HAL open science

Random Generation of Positive TAGEDs wrt. the Emptiness Problem

Pierre-Cyrille Heam, Vincent Hugot, Olga Kouchnarenko

▶ To cite this version:

Pierre-Cyrille Heam, Vincent Hugot, Olga Kouchnarenko. Random Generation of Positive TAGEDs wrt. the Emptiness Problem. [Research Report] RR-7441, INRIA. 2010, pp.43. inria-00531350

HAL Id: inria-00531350

https://inria.hal.science/inria-00531350v1

Submitted on 2 Nov 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Random Generation of Positive TAGEDs wrt. *the Emptiness Problem*

Pierre-Cyrille Héam — Vincent Hugot — Olga Kouchnarenko

N° 7441

October 2010

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--7441--FR+ENG

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE
INRIA
centre de recherche NANCY - GRAND EST

Random Generation of Positive TAGEDs *wrt.* the Emptiness Problem

Pierre-Cyrille Héam* , Vincent Hugot† , Olga Kouchnarenko‡

Thème : Programmation, vérification et preuves
Équipe-Projet CASSIS

Rapport de recherche n° 7441 — October 2010 — 43 pages

Abstract: Tree automata are a widely used formalism in Computer Science. Since their creation in the fifties, numerous more expressive extensions have been proposed. Unfortunately, the decision problems associated with these extensions are quite often undecidable or in prohibitive classes of algorithmic complexity (*NP*-complete or worse), and little work has gone into finding efficient heuristics for them. Beyond the inherent difficulty of those problems, a common hitch in this line of research is the experimental evaluation of new algorithms. As those extensions of tree automata have remained in chiefly theoretical spheres, there are no established testbeds from the "real world" against which to quantify the efficiency (or lack thereof) of new algorithms. Failing that, there is a need to generate suitable testbeds at random. Regrettably, there is little material in the literature regarding random generation of tree automata, and none at all regarding extensions such as Tree Automata with Global Equality and Disequality Constraints (`TAGED`s). It should also be noted that what little material there is does not concern itself with the *interest* of the generated automata *wrt.* specific decision problems. In this report we present a scheme for random generation of positive `TAGED`s, with a focus on making them interesting *wrt.* the Emptiness problem.

Key-words: Tree automata with constraints, `TAGED`, Membership problem, Emptiness problem, Random generation, Difficult cases

This work is partly supported by INRIA Collaborative Research Action (ARC) "ACCESS", `http://acxml.gforge.inria.fr/`

* University of Franche-Comté and INRIA/Cassis
† University of Franche-Comté and INRIA/Cassis
‡ University of Franche-Comté and INRIA/Cassis

Centre de recherche INRIA Nancy – Grand Est
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

Génération Aléatoire de TAGEDs Positifs pour le Problème de Vacuité

Résumé : Les automates d'arbres sont un formalisme très utilisé en informatique. Depuis leur création dans les années cinquante, de nombreuses extensions, plus expressives, ont été créées, principalement pour résoudre des problèmes théoriques. Malheureusement, les problèmes de décision associés à ces extensions sont le plus souvent indécidables ou dans des classes de complexité prohibitives (*NP*-complet ou pire), et peu de recherches ont été effectuées pour leur trouver des heuristiques efficaces. Mis à par la difficulté inhérente de ces problèmes, de telles recherches se heurtent souvent à la nécessité d'une évaluation expérimentale des nouveaux algorithmes. Étant donné que ces extensions des automates d'arbres restent à l'heure actuelle principalement théoriques, il n'existe pas de bancs d'essai établis à l'aide desquels l'efficacité de nouvelles approches pourrait être évaluée et optimisée. Faute de cela, des bancs d'essai adéquats doivent être générés aléatoirement. Malheureusement, la littérature concernant la génération aléatoire d'automates d'arbres n'est pas abondante, voire inexistante dans le cas d'extensions telles que les automates d'arbres avec des contraintes globales d'égalité et de différences (TAGEDs). Il est également à noter que les travaux existants ne s'intéressent pas à la difficulté des instance générées par rapport à des problèmes de décision spécifiques. Dans ce rapport nous présentons une méthode pour générer des TAGEDs positifs aléatoires, axée sur la difficulté des instances générées par rapport au problème de décision du vide.

Mots-clés : Automates d'arbres avec contraintes, TAGED, Problème d'appartenance, Problème de vacuité, Génération aléatoire, Instances difficiles

Contents

Contents 3

List of Figures 4

1 Introduction and motivation 5

2 Preliminaries 6
- 2.1 Bottom-up Nondeterministic Finite Tree Automata . . . 7
  - 2.1.1 Symbols, trees, terms and subterms . . . 7
  - 2.1.2 Tree automata . . . 8
- 2.2 Some extensions . . . 11
  - 2.2.1 The class AWEDC (1981) . . . 11
  - 2.2.2 Subclasses of AWEDC (≈ 1990) . . . 12
  - 2.2.3 `TAGED` (2006) . . . 12
- 2.3 The Emptiness decision problem . . . 14

3 Objectives and Strategy 15

4 Getting rid of some obvious cases and dead branches 18
- 4.1 Detecting "spurious" states and rules . . . 19
- 4.2 Example and conclusion . . . 26

5 Generating raw `TAGED`s 27
- 5.1 Generating the underlying tree automaton . . . 28
  - 5.1.1 Related work and previous attempts . . . 28
  - 5.1.2 Outline of the height-driven algorithm . . . 33
- 5.2 Generating the global equality constraints . . . 36

6 Experiments 37

7 Conclusion and future work 40

References 41

List of Figures

1 Reduction algorithm, from [Comon et al., 2007, page 25] 21
2 Rough outline of height-driven generation 34
3 Very simple constraints generation algorithm 37

1 Introduction and motivation

Tree automata are a widely used formalism in Computer Science. Since their creation in the fifties in the context of circuit verification, they turned out to be a very convenient way for modelling and proving properties on infinite systems such as communication protocols [Boichut et al., 2008b, Boichut et al., 2008a, Boichut et al., 2009], multi-threaded Java byte code programs [Boichut et al., 2007, Courbis et al., 2009], etc. Moreover, numerous recent works on analysis of structured XML-like documents and on validation of their transformations [Schwentick, 2007, Abiteboul et al., 2009, Jacquemard and Rusinowitch, 2009, Bojanczyk et al., 2009] exploit tree automata for their encoding.

They are not quite expressive enough for all needs, though. For all above-mentioned applications, it is useful to be able to express constraints such as "in the term, two subterms do not have the same leaf". For example, when evaluating queries over XML documents, it is important to express constraints like "in the document, two nodes do not have the same key". Unfortunately, vanilla tree automata are in general not capable of conveying this kind of constraints. To this end numerous more expressive extensions of tree automata have been proposed, let us quote hedge tree automata [Murata, 1999], visibly tree automata with memory and constraints [Comon-Lundh et al., 2008], rigid tree automata [Jacquemard et al., 2009], tree automata with equality and difference constraints (*aka.* AWEDC), tree automata with constraints between brothers, and lastly tree automata with global equality and disequality constraints (`TAGED`s for short) [Filiot et al., 2008]. Although this recent work provides theoretical results promising *wrt.* practical applications, they are not yet supported by efficient verification and validation tools.

As it turns out, the implementation of such tools in a reasonably efficient manner would unfortunately be far from trivial. The main reason for this lies in that the decision problems associated with these extensions are unfortunately either undecidable or in prohibitive classes of algorithmic complexity (*NP*-complete or worse), and little work has gone into finding efficient heuristics for them. This makes it quite unlikely that any undemanding implementation could be of any practical use. In our current work, we focus on the design of algorithms for extended tree automata efficient enough for being implemented.

Beyond the inherent difficulty of those problems, we found that a common hitch in this line of research is the experimental evaluation of new algorithms. Indeed, there is only so much one can do purely "in theory" to improve efficiency. Beyond the few obvious improvements which demonstrably yield strictly better algorithms, most of the work consists in isolating special cases for which the problem can be simplified or even resolved quickly, and then it comes down to the question of whether the computational

costs involved in detecting and dealing with said cases were well-invested. And this depends solely on how frequently those cases turn up in practice.

Regrettably, as those extensions of tree automata have remained in chiefly theoretical spheres, there are no established testbeds from the "real world" against which to quantify the efficiency (or lack thereof) of new algorithms. Failing that, there is a need to generate such suitable testbeds at random. We found that there is little material in the literature regarding random generation of tree automata, and none at all regarding extensions such as Tree Automata with Global Equality and Disequality Constraints (`TAGED`s). It should also be noted that what little material there is does not concern itself with the *interest* of the generated automata *wrt.* specific decision problems. An *interesting* automaton *wrt.* a given problem should be such that the answer in not blatantly obvious, *ie.* not within the immediate reach of the most naive algorithm, and should not look absolutely unlike anything that might be expected to come up in the "real world". If this first constraint is not satisfied, then *all* algorithms one could come up with can be expected to perform well, and the results will fail to discriminate between the efficient ones and the rest. On the other hand, if the second constraint is not satisfied, the results, even if they do serve to discriminate between our algorithms, will be of questionable relevance to the performance of practical tools, dealing with real cases.

In this report we present a scheme (which we call *height-driven*) for random generation of positive `TAGED`s, with a focus on making them interesting *wrt.* the emptiness problem.

Layout of the paper: This paper is organised as follows: We offer detailed preliminaries about terms, tree automata, `TAGED`s and the emptiness problem in section 2[p6]. Section 3[p15] details the objectives of the random generation scheme, the constraints it should satisfy and the characteristics which it should avoid. This serves as a detailed introduction for the next two sections, which present in detail two aspects of the generation scheme. Section 4[p18] presents a method for cutting obvious "dead branches" off of generated `TAGED`s, while section 5[p27] quickly presents previous, unsuccessful schemes before detailing the height-driven generation algorithm. Finally, section 6[p37] presents some experimental results obtained with both past schemes – which provide a basis for comparison – and the current height-driven scheme, and section 7[p40] concludes.

2 Preliminaries

This section presents the necessary vocabulary, notations and concepts which will be used throughout the document. References for this section include notably [Comon et al., 2007, Filiot et al., 2008].

2.1 Bottom-up Nondeterministic Finite Tree Automata

2.1.1 Symbols, trees, terms and subterms

Before speaking of "tree automata", let us start by defining the notions of "tree", and that of "term". We shall see that those two notions can be considered equivalent in the context which interests us, and so we will confuse them in the remainder of the document. Let Σ be a finite set of symbols, and let $arity : \Sigma \to \mathbb{N}$ be the arity function. Intuitively, this function associates to a symbol $f \in \Sigma$ the number of "arguments" which it may take. We denote $\Sigma_n = \{ f \in \Sigma \mid arity(f) = n \}$ the set of all symbols of arity n, called "n-ary symbols", and $\mathfrak{Ar}_\Sigma = \{ k \in \mathbb{N} \mid \Sigma_k \neq \varnothing \}$ the set of all arities for which there exists at least a symbol in Σ. Whenever it is convenient, we shall denote $f/_n$ a symbol $f \in \Sigma_n$. It is assumed that the set of "constants" Σ_0 is non-empty. The couple $(\Sigma, arity)$ forms a "ranked alphabet". We will most often refer simply to the "ranked alphabet Σ" and omit the explicit mention of *arity*. We denote by $\mathcal{T}(\Sigma)$ the set of "ground terms" or more simply "terms", over the ranked alphabet Σ. It is defined as the smallest set such that

1. $\Sigma_0 \subseteq \mathcal{T}(\Sigma)$ and
2. for any $n \geqslant 1$ if $t_1, \ldots, t_n \in \mathcal{T}(\Sigma)$ then $f(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma)$, for any $f \in \Sigma_n$.

A set of words S is said to be "prefix-closed" if it is such that for any $w \in S$, all prefixes of w are also in S. A tree over a set of labels L is a mapping from a prefix-closed set $S \subseteq \mathbb{N}^*$ into L. Let $t \in \mathcal{T}(\Sigma)$; it can be seen as a tree by defining the set of "positions" $\mathcal{P}os(t)$ inductively as follows:

1. $\mathcal{P}os(t) = \{ \varepsilon \}$ if $t \in \Sigma_0$
2. $\mathcal{P}os\Big(f(t_1, \ldots, t_n)\Big) = \Big\{ \varepsilon \Big\} \cup \Big\{ i.\alpha \mid i \in [\![1, n]\!] \text{ and } \alpha \in \mathcal{P}os(t_i) \Big\}$ otherwise.

Then t is a mapping from $\mathcal{P}os(t)$ to Σ, such that leaves map to Σ_0 and nodes map to symbols of corresponding arity. As announced above, we will from now on confuse those two notions: for instance for any term $t \in \mathcal{T}(\Sigma)$ and any $\alpha \in \mathcal{P}os(t)$, $t(\alpha)$ is the symbol at position α in the term t. Note that then $\mathcal{P}os(t)$ and $\mathrm{dom}(t)$ are two different notations for the same object, but in the context of trees we will systematically prefer the former notation. There remains to define the notion of "subterm" (or "subtree"). Let $t \in \mathcal{T}(\Sigma)$ and $\alpha \in \mathcal{P}os(t)$, we denote $t|_\alpha$ the subterm of t at position α, which is defined as follows:

1. $\mathcal{P}os(t|_\alpha) = \{ \beta \mid \alpha.\beta \in \mathcal{P}os(t) \}$
2. for any $\beta \in \mathcal{P}os(t|_\alpha)$, $t|_\alpha(\beta) = t(\alpha.\beta)$.

