have $k+l+m$ slots representing the inputs, the used algorithms, and the outputs. For instance, if a component has an output slot s of type $\mathit{tree}^{\mathit{class}}\langle \mathit{true}^{\mathit{ranked}}\rangle \to \mathit{bool}^{\mathit{class}}$ (a recognizer for ranked trees) and another one has an input slot s' of type $\mathit{tree}^{\mathit{class}} \to \mathit{bool}^{\mathit{class}}$ (a recognizer for any sort of trees), then the data flow can be directed from s to s'.

5 Concluding Remarks

In this paper, ideas and plans regarding a successor of the system TREEBAG have been presented. While this work is still in a very preliminary phase, the overall goal is clear. MARBLES should make it possible to experiment with configurations of tree automata algorithms in a similar way as TREEBAG makes it possible to experiment with tree-based generators. Moreover, MARBLES should be extensible by researchers who are not directly involved in the development of the system itself, but want to use it for their own purposes. For this, concepts such as those presented in Section 4 seem to be a necessity, because the GUI must be able to handle extensions without explicitly being adapted.

An aspect that has not been discussed in the present paper, but which is a necessity as well, is to provide programmers with a well-documented library and a clearly structured application programmer's interface (API). Without this, it would be too difficult, error prone, and time consuming for other researchers to make their own extensions. In fact, it should also be possible to make use of the API without adopting the rest of MARBLES, and especially its GUI. This would give programmers the possibility to apply tree automata algorithms in their own applications. Another aspect that has not yet been decided upon is whether and to what extent MARBLES shall be compatible and able to interoperate with other systems dealing with tree automata, such as those mentioned in Section 1.

Acknowledgements: I thank the anonymous referees for their unusually thorough reading of the manuscript, resulting in many appreciated comments, suggestions, and corrections.

Most of all, however, I want to thank you, Hans-Jörg, for your support during all those years, for teaching me so much about our profession, and for being a good example in all respects. I wish you many more years of continued scientific productivity to come, and want to conclude this paper by paraphrasing the final sentence you once formulated in our survey paper [DK99]: Whichever serious goals one aims at in the investigation of theoretical formalisms, they can also be great fun!

Bibliography

[AJMd02] P. A. Abdulla, B. Jonsson, P. Mahata, J. d'Orso. Regular Tree Model Checking. In Brinksma and Larsen (eds.), *Proc. 14th Intl. Conf. on Computer Aided Verification (CAV'02)*. Lecture Notes in Computer Science 2404, pp. 555–568. 2002.

[Bor04] B. Borchardt. Code Selection by Tree Series Transducers. In Domaratzki et al. (eds.), *Proc. 9th Intl. Conf. on Implementation and Application of Automata (CIAA 2004)*. Lecture Notes in Computer Science 3317, pp. 57–67. Springer, 2004.

[CDG+07] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, C. Löding, D. Lugiez, S. Tison, M. Tommasi. *Tree Automata Techniques and Applications*. Internet publication available at http://tata.gforge.inria.fr, 2007. Release October 2007.

[CF82] B. Courcelle, P. Franchi-Zannettacci. Attribute Grammars and Recursive Program Schemes I, II. *Theoretical Computer Science* 17:163–191, 235–257, 1982.

[DE98] F. Drewes, J. Engelfriet. Decidability of the Finiteness of Ranges of Tree Transductions. *Information and Computation* 145:1–50, 1998.

[DE04] F. Drewes, J. Engelfriet. Branching Synchronization Grammars with Nested Tables. *Journal of Computer and System Sciences* 68:611–656, 2004.

[DEKK03] F. Drewes, S. Ewert, R. Klempien-Hinrichs, H.-J. Kreowski. Computing Raster Images from Grid Picture Grammars. *Journal of Automata, Languages and Combinatorics* 8:499–519, 2003.

[DHK97] F. Drewes, A. Habel, H.-J. Kreowski. Hyperedge Replacement Graph Grammars. In Rozenberg (ed.), *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 1: Foundations*. Chapter 2, pp. 95–162. World Scientific, Singapore, 1997.

[DHKT95] F. Drewes, A. Habel, H.-J. Kreowski, S. Taubenberger. Generating self-affine fractals by collage grammars. *Theoretical Computer Science* 145:159–187, 1995.

[DK96] F. Drewes, H.-J. Kreowski. (Un-)Decidability of Geometric Properties of Pictures Generated by Collage Grammars. *Fundamenta Informaticae* 25:295–325, 1996.

[DK99] F. Drewes, H.-J. Kreowski. Picture generation by collage grammars. In Ehrig et al. (eds.), *Handbook of Graph Grammars and Computing by Graph Transformation*, Vol. 2: *Applications, Languages, and Tools*. Chapter 11, pp. 397–457. World Scientific, Singapore, 1999.

[Dre96] F. Drewes. Language Theoretic and Algorithmic Properties of d-dimensional Collages and Patterns in a Grid. *Journal of Computer and System Sciences* 53:33–60, 1996.

[Dre00] F. Drewes. Tree-Based Picture Generation. *Theoretical Computer Science* 246:1–51, 2000.