We denote by $u \trianglelefteq t$ the fact that u is a subterm of t,*ie.* there exists $\alpha \in \mathcal{P}os(t)$ such that $u = t|_\alpha$. The relation $\trianglelefteq$ is a partial order on $\mathcal{T}(\Sigma)$. The same notation is used between

positions: for two positions $\alpha, \beta \in \mathcal{P}os(t)$, we say that "$\alpha$ is under β" and note $\alpha \trianglelefteq \beta$ the fact that β is a prefix[(a)] of α. We define the induced strict order in the usual way, *ie.* $x \triangleleft y \iff x \trianglelefteq y$ and $x \neq y$, regardless of whether x and y are terms or positions. On some limited occasions, we will need a more precise evaluation of the degree to which one term is below another, and to this purpose we will denote $\alpha \trianglelefteq_n \beta$ (*resp.* $\alpha \triangleleft_n \beta$) the fact that $\alpha \trianglelefteq \beta$ (*resp.* $\alpha \triangleleft \beta$) and $|\alpha| - |\beta| = n$, for $n \in \mathbb{N}$ (*resp.* $n \in \mathbb{N}^*$). Note that $\alpha \trianglelefteq_0 \beta \iff \alpha = \beta$ and for all $n \geqslant 1$, $\alpha \triangleleft_n \beta \iff \alpha \trianglelefteq_n \beta$. In the vernacular, $\alpha \triangleleft_1 \beta$ means that α is a direct child of β, $\alpha \triangleleft_2 \beta$ means that α is a grand-child of β and so on. We have obviously $(\trianglelefteq) = \bigcup_{n\in\mathbb{N}} (\trianglelefteq_n)$ and $(\triangleleft) = \bigcup_{n\in\mathbb{N}^*} (\triangleleft_n)$.

Terms are quite often represented graphically. For instance

$$t \overset{\text{def}}{=} \begin{array}{l} f \\ \quad f \\ \quad\quad a \\ \quad\quad a \\ \quad g \\ \quad\quad a \\ \quad\quad b \end{array}$$

represents the term $t = f(f(a,a), g(a,b))$, with $a \in \Sigma_0$ and $f, g \in \Sigma_2$. If we add the positions of $\mathcal{P}os(t)$ as subscripts we get

$$t \overset{\text{def}}{=} \begin{array}{l} f_\varepsilon \\ \quad f_1 \\ \quad\quad a_{11} \\ \quad\quad a_{12} \\ \quad g_2 \\ \quad\quad a_{21} \\ \quad\quad b_{22} \end{array}.$$

Then we see that we have clearly, for instance $t(1) = f$ and $t|_2 = \begin{array}{l} g \\ \quad a \\ \quad b \end{array}$.

2.1.2 Tree automata

Before introducing tree automata formally, let us just state that they can be seen as an extension of finite state machines, which the reader is assumed to be familiar with. Indeed, a word can be seen as a term: let us take for instance the word $w = abc$. Then one can take the ranked alphabet $\Sigma = \{a/_1, b/_1, c/_1, \#/_0\}$ and build the corresponding unary term $t_w = a(b(c(\#)))$. Tree automata have the same expressiveness over unary terms as finite state machines have over words. The difference is that they recognise tree languages instead of just word languages, which makes them much more general.

A "non-deterministic finite tree automaton" (NFTA) over a ranked alphabet Σ is a tuple $\mathcal{A} = (\Sigma, Q, F, \Delta)$ where

- Σ is the ranked alphabet
- Q is a set of states, which we will see as constant symbols. Of course, states and standard symbols must not mix: $\Sigma \cap Q = \varnothing$. The set $\mathcal{T}(\Sigma \cup Q)$ is called the "set of configurations".

[(a)] Note that this is the reverse notation from that taken in [Filiot et al., 2008], which may be confusing to some readers. The advantage of choosing this notation the way we have done it is that the symbol "$\trianglelefteq$" remains consistent with respect to terms and positions, *ie.* for all $\alpha, \beta \in \mathcal{P}os(t)$, we have $\alpha \trianglelefteq \beta \implies t|_\alpha \trianglelefteq t|_\beta$.

- $F \subseteq Q$ is the subset of "final" states
- Δ is a set of transition rules.

The rules of Δ define a "ground rewrite system" on $\mathcal{T}(\Sigma \cup Q)$. They are of the form

$$f(q_1, \ldots, q_n) \to q \quad \text{with } q, q_1, \ldots, q_n \in Q \text{ and } f \in \Sigma_n.$$

Thus, a tree automaton over Σ runs on ground terms over Σ, starting with the leaves. Indeed, rules for leaves are of the form $a \to q$, and can be considered "initial". The reader will have noticed that no set of initial states has been defined... We denote $\to_\Delta$ the rewriting relation, called "move relation" induced by Δ over $\mathcal{T}(\Sigma \cup Q)$, and $\to_\Delta^*$ its transitive reflexive closure. A term $t \in \mathcal{T}(\Sigma)$ is "accepted" by $\mathcal{A}$ if and only if there exists a final state $q_f \in F$ such that $t \to_\Delta^* q_f$. The "recognised tree language $\mathcal{L}\text{ng}(\mathcal{A})$" of $\mathcal{A}$ is the set of all accepted terms:

$$\mathcal{L}\text{ng}(\mathcal{A}) \stackrel{\text{def}}{=} \left\{ t \in \mathcal{T}(\Sigma) \mid \exists q_f \in F : t \to_\Delta^* q_f \right\}.$$

Or one can equivalently use the alternative definitions

$$\mathcal{L}\text{ng}(\mathcal{A}, q) \stackrel{\text{def}}{=} \left\{ t \in \mathcal{T}(\Sigma) \mid t \to_\Delta^* q \right\} \quad \text{and} \quad \mathcal{L}\text{ng}(\mathcal{A}) \stackrel{\text{def}}{=} \bigcup_{q_f \in F} \mathcal{L}\text{ng}\left(\mathcal{A}, q_f\right).$$

Note that the move relation, such as we have defined it, destroys the term t until only one state is left. In order to keep track of the moves which led to this result, that is to say, of the states the different subterms evaluated to, another notion is needed. We call "run of $\mathcal{A}$ on t" a mapping $\rho : \mathcal{P}os(t) \to Q$ (in other words, a tree) compatible with the rules of Δ. That is to say, for every position $\alpha \in \mathcal{P}os(t)$, if $t(\alpha) = f \in \Sigma_n$, $\rho(\alpha) = q$ and $\forall i \in [\![1, n]\!] : \rho(\alpha.i) = q_i$, then there must exist some rule $f(q_1, \ldots, q_n) \to q \in \Delta$. A run ρ is said to be "accepting" or "successful" if $\rho(\varepsilon) \in F$. It follows that a term $t \in \mathcal{T}(\Sigma)$ is accepted by $\mathcal{A}$ if and only if there exists a successful run ρ of $\mathcal{A}$ on t. This is the definition which we will use most throughout this document.

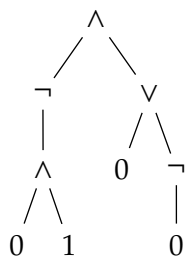
Let us take a very classical example:

$$\mathcal{A} \stackrel{\text{def}}{=} \left(\Sigma = \{\wedge, \vee/_2, \neg/_1, 0, 1/_0\},\ Q = \{q_0, q_1\}, F = \{q_1\}, \Delta\right)$$

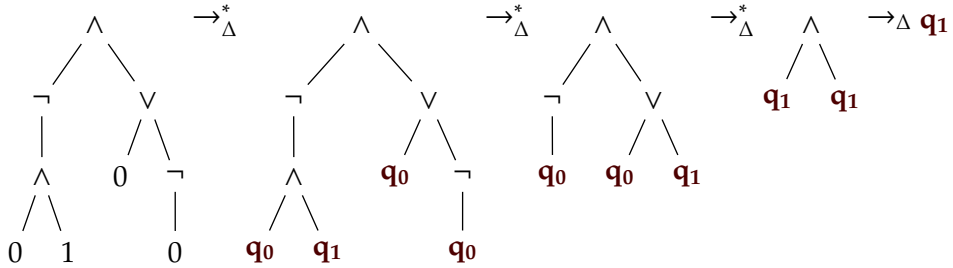
where the transition rules correspond closely to the usual rules of propositional logic:

$$\Delta = \left\{ b \to q_b,\ \wedge(q_b, q_{b'}) \to q_{b \wedge b'},\ \vee(q_b, q_{b'}) \to q_{b \vee b'},\ \neg(q_b) \to q_{\neg b} \mid b, b' \in \{0, 1\} \right\}.$$

For instance, $\wedge(q_0, q_1) \to q_0 \in \Delta$. Then if we consider the following term t



it can be rewritten with the help of the transition rules:

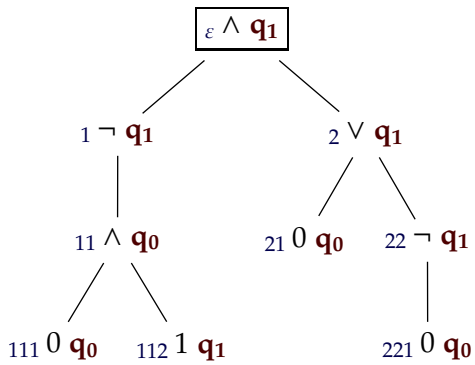


Note that each of the three first transformations above make use of several rules at once, as it would have been somewhat tedious to separate each and every step. Here is a breakdown of the rules which were used at each step of the transformation:

1. $0 \to q_0, 1 \to q_1 \in \Delta$
2. $\wedge(q_0, q_1) \to q_0, \neg(q_0) \to q_1 \in \Delta$
3. $\neg(q_0) \to q_1, \vee(q_0, q_1) \to q_1 \in \Delta$
4. $\wedge(q_1, q_1) \to q_1 \in \Delta$

This lengthy transformation $t \to^*_\Delta q_1$ can also be summarised more conveniently by the run ρ:

$\rho =$



Since $\rho(\varepsilon) = q_1 \in F$, the term t is accepted by $\mathcal{A}$. Of course, $\mathcal{A}$ recognises all true propositional formulæ, coded as trees.

Before moving on to extensions of tree automata, let us give in table 1[p11] a survey of common decision problems and their complexity, in both non-deterministic and deterministic cases, and state some closure properties.

Theorem 1 (Closure properties of tree automata)**.** *The class of recognisable tree languages is closed under union, under complementation, and under intersection.*

Decision Problem	NFTA	DFTA
Emptiness	Linear time	Linear time
Equivalence	*EXPTIME*-complete	$O(\|\|\mathcal{A}_1\|\| \times \|\|\mathcal{A}_2\|\|)$
Finiteness	Polynomial	Polynomial
Inclusion	*EXPTIME*-complete	*EXPTIME*-complete
Intersection non-emptiness	*EXPTIME*-complete	*EXPTIME*-complete
Membership	*ALOGTIME*-complete	$O(\|\|t\|\|)$
Singleton set	Polynomial	Polynomial
Uniform Membership	$O(\|\|t\|\| \times \|\|\mathcal{A}\|\|)$	$O(\|\|t\|\| + \|\|\mathcal{A}\|\|)$
Universality	*EXPTIME*-complete	Polynomial

Table 1 — Decision problems for tree automata

2.2 Some extensions

An aspect which is lacking in vanilla tree automata is testing equality and enforcing difference of some subterms. For instance, the language $\{ f(t,t) \mid f \in \Sigma, t \in \mathcal{T}(\Sigma) \}$ is non-regular, meaning that there is no tree automaton which recognises it. The same goes for $\{ f(t,t') \mid f \in \Sigma, t, t' \in \mathcal{T}(\Sigma) : t \neq t' \}$. Many extensions to tree automata have been proposed over the years to deal with such constraints, the challenge being to add enough expressiveness to solve whatever problem was at hand, while preserving reasonable decidability and complexity, good closure properties etc... Two different approaches have been taken: either by considering local constraints, comparing only "relative" subterms, or by using global constraints, which allow comparison of arbitrary subterms.

In this section, which is very strongly inspired by [Filiot et al., 2008], we will quickly go over some of these extensions.

2.2.1 The class AWEDC (1981)

The class *AWEDC* (short for *Tree Automata With Equality and Difference Constraints*) was first introduced by Max Dauchet and Jocelyne Mongy in Mongy's PhD thesis. Equality and difference constraints can be specified between any subterms. Transition rules are of the form:

$$f(q_1, q_2, q_3) \rightarrow_{1.3=2,\ 1.2\neq 1.3} q.$$

Then the rule only applies when, letting α be the position of the current term t, we have $t|_{\alpha.1.3} = t|_{\alpha.2}$ and $t|_{\alpha.1.2} \neq t|_{\alpha.1.3}$. This class is quite expressive and has good closure properties, unfortunately emptiness is undecidable.

2.2.2 Subclasses of AWEDC (≈ 1990)

About ten years later, driven by new motivation in fields such as term rewriting, new subclasses of AWEDC were introduced. One of them is very simple: Automata With Constraints Between Brothers restricts AWEDC to constraints between immediate siblings. So for instance this rule

$$f(q_1, q_2, q_3) \to_{1.3=2,\ 1.2\neq 1.3} q.$$

is not allowed anymore while this one

$$f(q_1, q_2, q_3) \to_{1=2,\ 1\neq 3} q.$$

would still be. Thanks to this restriction emptiness becomes decidable, although it remains *EXPTIME*-complete in the case of non-deterministic automata.

The second class, called "reduction automata", involves ordering the states of Q and applying a transition rule

$$f(q_1, \ldots, q_n) \to_{\varphi} q \quad \text{where } \varphi \text{ is an AWEDC constraint}$$

only if q is strictly smaller than each q_i. This time emptiness is decidable (efficiently) in the deterministic case, but not in the non-deterministic one.

2.2.3 TAGED (2006)

Until now, the extensions which we have mentioned only used local constraints, that is to say, constraints between related subterms. There exist other classes which use *global* constraints, which apply between arbitrary subterms. The most recent addition to this family is the class TAGED: Tree Automata with Global Equality and Disequality Constraints, which were introduced in Emmanuel Filiot's PhD thesis [Filiot, 2008] and in the article [Filiot et al., 2008].

Definition 2 (TAGED)**.** A TAGED is a tuple $\mathcal{A} = (\Sigma, Q, F, \Delta, =_{\mathcal{A}}, \neq_{\mathcal{A}})$, where

- (Σ, Q, F, Δ) is a tree automaton
- $=_{\mathcal{A}}$ is a reflexive symmetric binary relation on a subset of Q
- $\neq_{\mathcal{A}}$ is an irreflexive and symmetric binary relation on Q. Note that in our work, we have dealt with a slightly more general case, where $\neq_{\mathcal{A}}$ is not necessarily irreflexive.

A TAGED $\mathcal{A}$ is said to be *positive* if $\neq_{\mathcal{A}}$ is empty and *negative* if $=_{\mathcal{A}}$ is empty.

There is also a special subclass of TAGEDs which will interest us when we study the emptiness problem:

Definition 3 (Identity relation)**.** For any set S, we let $\mathfrak{id}_S$ be the identity relation on S, that is to say: $\mathfrak{id}_S = \{ (x,x) \mid x \in S \}$.

Definition 4 (Diagonal TAGED)**.** A TAGED $\mathcal{A} = (\Sigma, Q, F, \Delta, =_{\mathcal{A}}, \neq_{\mathcal{A}})$ is said to be *diagonal* if $(=_{\mathcal{A}}) \subseteq \mathfrak{id}_Q$.

The notion of run is subject to a new constraint: it must be compatible with the equality and disequality constraints:

Definition 5 (Compatibility with the global constraints)**.** A run ρ is compatible with the equality constraint $=_{\mathcal{A}}$ if

$$\forall \alpha, \beta \in \mathcal{P}os(t) : \rho(\alpha) =_{\mathcal{A}} \rho(\beta) \implies t|_\alpha = t|_\beta .$$

In the same way, ρ is compatible with the disequality constraint $\neq_{\mathcal{A}}$ if

$$\forall \alpha, \beta \in \mathcal{P}os(t) : \rho(\alpha) \neq_{\mathcal{A}} \rho(\beta) \implies t|_\alpha \neq t|_\beta .$$

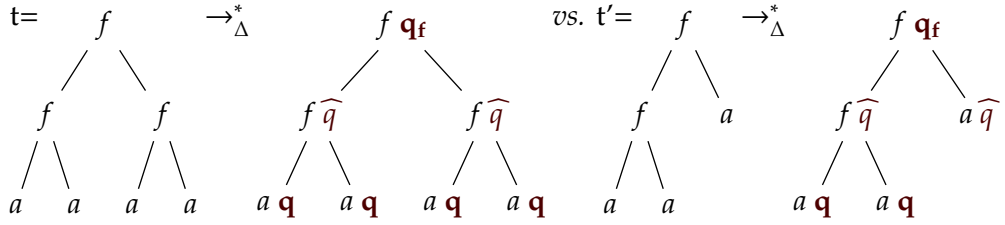
If $\neq_{\mathcal{A}}$ is not assumed to be irreflexive, this last definition must be extended into

$$\forall \alpha, \beta \in \mathcal{P}os(t) : \alpha \neq \beta \wedge \rho(\alpha) \neq_{\mathcal{A}} \rho(\beta) \implies t|_\alpha \neq t|_\beta .$$

Every other notion remains unchanged compared to vanilla tree automata. It is clear that TAGEDs are at least as expressive as the later, since they coincide exactly when $=_{\mathcal{A}}$ and $\neq_{\mathcal{A}}$ are both empty. They are in fact strictly more expressive, since they can recognise languages which vanilla tree automata cannot, such as the aforementioned $\{ f(t,t) \mid f \in \Sigma, t \in \mathcal{T}(\Sigma) \}$ and $\{ f(t,t') \mid f \in \Sigma, t, t' \in \mathcal{T}(\Sigma) : t \neq t' \}$. Let us convince ourselves that this is the case with the following, very classical example: the very simple positive TAGED $\mathcal{A}$ below recognises the language $\{ f(t,t) \mid f \in \Sigma, t \in \mathcal{T}(\Sigma) \}$.

$$\mathcal{A} \stackrel{\text{def}}{=} \left(\Sigma = \{ a/_0, f/_2 \},\ Q = \left\{ q, \widehat{q}, q_f \right\},\ F = \left\{ q_f \right\},\ \Delta,\ \widehat{q} =_{\mathcal{A}} \widehat{q},\ \widehat{q} \neq_{\mathcal{A}} q_f \right),$$
$$\text{where } \Delta \stackrel{\text{def}}{=} \left\{ f(\widehat{q}, \widehat{q}) \to q_f,\ f(q,q) \to q,\ f(q,q) \to \widehat{q},\ a \to q,\ a \to \widehat{q}, \right\}$$

To see how this works, consider the two terms below, and their runs for the underlying tree automaton. Both have accepting runs, since q_f if a final state, and are therefore in the language of the underlying tree automaton. The run over the first term has two instances of $\widehat{q}$ at positions 1 and 2, both over subterms structurally equal to $f(a,a)$. In other words we have $t|_1 = t|_2 = f(a,a)$. This term t is therefore compatible with the equality constraint $\widehat{q} =_{\mathcal{A}} \widehat{q}$ and it follows that the first term is accepted by the $\mathcal{A}$. On the other hand, in the case of the second term t' we also have two instances of $\widehat{q}$, but this time, one is over the subterm $t'|_1 = f(a,a)$ while the other is over $t'|_2 = a$. And because of this, we have $t'|_1 \neq t'|_2$ which violates the equality constraint $\widehat{q} =_{\mathcal{A}} \widehat{q}$, and t' is rejected by the TAGED $\mathcal{A}$.



Decidability and complexity of several decision problems are given in table 2[p14]. In the table, TAGED+ and TAGED- stand for positive and negative TAGEDs, respectively.

Decision Problem	TAGED complexity
Emptiness, TAGED+	*EXPTIME*-complete
Emptiness, TAGED-	*NEXPTIME*
Finiteness, TAGED+, $(=_{\mathcal{A}}) \subseteq \mathfrak{id}_Q$	$O(\|\|A\|\| \times \|Q\|^2)$
Finiteness, TAGED+	*EXPTIME*
Membership	*NP*-complete
Universality	Undecidable
Inclusion, TAGED+	Undecidable

Table 2 — Decision problems for TAGED

Theorem 6 (Closure properties of TAGEDs)**.** *The class of languages recognisable by* TAGEDs *is closed under union, and under intersection. It is not closed under complementation.*

2.3 The Emptiness decision problem

In this work, we focus on the generation of automata which are interesting with respect to the emptiness decision problem, which has been mentioned before and is here formally defined:

Definition 7 (Emptiness decision problem[(b)])**.**
Input: $\mathcal{A}$, an automaton (in this case a positive TAGED)
Output: $\mathcal{L}\mathrm{ng}(\mathcal{A}) = \varnothing$?

This problem arises in many circumstances, for instance in model checking, where the question of whether a "bad" state is reachable translates to emptiness of the tree automaton accepting the intersection of the language of bad states and that of reachable states. In this context, the additional expressivity granted by positive TAGEDs over vanilla tree automata could find applications to, for instance, the verification of cryptographic

[(b)] Note that we will sometimes take the shortcut of writing "the automaton is empty" to mean "the automaton tests positive *wrt.* to the emptiness problem", *ie.* "the language accepted by the automaton is empty".

protocol, where the equality constraints could guarantee that, say, the same key is used for encryption and decryption.

Decidability of the emptiness problem for TAGEDs in general remained an open problem for a while, but it recently turned out to be decidable, although the precise complexity of the problem is still unknown. See for instance [Barguñó et al., 2010] [(c)] and Camille Vacher's PhD. thesis.

More is known about the two main subclasses of TAGEDs: positive and negative TAGEDs. As shown in table 2[p14], the emptiness problem of in *NEXPTIME* for negative TAGEDs, and *EXPTIME*-complete for positive TAGEDs – and this is the best known lower bound for the complexity of emptiness of TAGEDs. [Filiot, 2008, Filiot et al., 2008, Barguñó et al., 2010].

3 Objectives and Strategy

As mentioned in the introduction, our present research goal is the development of reasonably efficient approaches for deciding the emptiness problem for positive TAGEDs. We expect our random generation scheme to be a means to that end, in that it should be the basis for a systematic and reliable experimental protocol enabling us to discriminate between approaches which are likely to be efficient in practice and approaches which are not. In order to function well in that role, the random generation scheme must satisfy the following two informal constraints:

Difficulty:
The generated automata should not be obvious instances of the problem, *ie.* simple or naive algorithms should only have a very slim chance of deciding quickly.

Realism:
The generated instances (*ie.* positive TAGEDs) should be reasonably similar (or at the very least, should not be unreasonably dissimilar) to instances such as those which one would expect to deal with in the "real world".

The necessity of those two constraints is quite clear. If the first one is not satisfied, then *all* algorithms one could come up with can be expected to perform well, and the results will fail to discriminate between the efficient ones and the rest. On the other hand, if the second constraint is not satisfied, the results, even if they do serve to discriminate between our algorithms, will be of questionable relevance to the performance of practical tools, dealing with real cases.

In this short section, we will elaborate on those informal constraints, which act as our main guidelines for the development of the random generation scheme. Instead of

[(c)] Available online: `http://www.lsi.upc.es/~ggodoy/papers/globalconstraints.pdf`.

tackling those two notions "head on" – which would be problematic as they are large and rather vague, we will take the simpler approach of listing characteristics which, in our experience, clearly violate either of those guidelines. Each of the cases described below corresponds to shortcomings which we identified in our previous attempts at developing random generation schemes, and which we wish to avoid this time around.

For instance, a random generation scheme clearly fails to generate difficult instances in the following cases:

- One can decide emptiness (surely or with high probability) without even needing to look at the instance. This is the case if **the generation scheme is deeply flawed**, and only generates automata whose accepted language is empty (*resp.* non-empty). Not only would there be little point in running a decision algorithm in such a case, but that would leave important aspects of the algorithms untested. Imagine, for instance, that we have developed an approach that is quite efficient on, say, empty instances, but takes forever in the other case. Then the experimental protocol would only serve to reveal one aspect, and utterly ignore the other one, which is obviously quite an undesirable characteristic. Ideally, the generator should yield roughly even proportions of empty and non-empty instances.

- Too many instances fall into **known easy cases**, which can be trivially – *eg.* linearly – detected as such.

  - If the underlying tree automaton is empty, then so is the `TAGED`. Emptiness of a vanilla tree automaton can be tested in linear time [Comon et al., 2007].
  - The emptiness of diagonal positive `TAGED`s is known to be equivalent to that of their underlying tree automaton [Filiot et al., 2008]. Detecting diagonality and testing emptiness of the underlying tree automaton can be done in linear time.

  Note that this is not to say that it would be inappropriate to generate *any* such cases. They do arise in practice, and it is of course essential that decision procedures should take advantage of them. However, having them arise too frequently would make the experimental protocol far too forgiving for our purposes.

- The instances can frequently be solved trivially by obvious, **brute-force algorithms**. A brute-force algorithm would typically start by generating leaves, that is to say, generating terms and the corresponding runs using leaf rules (*ie.* rules of the form $a \to q$), and then combine them using higher-arity rules, building terms of greater height with each iteration. That approach decides emptiness and terminates thanks to the pumping lemma for positive `TAGED`s [Filiot et al., 2008]. Suppose that we generate a sizable proportion of `TAGED`s which accept terms of very small height; then such an algorithm would terminate very quickly. The

worst case is when non-empty automata are very likely to have final leaf rules, *ie.* rules of the form $a \to q_f$, where $q_f \in F$. In that case a brute-force algorithm would not even enter the main loop: the initial phase – which is linear – would be enough in almost all cases. This remark leads to the important notion that, in order to ensure that our generated (non-empty) instances present a modicum of difficulty, we have to pay attention to the *minimum accepted height* of our instances, that is to say, the height of the smallest term recognised by our instances.

- It is often the case that **all final states are in dead branches** of the generated instances, in which case the automata are clearly empty. What we call *dead branches* here are states and rules which can be easily (*ie.* in polynomial time and space) recognised as useless to the automaton. This is for instance the case of unreachable states, but it is often possible to find states which, although they are reachable, contribute nothing to the accepted language of the automaton. We shall see several such cases in some detail in the next section.

Similarly, the generated instances are not very realistic in the following cases:

- Arguably, if they are either of **very great or very small size**. The driving idea is to generate cases which are both difficult and of reasonable size; achieving difficulty through the generation of gargantuan cases would not really be conducive to our overall objectives. That is not to say that tree automata, and by extension `TAGED`s, cannot get quite big in practice, for instance in model-checking; they certainly can. But for the sake of our experimental protocol we wish to isolate cases of the emptiness problem which are inherently hard to solve instead of merely being oversized versions of otherwise easy cases.

- If they are **too dense**. In our experience, tree automata used in model checking and other such applications are rather sparse.

- If they fall too often in one of the two pathological categories of tree automata which we will call informally ***"soup blenders"* and *"waffle irons"***.

  - We call *soup blenders* automata whose accepted languages, often finite, are almost entirely composed of leaves, with very few, if any, terms of higher heights. That informal nickname was chosen to account for the apparent lack of any identifiable "structure" in the accepted languages. We will see that randomly generated tree automata tend to fall in that category quite easily, unless special precautions are taken to prevent it.
  - At the other end of the spectrum, *waffle irons* are tree automata whose recognised languages are finite and such that every accepted term is isomorphic to one of a few trees – regardless of the height of those terms.

In order to attain some degree of realism, our generated automata must strike a balance between the two, and have good chances of accepting infinite languages.