[Dre06] F. Drewes. *Grammatical Picture Generation – A Tree-Based Approach*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2006.

[Dre09a] F. Drewes. MAT Learners for Recognizable Tree Languages and Tree Series. *Acta Cybernetica* 19:249–274, 2009.

[Dre09b] F. Drewes. Towards the Tree Automata Workbench MARBLES. In Drewes et al. (eds.), *Manipulation of Graphs, Algebras and Pictures. Essays Dedicated to Hans-Jörg Kreowski on the Occasion of His 60th Birthday*. Pp. 83–98. 2009.

[Eng75] J. Engelfriet. Bottom-up and Top-down Tree Transformations – a Comparison. *Mathematical Systems Theory* 9:198–231, 1975.

[Eng80] J. Engelfriet. Some open questions and recent results on tree transducers and tree languages. In Book (ed.), *Formal Language Theory: Perspectives and Open Problems*. Pp. 241–286. Academic Press, New York, 1980.

[Eng94] J. Engelfriet. Graph Grammars and Tree Transducers. In Tison (ed.), *Proceedings of the CAAP 94*. Lecture Notes in Computer Science 787, pp. 15–37. Springer, 1994.

[EV85] J. Engelfriet, H. Vogler. Macro Tree Transducers. *Journal of Computer and System Sciences* 31:71–146, 1985.

[FGV04] G. Feuillade, T. Genet, V. Viet Triem Tong. Reachability Analysis over Term Rewriting Systems. *Journal of Automated Reasoning* 33:341–383, 2004.

[FSW94] C. Ferdinand, H. Seidl, R. Wilhelm. Tree Automata for Code Selection. *Acta Informatica* 31(8):741–760, 1994.

[FV98] Z. Fülöp, H. Vogler. *Syntax-Directed Semantics: Formal Models Based on Tree Transducers*. Springer, 1998.

[FV09] Z. Fülöp, H. Vogler. Weighted Tree Automata and Tree Transducers. In Kuich et al. (eds.), *Handbook of Weighted Automata*. Chapter 9, pp. 313–403. Springer, 2009.

[GK00] T. Genet, F. Klay. Rewriting for Cryptographic Protocol Verification. In McAllester (ed.), *Proc. 17th International Conference on Automated Deduction (CADE'00)*. Lecture Notes in Computer Science 1831, pp. 271–290. 2000.

[GKM08] J. Graehl, K. Knight, J. May. Training Tree Transducers. *Computational Linguistics* 34(3):391–427, 2008.

[GS84] F. Gécseg, M. Steinby. *Tree Automata*. Akadémiai Kiadó, Budapest, 1984.

[GS97] F. Gécseg, M. Steinby. Tree Languages. In Rozenberg and Salomaa (eds.), *Handbook of Formal Languages*. Vol. 3: *Beyond Words*. Chapter 1, pp. 1–68. Springer, 1997.

[HKL93] A. Habel, H.-J. Kreowski, C. Lautemann. A comparison of compatible, finite, and inductive graph properties. *Theoretical Computer Science* 110:145–168, 1993.

[HKT93] A. Habel, H.-J. Kreowski, S. Taubenberger. Collages and Patterns Generated by Hyperedge Replacement. *Languages of Design* 1:125–145, 1993.

[HKV91] A. Habel, H.-J. Kreowski, W. Vogler. Decidable Boundedness Problems for Sets of Graphs Generated by Hyperedge-Replacement. *Theoretical Computer Science* 89:33–62, 1991.

[Kaa08] L. Kaati. Reduction Techniques for Finite (Tree) Automata. Doctoral dissertation, Uppsala University, Sweden, 2008.

[KG05] K. Knight, J. Graehl. An Overview of Probabilistic Tree Transducers for Natural Language Processing. In Gelbukh (ed.), *Proc. 6th Intl. Conf. on Computational Linguistics and Intelligent Text Processing (CICLing 2005)*. Lecture Notes in Computer Science 3406, pp. 1–24. Springer, 2005.

[Kui00] W. Kuich. Linear systems of equations and automata on distributive multioperator monoids. In Dorninger et al. (eds.), *Proc. 58th Workshop on General Algebra (1999)*. Contributions to General Algebra 12, pp. 247–256. Johannes Heyn, Klagenfurt, 2000.

[Löd02] C. Löding. Model-Checking Infinite Systems Generated by Ground Tree Rewriting. In Nielsen and Engberg (eds.), *Proc. 5th Intl. Conf. on Foundations of Software Science and Computation Structures (FOSSACS'02)*. Lecture Notes in Computer Science 2303, pp. 280–294. 2002.

[MW67] J. Mezei, J. B. Wright. Algebraic Automata and Context-Free Sets. *Information and Control* 11:3–29, 1967.

[NP92] M. Nivat, A. Podelski (eds.). *Tree Automata and Languages*. Elsevier, Amsterdam, 1992.

[Sch07] T. Schwentick. Automata for XML - A Survey. *Journal of Computer and System Sciences* 73(3):289–315, 2007.