- If they are **"Frankenstein" TAGEDs**, whose rules and constraints do not quite seem to fit together. Automata from real-world applications are not thrown together haphazardly: they are carefully written to best formalise a specific language and each state, rule, and in the case of TAGEDs, each constraint must have a part to play *ie.* cannot simply be removed without affecting the accepted language. To reflect that, generated instances should avoid having any of the following:
  - unreachable states
  - states that *cannot* appear in any accepted term
  - rules that immediately violate the constraints
  - everything which we will call "dead branches" in general.

With this in mind, here is an outline of the strategy of generation:

1. Generate a "raw" random positive TAGED $\mathcal{A}$. It is best if $\mathcal{A}$ avoids as many of the above pitfalls from the get-go, but this step should be kept computationally inexpensive.
2. Test whether $\mathcal{A}$ is a suitable instance, *ie.* if it is within the parameters of the remaining constraints. If it is not, then discard it and go back to the first step.
3. Remove all dead branches from $\mathcal{A}$, and ship it.

There are two main themes which appear, each of which will be the object of a section of this report:

Detection of easy cases, removal of dead branches
This serves to ensure both difficulty and realism, and is in fact done in one single, polynomial operation, called the *cleanup*, which is detailed in the next section 4[p18].

Generation of "raw" positive TAGEDs
The directing idea here will be to guide the random generation with constraints on both the structure of the accepted language (*eg.* minimum accepted height) and that of the automaton itself (*eg.* avoiding unreachable states). See section 5[p27].

4 Getting rid of some obvious cases and dead branches

In this chapter we will show that the presence of a global equality constraint may render a number of transition rules and states visibly inoperative. Those rules and states will

be called *spurious* if it is clear that their use would fatally be in contradiction with the equality constraint, and therefore that they can be removed without altering the language recognised by the TAGED.

This chapter formalises and justifies this notion of *spurious constructions* and presents algorithms to sanitise TAGEDs, that is to say, to remove all spurious constructions from the automaton. This operation can be seen as an extension of the reduction algorithm for vanilla tree automata to positive TAGEDs. Though such an operation can well be used for instance to lighten the load of expensive decision algorithms, our aim here is purely to improve the realism of the generated instance, by pruning branches which are obviously useless and out of place.

4.1 Detecting "spurious" states and rules

Let us begin with an observation which applies to vanilla tree automata as well as TAGEDs; the classical reduction algorithm given in [Comon et al., 2007] removes from the automaton all those states which are not reachable and thus contribute nothing. However this does not mean that every reachable state *does* contribute something; if a state has no possible use whatsoever in building an accepted term, that is to say, if that state is neither final nor usable in a run which leads to a final state, then it is useless and can safely be removed, even if it is reachable. Of course, by "safely" we mean that the language accepted by the automaton is not affected in any way by this operation. This is the object of theorem 14[p21]; first let us introduce two simple definitions for syntactic convenience:

Definition 8 (Associated rules)**.** Let $\mathcal{A}$ be a TAGED and $q \in Q$. The the *associated rules* of q are defined as $\mathfrak{Rul}(q) \stackrel{\text{def}}{=} \{ r \in \Delta \mid r = f(\dots) \to q \}$.

Definition 9 (Antecedents)**.** Let $\mathcal{A}$ be a TAGED and $r = f(q_1, \dots, q_n) \to q \in \Delta$. We call *antecedents of* r, and denote $\mathfrak{Ant}(r)$, the set $\{ q_1, \dots, q_n \}$.

With these definitions, let us express the idea that for each state, say, q, there are a limited number of rules which could have produced it – the rules in $\mathfrak{Rul}(q)$ – and therefore in a run, q's children can only be chosen among those states which are antecedents to one one those rules. This is what we can *potential requirements of* q: the set of states which *can*, potentially, appear as direct children of q in a well-formed run.

Definition 10 (Potential requirements)**.** Let $\mathcal{A}$ be a TAGED, and let $q \in Q$. The *potential requirements* of state q are defined as

$$\mathfrak{pReq}(q) \stackrel{\text{def}}{=} \bigcup_{r \in \mathfrak{Rul}(q)} \mathfrak{Ant}(r).$$

Now, we can generalise this notion: if only a few states, say, p_k, can be q's children, then only the states which can be p_k's children for some k – *ie.* are a potential requirement of p_k

– can be q's grand-children... And thus we define the set of all states which can appear under q, either as direct children, grand-children, etc. We call these states "friends of q".

Definition 11 (Friend states)**.** Let $\mathcal{A}$ be a TAGED, and $q \in Q$. We define $\mathfrak{Frnd}(q)$ as the smallest subset of Q satisfying the two following properties:

1. $\mathfrak{pReq}(q) \subseteq \mathfrak{Frnd}(q)$
2. if $p \in \mathfrak{Frnd}(q)$ then $\mathfrak{pReq}(p) \subseteq \mathfrak{Frnd}(q)$

The next lemma formalises and justifies what we have been saying informally: it states and proves that if a certain state q appears in a well-formed run, then we know that all the states which appear under q are its friends.

Lemma 12 ("Rely on your Friends" principle)**.** *Let $\mathcal{A}$ be a TAGED, $t \in \mathcal{T}(\Sigma)$ a term, and ρ a run of the underlying tree automaton $\mathfrak{ta}(\mathcal{A})$ on t. Then the following holds : $\forall \alpha, \beta \in \mathcal{P}os(t)$: $\beta \lhd \alpha \implies \rho(\beta) \in \mathfrak{Frnd}(\rho(\alpha))$.*

Proof. We will prove the equivalent statement $\forall \alpha \in \mathcal{P}os(t), \forall \beta \in \mathcal{P}os(t) : \exists n \geqslant 1 : \beta \lhd_n \alpha,\ \rho(\beta) \in \mathfrak{Frnd}(\rho(\alpha))$ by induction on n. Let $\alpha \in \mathcal{P}os(t)$, fixed but arbitrary.

1. (base case) let $\beta \lhd_1 \alpha$; then β is a direct child of α. It follows immediately from definition 10[p19] that we have $p \in \mathfrak{pReq}(q) \iff \exists f(\dots p \dots) \to q \in \Delta$, and thus if we were to assume that $\rho(\beta) \notin \mathfrak{pReq}(\rho(\alpha))$, it would follow that there is no rule $f(\dots \rho(\beta) \dots) \to \rho(\alpha) \in \Delta$, which would imply that ρ is not compatible with the transition rules and is therefore not a run. Since ρ is in fact a run this is absurd, and $\rho(\beta) \in \mathfrak{pReq}(\rho(\alpha)) \subseteq \mathfrak{Frnd}(\rho(\alpha))$.
2. (inductive case) let us assume that, for some n, $\forall \beta \lhd_n \alpha : \rho(\beta) \in \mathfrak{Frnd}(\rho(\alpha))$. Let $\gamma \in \mathcal{P}os(t)$ such that $\gamma \lhd_{n+1} \alpha$; then, $\mathcal{P}os(t)$ being prefix-closed, there must exists some $\beta \in \mathcal{P}os(t)$ such that $\gamma \lhd_1 \beta \lhd_n \alpha$. By the same reasoning as in the base case we have $\rho(\gamma) \in \mathfrak{pReq}(\rho(\beta))$, it follows by the definition of the friends states and our induction hypothesis that $\rho(\gamma) \in \mathfrak{Frnd}(\rho(\alpha))$.

Thus we have proved the result by induction. □

Before moving on to the announced theorem, we need to formalise what we meant by "removing a state from an automaton", which we call *restriction*. Since it is an operation which we will use quite frequently it deserves its own notation. Note that this is a straightforward adaptation of the notion used implicitly in [Comon et al., 2007], for instance when describing the reduction algorithm (*cf.* figure 1[p21] for the algorithm). We also introduce the *projection*, which consists simply in changing the set of final states.

Definition 13 (Restriction by states, projection)**.** Let $\mathcal{A} = (\Sigma, Q, F, \Delta, =_{\mathcal{A}}, \neq_{\mathcal{A}})$ be a TAGED, and let $S \subseteq Q$ be a set of states. We call *restriction of $\mathcal{A}$ to S* and denote $\mathfrak{Rst}(\mathcal{A}, S)$ the TAGED $(\Sigma, S, F \cap S, \Delta', =_{\mathcal{A}} \cap S^2, \neq_{\mathcal{A}} \cap S^2)$ where

$$\Delta' \stackrel{\text{def}}{=} \{ f(q_1, \dots, q_n) \to q \in \Delta \mid \{q, q_1, \dots, q_n\} \subseteq S \}.$$

Data: A TAGED $\mathcal{A}$
Result: A TAGED $\mathcal{A}'$ such that $\mathcal{L}\mathrm{ng}(\mathcal{A}) = \mathcal{L}\mathrm{ng}(A')$
begin
  Reach $\leftarrow \emptyset$;
  repeat
    add q to Reach where $r \in \mathfrak{Rul}(q), \mathfrak{Ant}(r) \subseteq$ Reach;
  until *no state can be added to* Reach ;
  return $\mathfrak{Rst}(\mathcal{A}, \text{Reach})$;
end

Figure 1 — Reduction algorithm, from [Comon et al., 2007, page 25]

We also call *projection of* $\mathcal{A}$ *on* S the TAGED $\mathfrak{Prj}(\mathcal{A}, S) \stackrel{\text{def}}{=} (\Sigma, Q, S, \Delta, =_{\mathcal{A}}, \neq_{\mathcal{A}})$.

With these new tools in hand, we can at last justify what we said at the beginning of this section: only those states which can possibly be used to build a final state – *ie.* which are friends of a final state – are of any real use for the automaton. The others can be removed without altering its recognised language.

Theorem 14 (Removal of useless states). *Let* $\mathcal{A} = (\Sigma, Q, F, \Delta)$ *be a tree automaton. Then*

$$\mathcal{L}ng(\mathcal{A}) = \mathcal{L}ng(\mathcal{A}') \quad \textit{with} \quad \mathcal{A}' \stackrel{def}{=} \mathfrak{Rst}\left(\mathcal{A}, F \cup \bigcup_{q_f \in F} \mathfrak{Frnd}(q_f)\right).$$

Furthermore, the accepting runs are the same for $\mathcal{A}$ *and* $\mathcal{A}'$.

Proof. Let us show that some run ρ is an accepting run of $\mathcal{A}$ is and only if it is an accepting run of $\mathcal{A}'$. Since $\mathcal{A}'$ is a restriction of $\mathcal{A}$, it is clear that *any* run of $\mathcal{A}'$ is also a run of $\mathcal{A}$. It remains to show that if ρ is an *accepting* run of $\mathcal{A}$, it is also an accepting run of $\mathcal{A}'$. Suppose that this is not the case, that is to say, there exists a term $t \in \mathcal{T}(\Sigma)$ such that $\mathcal{A}$ accepts t through the run ρ, but ρ is not an accepting run of $\mathcal{A}'$. This could happen if ρ was a run for $\mathcal{A}'$, but not an accepting one; that is to say $\rho(\varepsilon) \in F_{\mathcal{A}}$ but $\rho(\varepsilon) \notin F_{\mathcal{A}'}$. However by definition of the restriction we have $F_{\mathcal{A}} = F_{\mathcal{A}'} = F$. Thus if ρ was a run for $\mathcal{A}'$, it would have to be accepting. Therefore ρ is not a run for $\mathcal{A}'$. It follows that ρ makes use of one of the rules which were removed, and by definition each removed rule makes use of a state which is neither final nor in $\bigcup_{q_f \in F} \mathfrak{Frnd}(q_f)$. So we can conclude that there exists $\alpha \in \mathcal{P}os(t), p \notin F \cup \bigcup_{q_f \in F} \mathfrak{Frnd}(q_f)$ such that $\rho(\alpha) = p$. On the other hand, we know by definition of an accepting run that $\rho(\varepsilon) \in F$, and either $\alpha = \varepsilon$, which is contradictory since $\rho(\alpha) = p \notin F$, or $\alpha \lhd \varepsilon$. But in that case lemma 12[p20] applies and $p = \rho(\alpha) \in \mathfrak{Frnd}(\rho(\varepsilon))$. that is also in contradiction with $p \notin \bigcup_{q_f \in F} \mathfrak{Frnd}(q_f)$. In all cases, we are faced with contradictions, and so our assumption is disproved, and ρ *is* an accepting run for $\mathcal{A}'$. □

The result applies only to vanilla tree automata so far. Fortunately the added complication of global equality (or even disequality) constraints does not invalidate the result.

Corollary 15 (Removal of useless states)**.** *The same result as theorem 14*$_{[p21]}$ *holds for* `TAGED`*s.*

Proof. By theorem 14 the accepting runs of $\mathtt{ta}(\mathcal{A})$ are those of $\mathtt{ta}(\mathcal{A}')$ and *vice versa*. Since $\mathcal{A}'$ is a restriction of $\mathcal{A}$, its constraints are weaker and therefore the accepting runs of the former are a superset of those of the latter. There remains to show that every accepting run of $\mathcal{A}'$ is also accepting for $\mathcal{A}$. Let ρ be a successful run of $\mathcal{A}'$ on a term $t \in \mathcal{T}(\Sigma)$. So for all $\alpha \in \mathcal{P}os(t)$, we have $\rho(\alpha) \in F \cup \bigcup_{q_f \in F} \mathfrak{Frnd}(q_f)$. Let us suppose that ρ is *not* a successful run for $\mathcal{A}$. Then since it is an accepting run of $\mathtt{ta}(\mathcal{A})$, it must be incompatible with a global constraint. So there exist $p, q \in Q$ such that, say, $p =_{\mathcal{A}} q$, and two positions $\alpha, \beta \in \mathcal{P}os(t)$ such that $\rho(\alpha) = p$ and $\rho(\beta) = q$ and $t|_\alpha \neq t|_\beta$. But ρ is compatible with the constraints of $\mathcal{A}'$, therefore at least p or q must be specific to $\mathcal{A}$, that is, be in $Q \setminus F \cup \bigcup_{q_f \in F} \mathfrak{Frnd}(q_f)$. This is a contradiction. Thus ρ is also an accepting run for $\mathcal{A}$. □

Now let us examine `TAGED`s, or more specifically, positive `TAGED`s in more detail, and study some immediate consequences of the introduction of constraints to the influence of some rules and states. It seldom hurts to state the obvious, so let us do so in this next lemma:

Lemma 16. *Let $\mathcal{A}$ be a* `TAGED`*. If $\mathcal{L}ng(\mathtt{ta}(\mathcal{A})) = \varnothing$ then $\mathcal{L}ng(\mathcal{A}) = \varnothing$.*

Proof. We know that, trivially, $\mathcal{L}\mathrm{ng}(\mathcal{A}) \subseteq \mathcal{L}\mathrm{ng}(\mathtt{ta}(\mathcal{A}))$, and this result follows. □

Testing emptiness of a tree automaton is linear, so this is a very inexpensive test – which is quite fortunate as we will be using it fairly often. In the particular case when a positive `TAGED` is diagonal – that is to say, $=_{\mathcal{A}} \subseteq \{(q,q) \mid q \in Q\}$: all its equality constraints are of the form $q =_{\mathcal{A}} q$ – then this linear test is enough to decide emptiness.

Theorem 17 (Diagonal testing)**.** *Let $\mathcal{A}$ be a diagonal positive* `TAGED`*. Then $\mathcal{L}ng(\mathcal{A}) = \varnothing \iff \mathcal{L}ng(\mathtt{ta}(\mathcal{A})) = \varnothing$.*

Proof. See beginning of proof of [Filiot et al., 2008, Theorem 1]. □

Now that those preliminary observations are over and done with, let us move on to what we announced earlier on: the observation of the contradictions which the introduction of global equality constraints can create in a tree automaton. We will see that some rules become absurd and some states unusable when certain conditions are met. We call those rules and states "spurious". We will define those conditions and show that, just as was the case for useless states, spurious elements can be removed from a `TAGED` without altering its accepted language. We start by the most obvious observation:

Definition 18 (Spurious rule)**.** Let $\mathcal{A}$ be a TAGED. A rule $f(q_1, \ldots, q_n) \to q \in \Delta$ is *spurious* if there exists $k \in [\![1, n]\!]$ such that $q_k \equiv_{\mathcal{A}} q$.

It is clear that no spurious rule can actually be used in any run. If that was the case then there would exist a term structurally equal to one of its strict subterms, which is absurd. So it follows that spurious rules have no influence whatsoever on the language recognised by a TAGED. This outlines the proof of the next lemma.

Lemma 19 (Removal of spurious rules)**.** *Let $\mathcal{A}$ be a TAGED, and let $S \subseteq \Delta$ be the set of all the spurious rules of Δ. Then, if we let $\mathcal{A}' \stackrel{def}{=} (\Sigma, Q, F, \Delta \setminus S, \equiv_{\mathcal{A}}, \not\equiv_{\mathcal{A}})$, we have $\mathcal{L}ng(\mathcal{A}) = \mathcal{L}ng(\mathcal{A}')$.*

Proof. We have trivially $\mathcal{L}ng(\mathcal{A}) \supseteq \mathcal{L}ng(\mathcal{A}')$. Let $t \in \mathcal{L}ng(\mathcal{A})$ and let ρ be the run by which t has been accepted. Being a run, ρ is compatible with the transition rules, that is to say for any position $\alpha \in \mathcal{P}os(t)$, there exists a transition rule

$$r = t(\alpha)\big(\rho(\alpha.1), \ldots, \rho(\alpha.arity(t(\alpha)))\big) \to \rho(\alpha) \in \Delta.$$

Suppose that r is spurious. Then there is a k such that $\rho(\alpha.k) \equiv_{\mathcal{A}} \rho(\alpha)$, and it follows that $t|_\alpha = t|_{\alpha.k}$. Thus t is structurally equal to its own child, which is absurd. Therefore r is not spurious: $r \in \Delta \setminus S$, and it follows that ρ is also a run for $\mathcal{A}'$. Finally we have $\mathcal{L}ng(\mathcal{A}) \subseteq \mathcal{L}ng(\mathcal{A}')$, which concludes the proof. □

We will now extend this notion of "spurious construction" to less direct cases, where instead of having immediate spurious rules, we have two or more rules leading to the same kind of contradictions. The watchful reader will notice an uncanny similarity between the *potential* requirements introduced at the beginning of the present chapter and the *sure* requirements which we are about to define. While the former generously encompassed all the states which *could* possibly be direct children of some state q, the latter is limited to the very closed circle of those states which *must* appear as direct children of q, for the simple reason that every single rule which builds q uses them as antecedents.

Definition 20 (Sure requirements)**.** Let $\mathcal{A}$ be a TAGED, and let $q \in Q$. The *sure requirements* of state q are defined as

$$\mathfrak{sReq}(q) \stackrel{\text{def}}{=} \bigcap_{r \in \mathfrak{Rul}(q)} \mathfrak{Int}(r).$$

We extend this notion in the same way we did before when we went from potential requirements to friends: if q is sure to need p as its direct child, and p is sure to need p' as its own direct child, then q is sure to need p' to be its grand-child. If p' does not appear in a run, neither can q. We call "needs of q" the set of states which, according to this principle, must appear in a run if q itself appears.

Definition 21 (Needs [d])**.** Let $\mathcal{A}$ be a TAGED, and $q \in Q$. We define $\mathfrak{Need}(q)$ as the smallest subset of Q satisfying the two following properties:

[d] Better definitions are possible for $\mathfrak{Need}(q)$, see paragraph on future work, section 7[p40].

1. $\mathfrak{sReq}(q) \subseteq \mathfrak{Need}(q)$
2. if $p \in \mathfrak{Need}(q)$ then $\mathfrak{sReq}(p) \subseteq \mathfrak{Need}(q)$

We now formalise and prove what we said informally: if a state appears in a run, then it is necessary that its needs should appear under it in this same run.

Lemma 22 (Needs). *Let $\mathcal{A}$ be a* TAGED*, and let $t \in \mathcal{T}(\Sigma)$, $\beta \in \mathcal{P}os(t)$ and $q \in Q$. Let ρ be a run of $\mathcal{A}$ on t, compatible with the global constraints, such that $\rho(\beta) = q$. Then for any $p \in \mathfrak{Need}(q)$, there exists a position $\alpha_p \lhd \beta$ such that $\rho(\alpha_p) = p$.*

Proof. We prove this result by induction on $\mathfrak{Need}(q)$.

1. (base case) Suppose $p \in \mathfrak{sReq}(q)$. Since ρ is a run, it is compatible with the transition rules of Δ. We have $\rho(\beta) = q$, therefore, letting n be the arity of $t(\beta)$, there exists a rule $f(q_1, \dots, q_n) \to q \in \mathfrak{Rul}(q)$ such that for all $k \in [\![1, n]\!]$, $q_k = \rho(\beta.k)$. By definition of $\mathfrak{sReq}(q)$, there exists $i \in [\![1, n]\!]$ such that $p = q_i = \rho(\beta.i)$. We have $\alpha_p = \beta.i \lhd \beta$.
2. (inductive case) Suppose that there exists $p' \in \mathfrak{Need}(q)$ such that $p \in \mathfrak{sReq}(p')$. By induction hypothesis there exists $\alpha_{p'} \lhd \beta$ such that $\rho(\alpha_{p'}) = p'$. We use the same arguments as in the base case. Let m be the arity of $t(\alpha_{p'})$. Then there exists a rule $f(q_1, \dots, q_m) \to p' \in \mathfrak{Rul}(p')$ such that for all $k \in [\![1, m]\!]$, $\rho(\alpha_{p'}.k) = q_k$. By definition of $\mathfrak{sReq}(p')$, there exists $i \in [\![1, m]\!]$ such that $p = q_i = \rho(\alpha_{p'}.i)$. We have $\alpha_p = \alpha_{p'}.i \lhd \alpha_{p'} \lhd \beta$.

Thus the proof is concluded. □

In other words, in order to "build" the state q, one must first be able to build any state $p \in \mathfrak{Need}(q)$ strictly under it. Suppose that we are in the following scenario: we have some states $q_0, \dots, q_n$ such that $q_0 \equiv_{\mathcal{A}} q_n$, some symbols $f_0, \dots, f_n$ (not necessarily all distinct) and the rules

$$\begin{array}{rl}
f_0(\dots, q_0, \dots) \to q_1 & \in \Delta \\
\dots & \\
f_k(\dots, q_k, \dots) \to q_{k+1} & \in \Delta \\
\dots & \\
f_{n-1}(\dots, q_{n-1}, \dots) \to q_n & \in \Delta
\end{array}$$

If, for any $k \in [\![1, n]\!]$, we have no rule $r \in \mathfrak{Rul}(q_k)$ such that $q_{k-1} \notin \mathfrak{Int}(r)$, that is to say, if there is no way to build q_k without first building q_{k-1}, then it is impossible to build a term which evaluates to q_n. Indeed, such a term would necessarily have one of its strict children evaluate to q_0. But this is not compatible with $q_0 \equiv_{\mathcal{A}} q_n$. Such a state q_n will be called "spurious". The following definitions and lemma characterise spurious states and formalise the intuitive notion that spurious states can be removed from a TAGED without altering its accepted language.

Definition 23 (Spurious states)**.** Let $\mathcal{A}$ be a TAGED. A state $q \in Q$ is said to be a *spurious state* if there exists $p \in \mathfrak{Need}(q)$ such that $p =_{\mathcal{A}} q$.

That definition is rather intuitive: if you *need* p to be a strict child of q, and at the same time you need p and q to be structurally equal, then you are in trouble... As announced, this generalises the notion of spurious rules to cases where the child-father structural equality is buried a little more deeply. Clearly, one can safely get rid of such states:

Lemma 24 (Removal of spurious states)**.** *Let $\mathcal{A}$ be a* TAGED*, $S \subseteq Q$ the set of all its spurious states, and $\mathcal{A}' = \mathfrak{Rst}(\mathcal{A}, Q \setminus S)$. Then $\mathcal{L}ng(\mathcal{A}) = \mathcal{L}ng(\mathcal{A}')$.*

Proof. It is clear that $\mathcal{L}\mathrm{ng}(\mathcal{A}) \supseteq \mathcal{L}\mathrm{ng}(\mathcal{A}')$. Let $t \in \mathcal{L}\mathrm{ng}(\mathcal{A})$ and let ρ be the run by which t has been accepted. Suppose that there is a position $\beta \in \mathcal{P}os(t)$ such that $q = \rho(\beta) \in S$. Then by definition of a spurious state there exists a state $p \in \mathfrak{Need}(q)$ such that $p =_{\mathcal{A}} q$, and by lemma 22[p24], there exists a position $\alpha_p \lhd \beta$ such that $\rho(\alpha_p) = p$. By the equality constraint we have $t|_\beta = t|_{\alpha_p}$, but this is impossible because no term may be structurally equal to one of its strict children. Thus for all positions $\beta \in \mathcal{P}os(t)$, $\rho(\beta) \in Q \setminus S$, and ρ is also a run for $\mathcal{A}'$. Finally we have $\mathcal{L}\mathrm{ng}(\mathcal{A}) \subseteq \mathcal{L}\mathrm{ng}(\mathcal{A}')$, which concludes the proof. □

Until now we have only focused on spurious constructions based on the obvious impossibility of building a term equal to one of its strict subterms. We will now see another impossibility, based on the symbols of Σ.

Definition 25 (Support of a state)**.** Let $\mathcal{A}$ be a TAGED, and let $q \in Q$ be a state. We call *support of* q and denote $\mathfrak{Sup}(q)$ the set of all symbols of Σ in which a term which evaluates to q may be rooted. $\mathfrak{Sup}(q) \stackrel{\text{def}}{=} \{ f \in \Sigma \mid \exists f(\dots) \to q \in \Delta \}$.

Short of actually testing a full structural equality, it can be useful, be very inexpensive, to at least see whether the roots of two trees evaluating to to different states *can* possibly have the same symbol. Say that you have the constraint $p =_{\mathcal{A}} q$, but all the rules of $\mathfrak{Rul}(p)$ are of the form $f(\dots) \to p$, while all the rules of $\mathfrak{Rul}(q)$ are of the form $g(\dots) \to q$. Clearly, while those states could very well appear in runs, they cannot in any way appear *together*. Suppose now that there is a state which *requires* p and q to appear together in the run; then clearly the use of this state yields a contradiction, and again, it can be safely removed. This is what we call a Σ-spurious state.

Definition 26 (Σ-spurious state)**.** Let $\mathcal{A}$ be a TAGED. A state $q \in Q$ is said to be a *Σ-spurious state* if there exists $p, p' \in \mathfrak{Need}(q)$ such that $p =_{\mathcal{A}} p'$ and $\mathfrak{Sup}(p) \cap \mathfrak{Sup}(p') = \varnothing$.

Lemma 27 (Removal of Σ-spurious states[(e)])**.** *Let $\mathcal{A}$ be a* TAGED*, $S \subseteq Q$ the set of all its Σ-spurious states, and $\mathcal{A}' = \mathfrak{Rst}(\mathcal{A}, Q \setminus S)$. Then $\mathcal{L}ng(\mathcal{A}) = \mathcal{L}ng(\mathcal{A}')$.*

[(e)] Note that it would have been quite wrong to define a Σ-spurious state simply as a state such that $\exists p' \in Q : p =_{\mathcal{A}} p'$ and $\mathfrak{Sup}(p) \cap \mathfrak{Sup}(p') = \varnothing$. Such a state *can* be used in an accepting run, provided that its "opposite" p' does not appear in the same run.

Proof. It is clear that $\mathcal{L}\mathrm{ng}(\mathcal{A}) \supseteq \mathcal{L}\mathrm{ng}(\mathcal{A}')$. Let $t \in \mathcal{L}\mathrm{ng}(\mathcal{A})$ and let ρ be the run by which t has been accepted. Suppose that there is a position $\beta \in \mathcal{P}os(t)$ such that $q = \rho(\beta) \in S$. Then by definition of a Σ-spurious state there exist two states $p, p' \in \mathfrak{Need}(q)$ such that $p =_{\mathcal{A}} p'$, and by lemma 22[p24], there exist two distinct positions $\alpha, \alpha' \lhd \beta$ such that $\rho(\alpha) = p$ and $\rho(\alpha') = p'$. Since ρ is a run, it is compatible with the transition rules, and therefore $t(\alpha) \in \mathfrak{Sup}(p)$ and $t(\alpha') \in \mathfrak{Sup}(p')$. By the equality constraint we have $t|_\alpha = t|_{\alpha'}$, and thus $t(\alpha) = t(\alpha')$ and it follows that $t(\alpha) \in \mathfrak{Sup}(p) \cap \mathfrak{Sup}(p') = \varnothing$. This is absurd. Thus for all positions $\beta \in \mathcal{P}os(t)$, $\rho(\beta) \in Q \setminus S$, and ρ is also a run for $\mathcal{A}'$. Finally we have $\mathcal{L}\mathrm{ng}(\mathcal{A}) \subseteq \mathcal{L}\mathrm{ng}(\mathcal{A}')$, which concludes the proof. □

The act of removing all the spurious constructions from the `TAGED` is called *sanitising*. It is legitimated by the following theorem, which summarises the results of this section.

Theorem 28 (Sanitising)**.** *Let $\mathcal{A}$ be a* `TAGED`*, and let $Q_s \subseteq Q$ the set of all its spurious states, $Q_\Sigma \subseteq Q$ the set of all its Σ-spurious states, and $\Delta_s \subseteq \Delta$ the set of all its spurious rules. Then if we let $\mathcal{A}' = \mathfrak{Rst}\Big((\Sigma, Q, F, \Delta \setminus \Delta_s, =_{\mathcal{A}}, \neq_{\mathcal{A}}), F \cup \bigcup_{q_f \in F} \mathfrak{Frnd}(q_f) \setminus (Q_s \cup Q_\Sigma)\Big)$ we have $\mathcal{L}ng(\mathcal{A}) = \mathcal{L}ng(\mathcal{A}')$.*

Proof. Immediate consequence of corollary 15[p22] and lemmas 19[p23], 24[p25] and 27[p25]. □

Note that the resulting automaton is not necessarily the smallest one can obtain using this method: in some edge cases, removal of useless states might render some other states spurious and *vice versa*. So, in practice, we shall use this theorem repeatedly until a fixed point is reached. This operation will be referred to as "cleanup".

4.2 Example and conclusion

In this chapter, we have introduced the *cleanup* operation, which improves on the standard reduction algorithm for vanilla tree automata (*cf.* [Comon et al., 2007]), and takes advantage of the global equality constraints of a `TAGED` to detect and remove even more rules and states. The cleanup operation itself has a low, polynomial complexity, and is primarily intended to be used as a preliminary to a more expensive algorithm, such as emptiness, or as a tool to improve realism of randomly generated instances. Nevertheless, it should be noted that in some cases, it can be quite enough to positively decide emptiness, in which case the generated instance is discarded: see for instance the two examples below [(f)].

```
TAGED 'example 1' [64] = {
  states = #7{q0, q1, q2, q3, q4, q5, q6}
  final = #1{q6}
  rules = #16{
    a2()->q0, a2()->q2, a2()->q4, a3()->q3, a5()->q0, a5()->q2,
    a5()->q4, f1(q5)->q5, f3(q1)->q5, g1(q1, q5)->q5, g3(q0, q0)->q5,
```

[(f)] Kindly provided by random generation. *cf.* 5[p27].

```
    g3(q1, q5)->q5, g5(q1, q1)->q5, h2(q2, q3, q4)->q1,
    h3(q0, q0, q1)->q6, h3(q2, q3, q4)->q1
  }
  ==rel = #3{(q0,q0), (q3,q4), (q4,q3)}
}
```

This TAGED ('example 1') is in fact empty, and a cleanup operation suffices to show that it is. Its sole final state is q_6, which depends on q_0 and q_1. The former is not a problem (leaf state) but the latter depends on both q_3 and q_4 – they are sure requirements. We have $q_3 \equiv_{\mathcal{A}} q_4$, but $\mathfrak{Sup}(q_3) = \{a_3\}$ and $\mathfrak{Sup}(q_4) = \{a_2, a_5\}$. Therefore q_1 is Σ-spurious. Remove q_1 from this TAGED, and q_6 becomes unreachable: without any final state, the automaton accepts the empty language.

Let us take another example:

```
TAGED 'example 2' [44] = {
  states = #6{q0, q1, q2, q3, q4, q5}
  final = #1{q5}
  rules = #11{
    a2()->q2, a2()->q3, a3()->q0, a3()->q3, a3()->q4, a4()->q2,
    a5()->q4, g3(q5, q0)->q5, g4(q1, q1)->q5, h1(q2, q3, q4)->q1,
    h3(q2, q3, q4)->q1
  }
  ==rel = #3{(q1,q3), (q3,q1), (q5,q5)}
}
```

Here again, we have only one final state q_5, which depends on q_1. There are two rules which generate q_1, but as it happens both of them have q_3 in their antecedents. We have $q_1 \equiv_{\mathcal{A}} q_3$, and thus those two rules are spurious. The state q_1 becomes unreachable, and consequently so does q_5. The TAGED recognises the empty language.

And these are in no way isolated cases: the method used to generate those examples reports that 13.5% of random (height-driven generation) TAGEDs of height 3 – such as those examples – 24.5% of TAGEDs of height 6 and 30.7% of TAGEDs of height 20 are non-empty when reduced but empty once cleaned up.

All in all, the *cleanup* operation is a valuable, cheap tool which can greatly simplify and speed up the operation of more expensive algorithms on TAGEDs – such as emptiness decision – and, in the context of this report, enables us to weed out many uninteresting randomly generated cases.

5 Generating raw TAGEDs

In this section we present the algorithm for generating raw (*ie.* not cleaned up) random positive TAGEDs. This operation is divided in two parts which are dealt with in two

subsections: first and foremost, the generation of the underlying bottom-up tree automaton, which is by far the most problematic part, and, secondly, the generation of the global equality constraints.

5.1 Generating the underlying tree automaton

The current height-driven algorithm evolved out of previous, unsuccessful attempts, each of which highlighted some of the difficulties listed in section 3[p15]. Although the object of this section is clearly to present the last, successful scheme, the first subsection briefly sketches the ideas and outlines the shortcomings of the previous schemes. This serves to introduce the main axes of the last scheme, and provides a basis for comparison of the experimental results presented in section 6[p37].

5.1.1 Related work and previous attempts

Related work Random generation of non-deterministic bottom-up tree automata has, to our knowledge, not yet been covered in the literature. However, work has been done on generating non-deterministic finite automata [Tabakov and Vardi, 2005] and deterministic top-down tree automata [Héam et al., 2009]. The former, in particular, has proven to be a successful random generation scheme for word automata, in the sense that it was used to test the efficiency of new algorithms on word automata in several articles besides the one which introduced it. The sketch of a random generation scheme for tree automata inspired by [Tabakov and Vardi, 2005] is also found in [Bouajjani et al., 2008]. Let us summarise the ideas of those schemes:

- In [Tabakov and Vardi, 2005], the authors introduced a probabilistic model for random generation of Nondeterministic Finite Automata (NFA), focused on the universality problem. Roughly, in order to generate a NFA $(\Sigma, Q, Q_0, F, \delta)$, they choose the alphabet Σ fixed to $\Sigma = \{0, 1\}$, one initial state, an arbitrary number of states $|Q| = 30$ – which can be considered a parameter of the model – and generate the transitions and final states according to the two metrics

  $$r = r_\sigma = \frac{\left|\{(p, \sigma, q) \in \delta\}\right|}{|Q|}, \forall \sigma \in \Sigma \quad \text{and} \quad f = \frac{|F|}{|Q|}.$$

  The value r can be thought of as the expected out-degree of each node of the associated graph, for each symbol σ. They argue that those two metrics r and f, called respectively *transition density* and *final state density*, cover interesting behaviours as they vary. This model has been used for instance in [Wulf et al., 2006], also for the universality problem.

- In [Bouajjani et al., 2008], experiments on random Tree Automata were performed in a manner very similar to [Tabakov and Vardi, 2005], with $|Q| = 20$. In this

context, the authors defined the transition and final state densities as follows:

$$r = \frac{|\Delta|}{\left|\{\, f(q_1,\ldots,q_n) \mid \exists q \in Q : f(q_1,\ldots,q_n) \to q \in \Delta \,\}\right|} \quad \text{and} \quad f = \frac{|F|}{|Q|}.$$

In other words, the transition density is defined as the average number of different right-hand side states for any given left-hand side of a transition rule.

However, some parameters are not explicitly given, such as the alphabet Σ, the total number of generated transition rules, and the way in which transition rules are generated.

First attempts In our first attempts at generating the underlying tree automata for our `TAGED`s, we retained the ideas of transition and final state densities introduced in [Tabakov and Vardi, 2005], which we adapted to the context of tree automata.

In the first probabilistic model, we generate a random Tree Automaton by first taking a fixed alphabet $\Sigma = \{\, a, b, c/_0, f, g, h/_2 \,\}$, and a number of states $|Q|$, which is a parameter of our model. Considering that a rule $f(q_1,\ldots,q_{arity(f)}) \to q \in \Delta$ is nothing more than a tuple $(f, q_1,\ldots,q_{arity(f)}, q) \in \Sigma_{arity(f)} \times Q^{arity(f)+1}$, we have

$$\Delta \subseteq \overline{\Delta} \qquad \text{with} \qquad \overline{\Delta} \stackrel{\text{def}}{=} \biguplus_{k\in\mathbb{N}} \Sigma_k \times Q^{k+1},$$

and we shall determine Δ by choosing each rule in the space of all possible rules $\overline{\Delta}$ with probability p_Δ (the transition density), another parameter of the model. Lastly, the final states of F are chosen in the same way: each state $q \in Q$ becomes final with probability p_F (the final state density).

This model has several weaknesses:

- The generated automata are very dense. This makes them quite unrealistic, as real-world automata are generally sparse.
- Furthermore, the idea of selecting transition rules uniformly in the space of all possible transition rules has a drawback: rules pertaining to symbols of high arities are overly represented, simply because there are many more of them in $\overline{\Delta}$. For instance, if we added another symbol $\sigma/_{10}$ (of arity 10) to the alphabet Σ, then the number of rules of the form $\sigma(p_1,\ldots,p_{10}) \to q$ would completely dwarf that of other rules, for any reasonable value of $|Q|$.
- The generated automata have lots of dead branches, which means that only a small fraction of the generated rules and states will actually be useful; the rest will be removed by the cleanup operation. Furthermore, they are typical "soup-blenders" and "Frankensteins", as described in section 3[p15], which compromises both realism and difficulty.

The second model corrects the first two of those flaws. In order to generate a tree automaton, the rules are still selected in $\overline{\Delta}$, but this time instead of using a fixed probability p_Δ, we make p_Δ a function of the arity of the rules. Furthermore, the density of the automaton is expressed in terms of the *expected in-degree* δ, which we define as the expected number of rules which yield q, for any state $q \in Q$. Thus we define [(g)]

$$\forall k \in \mathbb{N}, \quad p_\Delta(k) = \begin{cases} \dfrac{\delta}{|\mathfrak{Ar}_\Sigma| \cdot |\Sigma_k| \cdot |Q|^k} & \text{if } \Sigma_k \neq \varnothing \\ 0 & \text{if } \Sigma_k = \varnothing \end{cases}.$$

We build Δ in the same way as before, with this new definition: each rule $f/_n(p_1, \ldots, p_n) \to q \in \overline{\Delta}$ is selected with probability $p_\Delta(n)$. This addresses our two first concerns:

- The generated automata are not too dense; it is easy to see that the expected number of transition rules is $|\Delta| = \delta\,|Q|$, which evolves linearly with $|Q|$ regardless of our choice of Σ.
- There is no explosion in the number of rules of higher arities. The generated rules are evenly distributed among the arities represented in Σ; *ie.* there are an expected $\delta|Q|/|\mathfrak{Ar}_\Sigma|$ rules of each arity.

However the generated automata are still "soup-blenders", and generally dramatically easy instances of the emptiness problem. Let us compute, for instance, the probability that such an automaton accepts at least one leaf-term. Let us denote L the expected number of generated leaf-rules – that is to say, rules of arity zero: since the rules are distributed evenly across all arities, we have

$$L = \frac{\delta\,|Q|}{|\mathfrak{Ar}_\Sigma|}.$$

By definition, the probability that a given leaf-rule is final if p_F. We are looking for the probability that "there exists a final leaf-rule", *ie.* "*not* all leaf-rules are *non*-final". So the probability P which we are looking for is

$$P = 1 - (1 - p_F)^L = 1 - (1 - p_F)^{\frac{\delta|Q|}{|\mathfrak{Ar}_\Sigma|}}.$$

For the value of Σ which we have chosen, we have $|\mathfrak{Ar}_\Sigma| = |\{0, 2\}| = 2$ and let us take, for the sake of example, $\delta = 2$, and $p_F = \frac{1}{5}$. Then we have $P = 1 - (\frac{4}{5})^{|Q|}$. In table 3[p31] we compute the values of $|Q|$ required to achieve certain key values of P: We see that, even for relatively low values of $|Q|$, the chances that the generated automaton accepts some leaf is overwhelming. Since the equality constraints are irrelevant to the

[(g)] The definition of $\mathfrak{Ar}_\Sigma$ can be found at the beginning of section 2.1.1[p7].

P	0.5	0.75	0.9	0.99	0.999
$\lvert Q \rvert$	3	6	10	20	30

Table 3 — Probability of final leaf-rules

recognition of leaf-terms (one needs at least two distinct positions in the term for them to apply), this means that for, say, $|Q| = 20$, generated `TAGED`s using such underlying tree automata stand *at the very least* a 99% chance of being non-empty, and linearly detectable as such by a brute-force algorithm. Furthermore, there are still dead branches, because no provision is taken to avoid them.

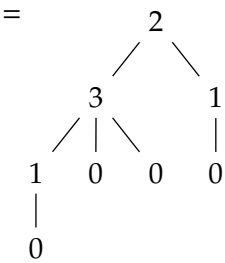
The third model, which we call *skeleton-driven*, takes a completely different approach: instead of reasoning solely on aspects of the automata, such as transition and final states ratios, as was done in the two previous schemes, we begin focusing on properties of their accepted language. Recall that an essential property *wrt.* to ensuring difficulty is the *minimum accepted height*; to ensure that this parameter is sufficient, the idea under this model is to first generate "what the accepted language must look like", and then to generate suitable transition rules.

This model uses a larger alphabet: we denote

$$\Sigma^n \stackrel{\text{def}}{=} \{ a_1, \ldots, a_n/_0, f_1, \ldots, f_n/_1, g_1, \ldots, g_n/_2, h_1, \ldots, h_n/_3 \},$$

and in the experiments, five symbols were used for each arity, so the tree automata use the alphabet Σ^5. To generate a random tree automaton, we first generate a (relatively small) number of *skeletons*, that is to say, trees without symbols [(h)], within certain parameters of width, height and arities. For instance the following tree t_s

$t_s =$



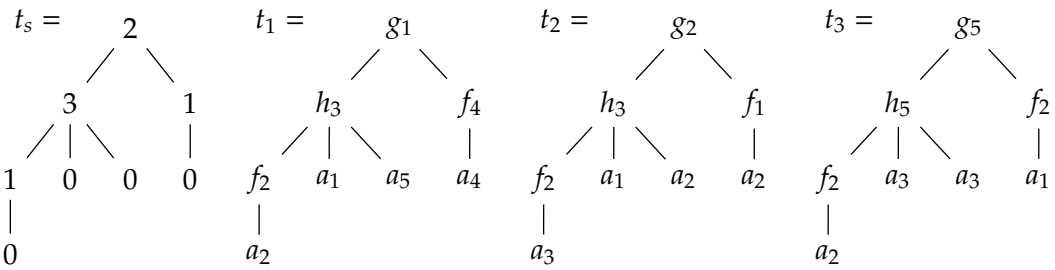
is a skeleton, of height and width equal to 4, using arities 0, 1, 2 and 3. The exact method used to generate those skeletons is of little import here. Then, for each skeleton, transition rules which accept terms isomorphic to it are generated using the following recursive function (`OCaml`):

[(h)] Though in practice we found it convenient to store the arity of each node in the node itself, so they can be seen as trees over the alphabet $\Sigma = \{ n/_n \mid n \in \mathbb{N} \}$.

```
let conversion δ skel =
  let Δ = ref Δ.∅ in
  let make_rules ar [q1,...,qn] q m = for k = 1 to m do
    let σ = gene_symbol ar in Δ.↩ (σ,[q1,...,qn],q) Δ
  done in let rec f = λ
    | Leaf 0 →
      let qx = fresh_state() in make_rules 0 ∅ qx δ; return qx
    | Node (ar, subs) →
      let qx = fresh_state() and [q1,...,qn] = ℒ.map f subs in
      make_rules ar [q1,...,qn] qx δ; return qx
  in let head = f skel in (!Δ, head)
```

- Parameter δ is similar to the δ from generation 2, where it denoted the average in-degree of the states, in that it determines the (maximum) number of rules generated for each state. In practice we have chosen $\delta = 2$.
- Function **fresh_state**: generates a new state for use in the automaton, never used before – hence *fresh*.
- Function **gene_symbol** k : returns a symbol in Σ_k^5, uniformly at random.

Note that the algorithm yields a couple (Δ, q_h), where Δ is the set of rules corresponding to the skeleton, and q_h is the "head state", that is to say the state to which the accepted terms evaluate. For instance, given the skeleton t_s, this algorithm might generate rules such that the terms, say, $t_1, t_2, t_3 \to_\Delta^* q_h$:



The state q_h is intended to be used as a final state of the final automaton. So, to sum it up, tree automata are generated thusly:

1. First, a number of random skeletons are generated within some constraints of height, maximum width etc. Let us denote $S = \{ s_1, \ldots, s_n \}$ the set of our skeletons.
2. Second, each skeleton is converted into a set of transition rules using the above *conversion* function. We obtain a set of rule sets $R = \{ \Delta_1, \ldots, \Delta_n \}$ and, if necessary, make it so that the states used each Δ_i do not appear in any Δ_j, for $i \neq j \in [\![1, n]\!]$.

3. All those rules are put together, *ie.* $\Delta = \bigcup R$, and we extract the set Q of all states which appear in at least one of the rules of Δ. As for the final states, they are taken to be the "head states" returned by the conversion function.

Note that, unlike previous generations, the generated automata are already completely reduced – it is indeed clear that all the states are reachable, and there are no useless states – as defined in theorem 14[p21], as all states actually serve to build a final state. It also clear that it completely solves the "minimum height" problem. However, this method generates tree automata that are clear cases of "waffle irons", as defined in section 3[p15], which compromises realism and, as it turned out, difficulty as well. Indeed, one of our heuristics for deciding emptiness made those cases trivial. For these reasons, skeleton-driven generation is not used on its own, but only as a building-block for hight-driven generation.

5.1.2 Outline of the height-driven algorithm

The height-driven approach is something of a hybrid of previous generations, notably the second and third (skeleton-driven). While it does not continue using skeletons, the idea of fixing the minimum height of recognised terms remains, as it was essential to avoid the pitfalls of generation 2. However, the aim was also to avoid having too rigid a structure, as opposed to generation 3.

So, height-driven generation is focused on two main parameters: the minimum height of terms, and the aspect of the set of rules. This time, we tried to achieve "difficult" cases by enabling the generation of all kinds of rules, for instance rules with immediate cycles ($f(\dots,q,\dots) \to q$), repetitions of the same state ($f(\dots,p,\dots,p,\dots) \to q$), and also rules of the form $f(\dots,p,\dots) \to q$, where p is an "old" state, as opposed to a freshly generated state. None of these kinds of rules could be generated by the third generation: all generated rules were of the form $f(p_1,\dots,p_n) \to q$, where the p_k were all distinct, and the terms in $\mathcal{L}\mathrm{ng}(\mathcal{A},p_k)$ were all exactly of the same height h, and those in $\mathcal{L}\mathrm{ng}(\mathcal{A},q)$ were of height $h+1$. As for the second generation, such rules *could* have been produced, but the odds against them actually coming into play were overwhelming, given the huge proportion of simple cases (leaf terms etc) which was characteristic of the second generation. Another requirement was that for each q, the signatures of the rules of $\mathfrak{Rul}(q)$ were sufficiently varied. As stated in the previous section, with the third generation, if you have a rule, say $f(p_1,\dots,p_n) \to q \in \Delta$, then it follows that $\mathfrak{Rul}(q) \subseteq \{\,\sigma(p_1,\dots,p_n) \to q \mid \sigma \in \Sigma_n\,\}$, so any rule $r \in \mathfrak{Rul}(q)$ is such that $\mathfrak{Ant}(r) = \{\,p_1,\dots,p_n\,\}$. This property was not representative of real world tree automata, and made the automata trivial instances of one of our algorithms for testing emptiness. We also wanted to avoid generating too many useless branches, which would be cut off by trivial observations; that is to say, we wanted to keep the ratio R, as defined for the second generation, to a low value. And lastly, we wished to keep the automata to a reasonable size, focusing on generating "difficult" instances, rather than "large" ones.

In short, the aim was to keep most of what we found interesting in the previous generations, without any of their shortcomings. Fortunately, the approach which we will now discuss seems to manage that.

The height-driven generation algorithm is outlined in figure 2[p34]. Note that we are working with the same alphabet as the third generation, namely $\Sigma = \Sigma^5$. Of course, this

```
Data: minimum height, a number of other parameters
Result: a random tree automaton
begin
    pool ← head states from (small) skeleton-driven [i] generation;
    Δ ← rules from skeleton-driven generation for pool;
    while minimum height not reached do
        q ← fresh state;
        δ ← random number of rules;
        for δ times do
            n ← random arity;
            σ ← random symbol in Σ_n;
            purge too old states from pool;
            p_1, ..., p_n ← random states from pool;
            add rule σ(p_1, ..., p_n) → q to Δ;
            add q to pool;
        endfor
    endw
    F ← random states in pool;
    return tree automaton based on Δ and F;
end
```

Figure 2 — Rough outline of height-driven generation

outline leaves many things in the dark: for instance, each time something is selected at "random", one can wonder about the exact implementation of these random selections:

- Random number of rules and arity are selected with a certain discrete probability distribution, which is hard-coded in the algorithm using the *w_choice* function described below.
- Random symbols in Σ_n are selected uniformly.
- Random states from the pool are selected according to a discrete probability distribution which is itself a function of the minimum height of terms which evaluate to q. The distribution is biased to favour states which recognise bigger terms. This

[i] Or potentially any other generation scheme.

is an important idea of the algorithm, which allows the generated automata to keep an acceptable size while allowing enough variety in their rules.

- Old states to be purged from the pool are states whose associated minimum height has become too low compared to the greatest associated height which has been generated. In other words, if you are currently building the top of the tree, you avoid reusing states from the bottom of the tree. The degree of tolerance for old states is called *cohesion*, and is a parameter of the procedure. Tighter cohesion means smaller, more focused automata.

Let us say a few more word about how the "pool" works. What we call pool is the set of states for which we have generated rules so far. Initially, we create a few sets of rules accepting small terms using the third generation. We store those rules in Δ, and put the head states [(j)] of those rule sets into the pool. For each q in the pool, we also keep track of the minimum height of terms which evaluate to q with respect to Δ. Let us denote it by $m(q)$. Since the skeleton-driven generation is directed by the height (among other things), this quantity is known for the generated head states. Let us denote the pool by $P = \{p_1, \ldots, p_n\}$, and let $\mathbb{b} : \mathbb{N} \to \mathbb{N}$ be a function which we will call "bias". Then, when we want to get states out of the pool, we select a random state X following the probability mass function:

$$\forall i \in [\![1, n]\!], \quad \mathbb{P}(X = p_i) = \frac{\mathbb{b} \circ m(p_i)}{\sum_{k=1}^{n} \mathbb{b} \circ m(p_k)}.$$

The implementation of such a choice is straightforward:

```
let w_choice wlist =
  let ⊥,weights = ℒ.split wlist in
  let totalW = ℒ.fold→ (+) weights 0 in
  let rec f dart = λ
    | (item,w) :: tl → if dart ⩽ w then item else f (dart − w) tl
  in λ()→f (1 + Random.int totalW) wlist
```

The call of **w_choice** $[(x_1, w_1), \ldots, (x_n, w_n)]$ returns a function which, when called, returns x_i with probability $w_i / \sum_{k=1}^{n} w_k$. The remaining point to discuss is the choice of the bias $\mathbb{b}$; it should be strong enough to favour states with a greater minimum height, but not so much so as to completely forbid the use of older states. We have chosen it to be

$$\mathbb{b}(w) = (w - h + d + c)^2,$$

where

- $h = \max_{p \in P} m(p)$ – in other words, it is the greatest minimum height associated with the states we have generated so far.

[(j)] See previous section

- c is the cohesion value, mentioned higher up. The cohesion requires the property $\forall p \in P : \; h - c \leqslant m(p) \leqslant h$ to be an invariant. Its default value in our experiments was 2.

- d is the "damping". It follows from the cohesion invariant that $\forall p \in P : \; d \leqslant m(p) - h + d + c \leqslant c + d$. The value d is chosen to be (an approximation of) the solution to the equation $(d+2)^2 = 2d^2$. So by taking $d \approx 2(1+\sqrt{2}) \approx 5$, we make it so that states two ranks higher than another have a twice greater chance of being selected than that other state.

Note that actually, for the choice of final states, the bias is stronger than for an ordinary choice: $\mathbb{b}(w) = (w - h + d + c)^4$.

To conclude this description of the *modus operandi* of height-driven generation, let us give the distributions used in our tests for arities and number of rules:

```
18  let new_arity = w_choice [1,2 ; 2,3 ; 3,1] in
19  let new_delta = w_choice [1,70; 2,25; 3,2; 4,1; 5,1; 6,1] in
```

As it turned out, this method generates sufficiently interesting random tree automata for our purposes – we will discuss that in the next part, and satisfies the wishes expressed at in section 3[p15]. See for instance table 4[p38] which shows that the size of generated automata is quite reasonable. Also note that, as in the third generation, the generated tree automata are reduced by design, *ie.* there are no unreachable states.
On a side-note, this generation proved to be useful for generating good "human-readable" examples of TAGEDs satisfying some properties: we implanted a simple procedure which, given a predicate p on TAGEDs and – optionally – a comparison function $\prec$ between TAGEDs, yields the best ($\prec$-wise) random TAGED satisfying p which it could find in reasonable time. Height-driven random TAGEDs seem to be varied enough that this approach works quite well, so long of course as p and $\prec$ remain easy to compute. For instance the example automata of section 4.2[p26] were generated using this method.

Also note that this algorithm depends on many parameters, some of which are discrete probability distributions, which can be changed to reflect data from the real world, should such data be available. For instance, statistics on, say, tree automata taken from model-checking applications could well replace the arbitrary distributions that we have been using, which are based mostly on our former, hand-written example TAGEDs.

5.2 Generating the global equality constraints

Generating the global equality constraints turned out to be far less problematic than generating the underlying tree automata. Several schemes were tried, the first of which had the number of constraints grow linearly with the size of the underlying tree automata. However we observed that the resulting TAGEDs did tend to become more and more frequently empty as their size grew, which we took to mean that there were

too many constraints. Furthermore, it is legitimate to suppose that real-world TAGEDs would not necessarily see the numbers of constraints grow linearly in their size. For instance, if a TAGED is the result of transformation effected under rewriting rules, the number of constraints could remain constant, though the underlying tree automaton becomes increasingly large. As a compromise, we fell back on a number of constraints logarithmic in $|Q|$. We believe this choice does not affect the complexity class in which emptiness falls – *ie.* even with a logarithmic number of constraints, the problem remains *EXPTIME*-complete. However some work would be needed in order to confirm this intuition. The algorithm to generate the constraints is exceedingly simple, and given

```
Data: set of states Q
Result: a set of random constraints
begin
    Csts ← ∅;
    for max(1, log10 |Q|) times do
        q, p, p' ← random states in Q (uniform);
        add { (q, q), (p, p'), (p', p) } to Csts;
    endfor
    return Csts;
end
```

Figure 3 — Very simple constraints generation algorithm

in the figure 3[p37]. Note that we have a bias towards diagonal constraints: roughly half the constraints are diagonal. Indeed, we observed that diagonal constraints seemed to come up very frequently in practice. It would of course not be advisable to generate too many diagonal positive TAGEDs because those can be decided in linear time [Filiot et al., 2008], but having a positive bias towards diagonal constraints seemed to serve realism without lessening the difficulty of the generated instances.

In practice, this simple model proved to be sufficient, and used in conjunction with height-driven generation of random tree automata, kept some balance between emptiness and non-emptiness of the resulting TAGEDs, which is of course what we aimed at.

6 Experiments

This section presents some experimental results obtained using the random generation scheme. Figure 4 shows statistics on the size of the randomly generated TAGEDs, using the default parameters; there are two interesting points. First, the number of states used evolves linearly with the minimum accepted height, and so does the overall size of the automata. Secondly, the generated automata are rather sparse: the average number of

Height	$\|Q\|$	$\|\|\mathcal{A}\|\|$	$\|\|\mathcal{A}\|\| / \|Q\|$	$\|\Delta\|$	$\|\Delta\| / \|Q\|$
4	6.89	43.49	6.31	11.30	1.64
10	18.14	119.84	6.61	27.12	1.50
16	29.58	196.94	6.66	43.13	1.46
22	41.31	276.70	6.70	59.67	1.44
28	52.58	353.26	6.72	75.47	1.44
34	64.47	434.65	6.74	92.36	1.43
40	75.38	507.81	6.74	107.55	1.43
46	87.00	588.54	6.76	124.14	1.43
52	99.45	672.86	6.77	141.87	1.43
58	110.41	745.74	6.75	156.70	1.42
64	122.41	826.10	6.75	173.27	1.42
70	133.68	903.50	6.76	189.26	1.42
76	145.09	981.29	6.76	205.39	1.42

Table 4 — Height-driven generation: size statistics

rules per states converges towards the relatively low value of 1.41. This is not surprising as it is the expected value of the random variable δ, given the probability distribution used in section 5[p27]. So they satisfy our realism constraints regarding density and size.

What remains to be seen is whether they satisfy our constraints regarding difficulty of the generated instances *wrt.* the emptiness problem. Figures 5 and 6 present tests performed

$\|Q\|$	Run ρ	$\mathcal{L}ng(\mathcal{A}) \neq \emptyset$	$\mathcal{L}ng(\mathcal{A}) = \emptyset$	Failure
4.	26.8%	73.2%	0.0%	0.0%
7.	43.6%	55.6%	0.8%	0.0%
10.	48.8%	50.8%	0.4%	0.0%
13.	49.2%	50.8%	0.0%	0.0%
16.	50.0%	50.0%	0.0%	0.0%
19.	42.4%	57.6%	0.0%	0.0%
22.	41.2%	58.4%	0.4%	0.0%
25.	34.8%	65.2%	0.0%	0.0%
28.	30.4%	69.6%	0.0%	0.0%
31.	36.4%	63.6%	0.0%	0.0%
34.	38.8%	61.2%	0.0%	0.0%
37.	35.6%	64.4%	0.0%	0.0%
40.	28.0%	72.0%	0.0%	0.0%

Table 5 — "Soup blender" typical results

with decision procedures for the emptiness problem which we have developed (which are beyond the scope of the report), for the second and last (height-driven) random

generation schemes, respectively. The first column corresponds to the main parameter

min H	Run ρ	$\mathcal{A} \neq \emptyset$	$\mathcal{A} = \emptyset$	Failure	$\prec$
6	0.4%	69.6%	28.8%	1.2%	2.8%
9	0.4%	69.2%	25.6%	4.8%	6.4%
12	0.0%	55.6%	36.4%	8.0%	9.2%
15	0.0%	61.2%	26.4%	12.4%	7.6%
18	0.0%	53.2%	30.0%	16.8%	6.4%
21	0.0%	50.8%	30.0%	19.2%	8.8%
24	0.0%	46.8%	35.6%	17.6%	7.2%
27	0.0%	45.6%	31.2%	23.2%	5.6%
30	0.0%	45.2%	31.2%	23.6%	6.8%
31	0.0%	50.8%	25.2%	24.0%	6.0%
34	0.0%	50.8%	26.8%	22.4%	6.4%
37	0.0%	43.6%	26.8%	29.6%	7.2%

Table 6 — Height-driven generation: results

of the random generation algorithm in use: in the case of the second generation, this is $|Q|$, while in that of height-driven generation, it is the minimum accepted height. The other columns correspond to different outcomes of the algorithm.

- **Run ρ** : no heuristic could be used to decide quickly, and the procedure fell back on a brute-force algorithm, which found an accepting run (and the corresponding recognised term).
- $\mathcal{A} \neq \emptyset$: The language accepted by the automaton was found to be non-empty using one of our fast approaches (and *not* the brute-force algorithm).
- $\mathcal{A} = \emptyset$: The language accepted by the automaton was found to be empty, either because a heuristic was able to conclude quickly or because the brute-force algorithm terminated without finding an accepted term. In practice, that latter case would happen with infinitesimal probability, so it was not isolated, as having a column full of zeroes is not useful...
- **Failure**: The efficient approaches were tried and failed. The decision procedure then fell back on the brute-force algorithm, which also failed to give any answer (timeout). Emptiness of the automaton is unknown.
- $\prec$ (figure 5 only): sub-case of non-emptiness concerning one of our heuristics.

Observe that in figure 5, the decision procedure *never fails*, and the language accepted by $\mathcal{A}$ is almost always non-empty. Furthermore, while the efficient heuristics do not

always suffice to conclude, the brutal algorithm succeeds whenever it is used. These results are of course useless to evaluate the efficiency of our approaches, and it is results such as these that motivated the development of more interesting random generation schemes. Case in point, the results of figure 6, using the height-driven scheme, are much more varied.

- There are both empty and non-empty cases, and in both cases the proportions are reasonable, that is to say, no outcome dwarfs the other.
- There is an increasing, and sizeable, rate of failure.
- The brutal algorithm fails whenever it is invoked, except for the smallest cases, and even then, it is only very marginally successful.

Thus, the height-driven random scheme is useful as a benchmark for the efficiency of our decision procedures for the emptiness problem, as it leaves room for improvement (there are many cases of failures) and discriminates between efficient and inefficient approaches (our heuristics have decent scores, while the brutal algorithm is of little help; which was to be expected).

7 Conclusion and future work

In this report, we outlined a random, height-driven generation scheme designed to yield positive `TAGED`s which are interesting (that is to say, difficult and realistic) instances of the emptiness decision problem. This scheme is meant to be used as the basis for a pertinent experimental evaluation of the efficiency of new approaches for deciding emptiness for positive `TAGED`s.

The different aspects of this random generation are documented in detail, and the pitfalls which we identified are illustrated by brief presentations of previous, unsatisfactory schemes whose shortcomings were purposefully avoided by the height-driven scheme. Furthermore, in order to generate more realistic instances, we introduced a "*cleanup*" operation which prunes dead branches from the generated automata and discards obviously uninteresting cases.

Future work: Although designed with the emptiness decision problem in mind, the height-driven generation scheme could be adapted for other decision problems. For instance, coupled with a terms generator, it could be used to evaluate approaches for deciding membership. We plan on using it to improve the significance of our experimental results in [Héam et al., 2010]. Moreover, we intend to improve the usefulness of the *cleanup* through both the addition of new cases and the extension of existing ones.

References

[Abiteboul et al., 2009] Abiteboul, S., Segoufin, L., and Vianu, V. (2009). Modeling and verifying active xml artifacts. *IEEE Data Eng. Bull.*, 32(3):10–15.

[Barguñó et al., 2010] Barguñó, L., Creus, C., Godoy, G., Jacquemard, F., and Vacher, C. (2010). The emptiness problem for tree automata with global constraints. In *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science (LICS'10)*, pages 263–272, Edinburgh, Scotland, UK. IEEE Computer Society Press.

[Boichut et al., 2008a] Boichut, Y., Courbis, R., Héam, P.-C., and Kouchnarenko, O. (2008a). Finer is better: Abstraction refinement for rewriting approximations. In *Rewriting Techniques and Application, RTA'08*, volume 5117 of *Lecture Notes in Computer Science*, pages 48–62. Springer.

[Boichut et al., 2007] Boichut, Y., Genet, T., Jensen, T. P., and Roux, L. L. (2007). Rewriting approximations for fast prototyping of static analyzers. In Baader, F., editor, *RTA'07*, volume 4533 of *Lecture Notes in Computer Science*, pages 48–62. Springer.

[Boichut et al., 2008b] Boichut, Y., Héam, P.-C., and Kouchnarenko, O. (2008b). Approximation-based tree regular model-checking. *Nordic Journal of Computing*, 14:194–219.

[Boichut et al., 2009] Boichut, Y., Héam, P.-C., and Kouchnarenko, O. (2009). Tree automata for detecting attacks on protocols with algebraic cryptographic primitives. *ENTCS*, 239:57–72.

[Bojanczyk et al., 2009] Bojanczyk, M., Muscholl, A., Schwentick, T., and Segoufin, L. (2009). Two-variable logic on data trees and XML reasoning. *J. ACM*, 56(3).

[Bouajjani et al., 2008] Bouajjani, A., Habermehl, P., Holík, L., Touili, T., and Vojnar, T. (2008). Antichain-based universality and inclusion testing over nondeterministic finite tree automata. *International Conference on Implementation and Applications of Automata*, pages 57–67.

[Comon et al., 2007] Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., and Tommasi, M. (2007). *Tree Automata Techniques and Applications*. release October, 12th 2007.

[Comon-Lundh et al., 2008] Comon-Lundh, H., Jacquemard, F., and Perrin, N. (2008). Visibly tree automata with memory and constraints. *Logical Methods in Computer Science*, 4(2).

[Courbis et al., 2009] Courbis, R., Héam, P.-C., and Kouchnarenko, O. (2009). Taged approximations for temporal properties model-checking. In Maneth, S., editor, *In-*

ternational Conference on Implementation and Applications of Automata, volume 5642 of *Lecture Notes in Computer Science*, pages 135–144. Springer.

[Filiot, 2008] Filiot, E. (2008). *Logics for n-ary queries in trees.* PhD thesis, Université des Sciences et Technologie de Lille - Lille I.

[Filiot et al., 2008] Filiot, E., Talbot, J.-M., and Tison, M. S. (2008). Tree automata techniques and applications (slides). www.lsv.ens-cachan.fr/Events/HubertMedaille/Slides/STison.pdf.

[Filiot et al., 2008] Filiot, E., Talbot, J.-M., and Tison, S. (2008). Tree Automata with Global Constraints. In *12th International Conference on Developments in Language Theory (DLT)*, pages 314–326, Kyoto Japon.

[Héam et al., 2010] Héam, P., Hugot, V., and Kouchnarenko, O. (2010). SAT Solvers for Queries over Tree Automata with Constraints. In *Third International Conference on Software Testing, Verification, and Validation Workshops*, pages 343–348. IEEE.

[Héam et al., 2009] Héam, P., Nicaud, C., and Schmitz, S. (2009). Random Generation of Deterministic Tree (Walking) Automata. *International Conference on Implementation and Applications of Automata*.

[Jacquemard et al., 2009] Jacquemard, F., Klay, F., and Vacher, C. (2009). Rigid tree automata. In Horia Dediu, A., Mihai Ionescu, A., and Martín-Vide, C., editors, *Proceedings of the 3rd International Conference on Language and Automata Theory and Applications (LATA'09)*, volume 5457 of *Lecture Notes in Computer Science*, pages 446–457, Tarragona, Spain. Springer.

[Jacquemard and Rusinowitch, 2009] Jacquemard, F. and Rusinowitch, M. (2009). Rewrite based verification of XML updates. *CoRR*, abs/0907.5125.

[Murata, 1999] Murata, M. (1999). Hedge automata: a formal model for XML schemata. http://www.horobi.com/Projects/RELAX/Archive/hedge_nice.html.

[Schwentick, 2007] Schwentick, T. (2007). Automata for XML—a survey. *J. Comput. Syst. Sci.*, 73(3):289–315.

[Tabakov and Vardi, 2005] Tabakov, D. and Vardi, M. (2005). Experimental evaluation of classical automata constructions. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 396–411. Springer.

[Wulf et al., 2006] Wulf, M. D., Doyen, L., Henzinger, T. A., and Raskin, J.-F. (2006). Antichains: A new algorithm for checking universality of finite automata. In Ball, T. and Jones, R. B., editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 17–30. Springer.

Centre de recherche INRIA Nancy – Grand Est
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399